INSTRUMENTE de programare NESECVENȚIALĂ

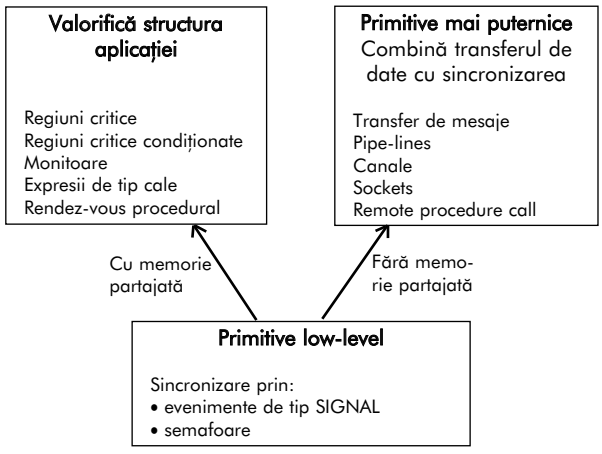
Anca Vasilescu, Oana Georgescu

În acest ultim episod al serialului despre problematica sistemelor nesecvențiale vom lua în discuție mecanismele specifice puse la dispoziție de limbajele de programare pentru implementarea aspectelor paralele, concurente sau distribuite evidențiate la nivelul diferitelor aplicații. Aceste instrumente de programare sunt cunoscute sub denumirea de mecanisme IPC, adică mecanisme de comunicare între procese (InterProcess Communication).

Tratarea mecanismelor *IPC* (de comunicare între procese) valorifică toate aspectele problematicii sistemelor nesecvențiale introduse în episoadele anterioare: probleme clasice specifice, clasificări, caracteristici.

Pentru programarea sistemelor paralele, concurente sau distribuite s-au dezvoltat o serie de mecanisme de implementare a concurenței la nivel *low-level* sau la nivelul limbajelor de programare de nivel înalt. Elementele discutate în acest context se vor axa pe prezentarea mecanismelor de comunicare între procese (*IPC - InterProcess Communication*), la diferite niveluri de abordare.

În figura următoare este reprezentată evoluția mecanismelor și primitivelor *IPC*.



Variantele *IPC* cu sau fără memorie partajată se regăsesc în două modalități de abordare a comunicării și anume: procesul activ poate să apeleze o procedură de interfață a unui modul pasiv (monitor), poate să trimită un mesaj în care formulează o cerere specifică. Aceste alternative se încadrează într-o clasificare mai generală a mecanismelor *IPC* de nivel înalt:

- mecanisme *IPC* pentru transmiterea de mesaje (*message passing*)
  - ♦ transmitere *asincronă* sau cu zone tampon;
  - ♦ transmitere *sincronă* sau fără zone tampon;
- mecanisme *IPC* procedurale
  - ♦ apel de procedură pentru obiecte pasive (se folosesc monitoare);
  - ♦ apel de procedură pentru obiecte active (mecanismul *rendez-vous*).

O clasă superioară de mecanisme *IPC* este dată de modalitățile de comunicare distribuită între procese (*Distributed IPC*). Prezența lor este necesară în sistemele *hard* și/sau *soft* distribuite și, implicit, trebuie privite și analizate prin prisma structurilor sistemelor pe care le deservesc. Din acest punct de vedere trebuie remarcat modelul *client-server* de implementare a aplicațiilor distribuite, care este în strânsă legătură atât cu suportul fizic și logic de comunicare disponibil în sistem, cât și cu limbajul pe care sistemul respectiv îl suportă. O alternativă la abordarea *client-server* este modelarea obiectuală a sistemelor. Aceasta aplică tehnologia programării orientate obiect la nivelul sistemelor prin interpretarea fiecărei resurse partajate ca un obiect [1].

Dintre mecanismele enumerate în clasificarea anterioară le vom lua în discuție pe cele mai bine reprezentate în limbajele de programare moderne și anume: *semafoarele*, *monitoarele*, *comunicarea prin transfer de mesaje* și *invo-*

carea la distanță (ca o generalizare a mecanismului de apel de procedură la distanță, *RPC*).

