Obliczenia rozproszone w sieci, możliwości Javy i oferta Loglanu

Andrzej Salwicki

Przedstawiono pewne narzędzia programowania rozproszonego. Mają one dwie istotne cechy: zwięzły, przejrzysty oraz jednolity mechanizm programowania do obliczeń rozproszonych i obliczeń współbieżnych, a także mechanizm w pełni obiektowy, wprowadzający protokół obcego wywołania metody procesu przez inny proces. Mechanizm ten został zrealizowany w języku programowania Loglan'82. Porównano to narzędzie z narzędziami oferowanymi przez Javę. Język programowania Java zawiera rozbudowany mechanizm programowania współbieżnego, którego podstawę stanowi predefiniowana klasa Thread (czyli wątek). Ponadto Javie towarzyszy odrębny mechanizm programowania rozproszonego, tzw. RMI. Jednak RMI nie jest częścią Javy.

programowanie rozproszone, programowanie współbieżne, obiekty aktywne, wątki

Wprowadzenie

Obliczenia współbieżne oraz rozproszone są dziś prawie powszechne w oprogramowaniu. Jeszcze kilka lat temu były to tematy zarezerwowane dla specjalistów tworzących systemy operacyjne. Dziś w komputerach jest kilka procesorów, a firmy mają kilkaset komputerów połączonych siecią. Wreszcie dzięki tak spektakularnemu oprogramowaniu jak Seti(at)Home każdy może włączyć się w sieć milionów komputerów, poszukujących inteligentnego życia w kosmosie. Techniki programowania rozproszonego w efektowny sposób są wykorzystywane przez wyszukiwarki internetowe, np. Google.

Jednym z pierwszych języków programowania, w którym klasy i obiekty służyły współbieżności, był Concurrent Pascal, autorstwa Per Brinch Hansena (ok. 1975 r.), język wykorzystywany w CalTech. Wcześniej próby stworzenia narzędzi programowania współbieżnego pojawiły się w językach PL/I i Algol'68. Autorzy podręczników programowania współbieżnego sugerują, że programowanie obiektowe istotnie wspomogło programowanie współbieżne (np. Concurrent Pascal). Powstaje jednak zasadnicze pytanie: *czy w świecie programowania obiektowego istnieją dobre mechanizmy do programowania współbieżnego i rozproszonego?* W języku Concurrent Pascal wprowadzono dwa rodzaje klas dedykowanych programowaniu współbieżnemu: *procesy* i *monitory*. Procesy mają wątki instrukcji do wykonania, a monitory służą tworzeniu obiektów pasywnych, władających zasobami wspólnie wykorzystywanymi przez obiekty procesów. Dostęp do zasobów polega na wywoływaniu metod zadeklarowanych w monitorach.

Twórcy języka Java, który powstał w 1995 r., podążali tym tropem. Jednak wykonanie odbiegło daleko od zamierzeń. Po pierwsze, stworzono dwa odrębne mechanizmy: jeden do programowania współbieżnego – to interfejs Runnable, klasa Thread i słowo kluczowe *synchronized*, drugi – mechanizm RMI (*Remote Method Invocation*) do programowania rozproszonego w sieci. Trzeba się ich uczyć osobno. Po drugie, zasugerowano, że programista powinien opanować wiele pojęć i instrukcji, zanim będzie tworzyć programy współbieżne w Javie.

W języku programowania Loglan'82 [1], [10], [12] zwraca uwagę oryginalna metoda programowania rozproszonego i współbieżnego. Wyjątkową pozycję zaś zapewniają mu następujące właściwości:

- System pojęć, operacji i własności tych operacji jest znacznie prostszy niż w innych językach programowania współbieżnego. Dla porównania, w Javie lista pojęć i operacji właściwych dla programowania współbieżnego zawiera ponad 40 pozycji, a w Loglanie jest ich mniej niż 10.
- Loglan oferuje jednolity model obliczeń współbieżnych i rozproszonych, inne podejścia przeciwstawiają sobie te dwa sposoby programowania, np. w Javie programowanie rozproszone wymaga opanowania odrębnego mechanizmu, tzw. RMI.
- Wykonywanie programów loglanowskich sekwencyjnych i współbieżnych odbywa się na wirtualnej maszynie loglanowskiej, natomiast programów rozproszonych – na wieloprocesorowym wirtualnym komputerze loglanowskim. Komputer taki powstaje, gdy wirtualne maszyny loglanowskie skomunikują się ze sobą przez sieć.
- Jako jedyny chyba język programowania, Loglan używa protokołu *obcego wołania* metod (*alien call*) i tzw. maski. Maska obiektu aktywnego oraz operacje na niej umożliwiają dynamiczne zmienianie statusu metody z prywatnej na publiczną i odwrotnie. Obce wołanie metod jest w pełni obiektowym protokołem synchronizacji oraz komunikacji wątków zawartych w obiektach aktywnych i jedynym takim protokołem.

Godną uwagi koncepcję obcego wywoływania metod przedstawił i zrealizował B. Ciesielski [1] w 1988 r. Pojęcie maski oraz instrukcje enable i disable rozważali U. Petermann, A. Szałas i D. Szczepańska [8], [9] w 1985 r. O. Świda [12] przeniósł implementację z platformy MS-DOS na system Linux, dodał terminal loglanowskiej maszyny virtualnej