Semafoare

Noțiunea de *semafor* a fost introdusă în 1968 de ***Dijkstra*** împreună cu descrierea proiectului de sistem de operare numit ***THE*** (***T**echnische **H**ogeschool **E**indhoven*). Sistemul *THE* proiectează o ierarhie de resurse virtuale gestionată de un sistem de procese strict organizate pe niveluri. [5]

Așa cum a fost introdus de *Dijkstra*, un ***semafor*** este un tip de date. Asupra unei variabile de tip semafor se pot executa operațiile `WAIT(SEMAFOR)` și `SIGNAL(SEMAFOR)` numite în acest context și ***primitive***, deoarece sunt definite ca operații atomice. În plus, asupra semafoarelor este permisă efectuarea unei singure operații, și anume inițializarea valorii variabilei semafor. Sintaxa acestor operații depinde de limbajul de programare în care se lucrează cu semafoare.

Reprezentarea specifică a tipului semafor este dată printr-un număr natural (întreg pozitiv) și o coadă atașată operației `WAIT(SEMAFOR)`, dar despre disciplina de coadă nu se precizează nimic [4].

Dacă valorile pe care le poate lua o variabilă semafor sunt numai 0 și 1, atunci semaforul este unul binar. Cele două tipuri de date, ***semafoarele generale*** și ***semafoarele binare***, sunt echivalente [4].

Definițiile operațiilor cu semafoare sunt următoarele:

- `WAIT(SEMAFOR)`:

  dacă `SEMAFOR > 0`
  atunci `SEMAFOR ← SEMAFOR - 1`
  altfel *blocare proces*

- `SIGNAL(SEMAFOR)`:

  dacă există *procese blocate*
  atunci *deblocare un proces*
  altfel `SEMAFOR ← SEMAFOR + 1`

Dijkstra a notat operația `WAIT()` cu `P(SEMAFOR)` și `SIGNAL()` cu `V(SEMAFOR)`, unde `P` și `V` sunt inițialele cuvintelor olandeze: *passeren* (a trece) și, respectiv, *vrygeven* (a elibera). De aici este imediată semnificația celor două operații: operația `WAIT()` permite trecerea procesului pe la semafor, dacă valoarea semaforului este nenulă și, respectiv, blocarea procesului la acel semafor, dacă valoarea lui este zero.

Pentru alegerea procesului care va fi deblocat de operația `SIGNAL()` programatorul poate implementa propria politică de deblocare. Aceasta poate fi aleatoare sau poate respecta mecanismul de lucru cu structuri de tip coadă, coadă cu prioritate ș.a.m.d. Evident, alegerea aleatoare poate amâna la nesfârșit deblocarea unui anumit proces, afectând vivacitatea sistemului.

Realizarea excluderii reciproce cu ajutorul semafoarelor

Fiecare dintre operațiile `WAIT(SEMAFOR)` și `SIGNAL (SEMAFOR)` este o operație atomică, ceea ce permite semafoarelor să fie utilizate cu succes în rezolvarea problemei excluderii mutuale. Concret, atomicitatea operației `WAIT (SEMAFOR)` înseamnă că succesiunea de operații interne *comparare* + *decrementare* este indivizibilă. De asemenea, pentru operația `SIGNAL(SEMAFOR)` succesiunile *comparare* + *deblocare* sau *comparare* + *incrementare* sunt indivizibile.

În timp ce două operații asupra aceluiași semafor se execută prin excludere reciprocă, este posibilă suprapunerea executării a două operații asupra unor semafoare diferite.

Algoritmul de rezolvare a problemei excluderii reciproce folosind un singur semafor `sem` este următorul:

```
repetă la infinit
WAIT(sem); // protocol de intrare în secțiunea critică
                execută secțiune critică;
SIGNAL(sem); // protocol de ieșire din secțiunea critică execută secvență necritică;
sfârșit repetă
```

Conform acestui algoritm, prin apelul primitivei `WAIT()` procesul își anunță intenția de a intra în secțiunea critică, iar prin `SIGNAL()` semnalează semaforului terminarea executării secțiunii critice.

Pentru ca semaforul `sem` să rezolve problema excluderii mutuale, valoarea sa inițială trebuie să fie 1. [4]

Exemple de utilizare a semafoarelor

Utilizarea semafoarelor este o modalitate de lucru folosită cu succes în rezolvarea excluderii mutuale a proceselor concurente, în sincronizarea proceselor cooperative, în instanțierea multiplă a unei resurse partajate.

Exemple

Fie o intersecție semaforizată a unor artere de circulație rutieră. În limbajul sistemelor concurente se poate spune că intersecția este o resursă critică partajată de procesele-mașini care încearcă să treacă prin intersecție, adică să folosească în comun, eventual chiar simultan, resursa critică-intersecție. Rolul semaforului din intersecție este *jucat* în programarea sistemului concurent de o variabilă de tip `SEMAFOR`. Este vorba de un semafor binar, deoarece el poate avea numai valorile 0 sau 1, respectiv *roșu* sau *verde*.

În acest caz procesele-mașini își anunță intenția de a intra în intersecție printr-o operație `WAIT(SEMAFOR)`. De asemenea, mașinile testează periodic starea semaforului prin apeluri `SIGNAL(SEMAFOR)` pentru a decide când au dreptul să intre în intersecție. Cât timp un semafor din intersecție este verde, nici un alt semafor din aceeași direcție de trafic nu poate fi verde, dar un semafor din altă direcție poate fi verde. Această situație practică asigură accesul exclusiv al mașinilor în intersecție, respectiv al proceselor la utilizarea resursei critice.

Atunci când o mașină încearcă să treacă prin intersecție - adică un proces execută o operație `WAIT(SEMAFOR)` - și găsește semaforul roșu (variabila `SEMAFOR` are valoarea 0) acea mașină va fi blocată la semafor. În continuare, mașina intenționează să plece de la semafor (procesul trebuie să fie deblocat). Pentru aceasta procesul va executa periodic operații `SIGNAL(SEMAFOR)` până în momentul în care semaforul va fi verde (variabila `SEMAFOR` va avea valoarea 1).

Dacă în momentul în care o mașină încearcă să treacă prin intersecție (un proces execută o operație WAIT (SEMAFOR)) și găsește semaforul verde (adică variabila SEMAFOR are valoarea 1) atunci acea mașină va fi liberă să intre în intersecție.

Problema producător-consumator poate fi rezolvată cu două semafoare cu următoarele semnificații: unul numit DATE, a cărui valoare să reprezinte datele din *buffer* la un moment dat (numărul articolelor produse și încă neconsumate) și celălalt numit SPAȚII, a cărui valoare să reprezinte locațiile libere din *buffer* la un moment dat. Valoarea inițială este 0 pentru DATE și numărul total de locații ale *buffer*-ului pentru SPAȚII.

Limitele semafoarelor. Perspective

Dacă presupunem că semaforul este singurul mecanism de tip *IPC* al unui limbaj de programare, atunci pot apărea o serie de probleme, cum ar fi:

- se poate greși ușor în programarea semafoarelor: dacă programatorul uită să folosească apelul WAIT atunci, accidental, poate să acceseze date partajate neprotejate; sau dacă programatorul uită să execute un apel SIGNAL, atunci poate să lase o structură de date blocată indefinit;
- nu este posibil să avem o listă de semafoare ca argument pentru WAIT; dacă această posibilitate ar fi fost acceptată, atunci ar putea fi programate diferite variante de ordonare a acțiunilor, conform cu starea curentă a sosirii semnalelor de tip SIGNAL.
- timpul în care un proces este blocat la un semafor nu este limitat;
- nu există nici un mijloc prin care un proces să poată controla un alt proces.

Ținând cont de toate aceste probleme, este evident că un limbaj de programare destinat aplicațiilor concurente trebuie să dețină și mecanisme de tip *IPC* care să completeze facilitățile oferite de semafoare. Multe dintre aceste inconveniente se elimină dacă situațiile de blocare în așteptare sunt asociate cu posibilitatea de a crea procese-fii care să continue execuția în timp ce părintele rămâne în așteptare.

Potrivit lui ***Bacon*** [2], alternativele pentru semafoare sunt tipurile de date EVENTCOUNT și SEQUENCER, precum și bibliotecile de lucru cu *thread*-uri cum ar fi *thread*-urile *POSIX*. În plus, putem aminti și tipurile abstracte de date mai evoluate cum este *monitor*ul.

Monitoare

Un ***monitor*** are structura unui obiect abstract de date. Datele încapsulate într-un monitor sunt partajate și fiecare operație este executată sub excludere mutuală. Implementarea monitoarelor trebuie să asigure că, în fiecare moment, numai un proces este activ în monitorul respectiv. Monitoarele au avantajul de a asigura excluderea mutuală la nivelul operațiilor interne dar, dacă se impune sincronizarea pe condiție, monitoarele pot folosi și *variabile de condiție*.

Monitoarele corespund modelului de calcul paralel bazat pe memorie partajată. Concret, datele unui monitor sunt partajate (publice).

Conform modelului introdus de ***Hoare*** în [8], un monitor are structura unui obiect de date abstract, care încapsulează date și operații. Datele încapsulate într-un monitor sunt partajabile, iar fiecare operație este executată sub excludere mutuală. Din aceste considerente rezultă imediat trei avantaje importante ale monitoarelor și anume:

- monitoarele introduc posibilitatea modularizării aplicațiilor concurente;
- executarea unei secțiuni critice la nivelul unui proces se poate lansa prin apelul unei metode a unui monitor;
- monitoarele rezolvă prin definiția lor problema excluderii mutuale a accesului la secțiunile critice.

Pe lângă asigurarea excluderii mutuale, pentru rezolvarea problemelor care apar în aplicațiile concurente, trebuie completate mecanismele pentru sincronizarea pe condiție (sincronizarea condiționată). În limbajele care permit efectuarea de operații cu monitoare, această problemă este rezolvată prin introducerea unui tip de date special pentru *variabilele de condiție*. Programatorul declară o variabilă de condiție necesară aplicației și modulul de gestionare a monitoarelor va defini o coadă asociată variabilei respective. Mai mult, gestionarea monitoarelor se referă implicit și la gestionarea acestor cozi astfel încât, împreună cu protocolul de organizare a proceselor care solicită accesul la monitor, să se asigure sincronizarea proceselor cu respectarea condițiilor asociate variabilelor definite.

Asupra variabilelor de condiție pot fi efectuate operațiile WAIT și SIGNAL cu următoarele semnificații: WAIT determină autoîntârzierea procesului care apelează WAIT (VAR_COND) prin depunerea lui în coada asociată variabilei VAR_COND; ulterior, acest proces este eliberat de către un alt proces care execută o operație SIGNAL(VAR_COND) asupra aceleiași variabile de condiție. Dacă există mai multe procese care așteaptă la VAR_COND atunci unul va fi cu siguranță eliberat. Implicit, SIGNAL nu are nici un efect în cazul în care coada asociată variabilei VAR_COND este vidă.

Un semafor se reprezintă printr-un întreg și o coadă, în timp ce o variabilă de condiție se reprezintă numai printr-o coadă de așteptare. De aici rezultă două îmbunătățiri ale limitelor semafoarelor:

- nu se pune problema semnalelor SIGNAL de tip "*wake-up waiting*";
- procesul care execută WAIT este depus necondiționat în coada de așteptare a variabilei de condiție.

Monitoarele sunt structuri pasive care permit proceselor active care le apelează metodele să se succeadă pentru a obține cooperarea și/sau competiția cu alte procese. Prin mecanisme specifice, monitoarele organizează cozi pentru resursele partajate și gestionează *buffer*-e de comunicare între diferite tipuri particulare de procese active (de exemplu, producător - consumator, cititor - scriitor).

Monitoarele reprezintă modelul de sincronizare folosit în *Java* [9].

Comunicarea prin mesaje

În paragrafele anterioare am luat în discuție semafoarele (ca mecanisme *low-level* pentru implementarea concurenței) și monitoarele (ca exemple de obiecte procedurale, pasive care se implementează folosind partajarea memoriei).

În această secțiune introducem mecanisme de rezolvare a comunicării datelor și sincronizării proceselor care se implementează *fără partajarea memoriei*.

Abordarea cea mai generală a primitivelor de limbaj fără partajarea memoriei presupune atât sincronizare cât și transfer de date. Concret, avem două modalități de exploatare: flux de date și transfer de mesaje.

Mecanismul de lucru cu un ***flux de date*** presupune că un proces poate trimite octeți într-un flux nestructurat și un alt proces poate citi octeți de pe fluxul respectiv. În acest caz, pe flux nu sunt prezente informații despre structura octeților sau despre tipul de date transferate pe flux. O aplicație care folosește un astfel de mecanism trebuie să poată interpreta datele de pe flux conform cu o structură definită anterior, în timp ce sistemul însuși nu cunoaște această structură și nici nu oferă suport pentru aceasta.

Transferul de mesaje este un mecanism de lucru puternic, înrudit cu transferul informațiilor care însoțește transmiterea parametrilor la apelul de procedură. Mesajul conține un antet și un corp, fiecare având funcții bine delimitate în structura mesajului. Astfel, antetul (*header*) conține destinația informației conținute de mesaj, iar corpul mesajului (*body*) conține informația propriu-zisă și argumentele de transfer. Aceste argumente pot să fie supuse sau nu unor operații de validare a mesajului care se pot desfășura în mai multe etape.

Programele concurente care angajează transfer de mesaje sunt numite ***programe distribuite***. [7]

În funcție de tipul de sincronizare folosit pentru comunicarea datelor între procese prin mesaje avem:

- comunicare asincronă - procesul expeditor nu se interesează dacă mesajul transmis este recepționat de destinatar, cu atât mai puțin dacă este recepționat corect; acest model poate fi comparat cu comunicarea prin poștă;
- comunicare sincronă - după ce a transmis un mesaj, procesul expeditor se blochează în așteptarea confirmării de primire de la expeditor; acest model poate fi comparat cu comunicarea prin fax sau prin telefon.

Aici, prin *expeditor* și *destinatar* ne referim la procesele care comunică printr-un mesaj oarecare, adică la procesul care trimite mesajul, respectiv la procesul care recepționează mesajul (procesul căruia îi este adresat mesajul).

Se poate observa că, în cazul comunicării sincrone expeditorul și destinatarul trebuie să-și dedice un interval de timp exclusiv acestei comunicări, trebuie *să se întâlnească* pentru a comunica, respectiv să se sincronizeze pentru predarea (preluarea) mesajului. Spunem că are loc o întâlnire (un *rendez-vous*) între procesele care comunică. Acesta este mecanismul de sincronizare simplă a comunicării.

Dacă, în plus, comunicarea este prevăzută astfel încât expeditorul să primească împreună cu confirmarea de recepție și un mesaj de răspuns din partea destinatarului, atunci are loc un *rendez-vous extins* sau *invocare la distanță*.

În continuare ne propunem să luăm în discuție transmiterea asincronă și transmiterea sincronă de tip *rendez-vous* a mesajelor, urmând ca în paragraful următor să abordăm invocarea la distanță.

Transmiterea asincronă de mesaje, transmiterea sincronă de mesaje, precum și primitivele pentru invocare la distanță sunt mecanisme echivalente pentru comunicarea datelor și sincronizarea proceselor în sisteme concurente. [4]

Transmiterea asincronă de mesaje

Modalitatea asincronă de transmitere a mesajelor este varianta generală, care nu impune restricții de timp asupra comunicării între procesele care comunică prin transfer de mesaje. După cum am afirmat anterior, structura tipică a unui mesaj constă dintr-un antet și corpul mesajului. La rândul lui, antetul este format din informații referitoare la destinatarul mesajului, informații referitoare la expeditorul mesajului și informații referitoare la tipul mesajului.

Pentru ca transferul mesajelor să fie funcțional, în sistem trebuie să existe un serviciu direct răspunzător de lucrul cu mesaje. Acest serviciu de transport al mesajelor folosește antetul fiecărui mesaj pentru a identifica tipul mesajului și procesele care comunică. În plus, acest serviciu poate stabili și urmări respectarea unor protocoale de comunicare între procese.

Primitivele specifice de lucru pentru transmiterea de mesaje sunt grupate în perechile: `SEND` și `WAIT`, respectiv `SEND` și `RECEIVE`.

Dacă avem două procese `A` și `B`, care comunică la un moment dat prin transmitere de mesaje, atunci acțiunile celor două procese sunt următoarele:

- procesul `A` construiește și configurează mesajul, după care execută o operație `SEND(destinatar, mesaj);`
- procesul `B` așteaptă la punctul de întâlnire cu `A` executând operații `WAIT/RECEIVE(expeditor, spatiu_adresa)`, unde al doilea parametru reprezintă adresa de memorie la care trebuie să se depună mesajul.

Transmiterea este asincronă deoarece procesul `A` nu așteaptă de la procesul `B` vreun semnal prin care să i se confime recepționarea mesajului.

Acest scenariu presupune că procesele care comunică își cunosc reciproc identitatea (pot să se numească reciproc prin identificatorii transmiși ca parametri) și, în plus, există un protocol de transmitere de mesaje pe baza căruia expeditorul construiește mesajul de transmis și destinatarul interpretează mesajul primit.

În general, recepționarea mesajului coincide cu operația de copiere a mesajului din spațiul de adresă al expeditorului `A` în spațiul de adresă al destinatarului `B`, eventual prin intermediul unui *buffer* de colectare a mesajelor adresate lui `B` care este gestionat de serviciul de transmitere de mesaje. Astfel rezultă că, pentru transmiterea mesajului este necesar să se creeze două copii ale acestuia.

Rezolvarea și implementarea transmiterii asincrone de mesaje a condus la definirea unor noțiuni și concepte noi, cum ar fi: comunicarea prin porturi și canale, comunicare de tip *broadcast* (1 *la n*, sau *one-to-everyone*), *multicast* (1 la fiecare dintr-un grup de receptori înscriși) sau *unicast* (1 *la* 1) [2], [4], [7]. Implicit, aceste concepte se regăsesc în diverse primitive la nivelul fiecărui limbaj de programare care le recunoaște.

În interiorul unui modul de program, un mesaj poate fi transmis la o anumită locație (de exemplu, către o variabilă locală a programului respectiv). Dacă această locație este destinată comunicării programului cu exteriorul, atunci limbajul programului poate să o identifice ca pe un ***port de ieșire*** sau un ***canal de comunicare***. În timp ce aceste entități pot fi stabilite la începutul programului, locația care preia mesajul la intrarea în program, adică ***portul de intrare***, se poate stabili numai în timpul configurării mesajului. În plus, la configurare se stabilește care este procesul asociat unui anumit port de ieșire, respectiv care este procesul care va prelua datele de pe un anumit canal.

Transmiterea sincronă de mesaje

Reluând mecanismul de transmitere de mesaje între procesele `A` și `B` descris anterior, transmiterea este sincronă dacă procesul expeditor `A` este întârziat până când destinatarul execută operația `WAIT` prin care confirmă disponibilitatea de a recepționa mesajul transmis de `A`.

Față de operațiile interne executate asupra mesajului la transmiterea asincronă, mesajul este copiat numai o singură dată, atunci când expeditorul și destinatarul s-au sincronizat (s-au întâlnit) și pot comunica. Deoarece condiția de sincronizare este implicită, rezultă că sistemele cu transmitere sincronă de mesaje sunt mai ușor de gestionat decât cele asincrone.

Cele două variante de transmitere sincronă de mesaje sunt:

- transmiterea cu sincronizare simplă (*rendez-vous*);
- invocarea la distanță.

La rândul ei, invocarea la distanță poate fi reprezentată de un *rendez-vous extins* sau de *apelul de procedură la distanță* (***RPC***).

Prin *rendez-vous*, operațiile de transmitere și preluare sunt sincronizate, în sensul că transmiterea are loc efectiv numai dacă expeditorul este pregătit să transmită și destinatarul este pregătit să recepționeze. În caz contrar, procesul care ajunge primul la "întâlnire" este suspendat în așteptarea celuilalt. Cu alte cuvinte, nici un proces nu poate avansa înainte de terminarea transmiterii.

Invocarea la distanță

Ambele metode ale invocării la distanță, atât *rendez-vous-ul extins*, cât și *RPC*, combină aspecte întâlnite la monitoare și la transmiterea sincronă de mesaje. Ca și la monitoare, un modul sau un proces exportă operații care sunt invocate printr-un apel de tip ***call***. La fel ca în cazul unei secvențe `SEND - WAIT` care este specifică transmiterii sincrone de mesaje, execuția unui apel *call* întârzie procesul care îl execută. [7]

Noutatea pe care o aduc mecanismele *RPC* și *rendez-vous extins* constă în faptul că o operație de transmitere poate fi asociată unui canal de comunicație bidirecțional: atât de la apelant la procesul care oferă serviciul solicitat de apel, cât și invers. Apelantul întârzie până când primește un răspuns legat de încheierea execuției operației solicitate.

Diferența dintre *RPC* și *rendez-vous extins* constă în modul în care sunt deservite cererile invocate. Astfel, o primă abordare este să se declare o procedură pentru fiecare operație și să se creeze un proces nou pentru fiecare apel nou. Această abordare se numește ***RPC*** (***R**emote **P**rocedure **C**all*) deoarece apelantul și corpul procedurii apelate pot să se găsească pe mașini diferite (la distanță). O a doua abordare este să se asigure faptul că procesul apelant "se întâlnește" (stabilește un *rendez-vous*) cu un proces existent. Un *rendez-vous extins* constă din trei elemente:

- o operație de *intrare*, `ACCEPT` - prin care un proces așteaptă să fie invocat;
- o operație de *prelucrare a cererii*;
- o operație de *furnizare a rezultatului* (răspunsului).

În cazul comunicării prin apel de procedură, *mesajul* constă în valorile parametrilor efectivi transmiși entității apelate, iar *răspunsul* este conținut de valorile parametrilor returnați prin referință entității apelante.

Dezvoltarea programării concurente se poate face folosind procese și monitoare, care partajează același spațiu de adrese. Într-un astfel de model, monitoarele încapsulează variabile partajate, în timp ce procesele comunică și se sincronizează prin apelarea metodelor monitoarelor. Cu mecanismul *RPC* se poate folosi o singură componentă de programare, un modul, care să conțină atât procesele cât și procedurile (metodele). De asemenea, se permite modulelor să fie rezidente în spații de adresă diferite, eventual chiar în noduri diferite ale rețelei. Procesele interne ale unui modul pot partaja variabile și pot apela proceduri declarate în acel modul. Totuși, procesele dintr-un modul pot comunica cu procesele dintr-un alt modul numai prin apeluri de proceduri.

În esență, *RPC* furnizează numai un mecanism de comunicare între module. În interiorul unui modul trebuie programată explicit sincronizarea proceselor. În plus, programatorul trebuie să creeze procese pentru manipularea datelor comunicate prin *RPC*.

Mecanismul *rendez-vous* combină acțiunea de deservire a unei cereri cu alte prelucrări solicitate de apel. Cu mecanismul *rendez-vous*, un proces exportă operații (în sensul că operațiile sunt vizibile din afară), care pot fi apelate de alte procese.

Ca și în cazul mecanismului *RPC*, un proces invocă o operație prin intermediul unui apel *call*. Pentru *RPC*, apelul *call* numește un alt proces care să execute și o operație din acel proces care să se execute. Spre deosebire de *RPC*, în cazul mecanismului *rendez-vous*, o operație este executată de același proces care o exportă.

Pentru a pune în evidență deosebirile dintre mecanismele folosite pentru transmiterea de mesaje, se pot face trei observații referitoare la comunicarea prin canal și invocarea la distanță:

- comunicarea prin canal este identificată cu transmiterea de mesaje între procesele care comunică, în timp ce invocarea la distanță constă în apelarea din cadrul unuia dintre procese a unei secvențe de instrucțiuni a celuilalt proces, urmând ca aceste instrucțiuni să se execute la punctul de întâlnire;
- în cazul comunicării prin canal, sursa și destinația sunt univoc determinate, în timp ce pentru invocarea la distanță este precizată numai destinația. De aici rezultă că invocarea la distanță permite stabilirea unor legături de comunicare de tip *n* la 1;
- prin canal informația circulă de la sursă la destinație, în timp ce invocarea la distanță permite transferuri de mesaje în ambele sensuri.

Comparând comunicarea cu precizarea indirectă a partenerilor și invocarea la distanță, putem spune că invocarea la distanță permite interacțiunea proceselor implicate ca entități active în comunicare, în timp ce comunicarea prin canal sau prin resurse trebuie să apeleze la entități intermediare, pasive.

Totuși, comunicarea prin canale și invocarea la distanță sunt mecanisme echivalente de realizare a aplicațiilor concurente [4].

În concluzie, putem da o altă clasificare, mai generală, utilă pentru prezentarea mecanismelor de comunicare între procese (*IPC*) și anume:

- low-level: semafoare
- memorie partajată
  - ♦ secțiuni critice
  - ♦ expresii de tip cale
  - ♦ mecanisme procedurale
    - ▪ pentru obiecte pasive: monitoare
    - ▪ pentru obiecte active: rendez-vous procedural
- fără memorie partajată = comunicare prin mesaje
  - ♦ asincron = cu buffer intermediar
    - ▪ porturi și canale
    - ▪ broadcast și multicast
  - ♦ sincron
    - ▪ sincronizare simplă = rendez-vous
    - ▪ invocare la distanță
      - ❖ rendez-vous extins
      - ❖ RPC

Pentru entitățile din această clasificare, netratate în acest serial se pot consulta [2], [3], [4], [6].

Bibliografie

1. ***Coulouris G.F. ș.a.***, *Distributed systems - concepts and design*, Addison-Wesley, Wokingham, 1999
2. ***Bacon J.***, *Concurrent Systems - operating systems, database and distributed systems: an integrated approach*, Addison-Wesley, 1998
3. ***Vasilescu A.***, *Sisteme concurente, Referat în cadrul stagiului de pregătire pentru doctorat*, Universitatea Babeș-Bolyai, Cluj-Napoca, 2001
4. ***Georgescu H.***, *Programare concurentă - teorie și aplicații*, Editura Tehnică, București, 1996
5. ***Dijkstra E.W.***, *The structure of THE operating system*, Comm. ACM 11(5), 1968
6. ***Boian F.M.***, *Programarea distribuită în Internet - metode și aplicații*, Editura Albastră, Cluj-Napoca, 2000
7. ***Andrews G.R.***, *Concurrent Programming - Principles and Practice*, Benjamin/Cummings, Redwood City CA, 1991
8. ***Hoare C.A.R.***, *Monitors: an operating system structuring concept*, Comm. ACM 17(10), 1974
9 ***Scheiber E.***, *Lecții de calcul paralel*, Reprografia Universității "Transilvania" Brașov, 1999

D-na Anca Vasilescu este asistent universitar la Universitatea Transilvania din Brașov și poate fi contactată prin e-mail la adresa ***vasilex@info.unitbv.ro****. D-na Oana Georgescu este asistent universitar la Universitatea Sextil Pușcariu din Brașov și poate fi contactată prin e-mail la adresa* ***o.georgescu@info.unitbv.ro****.*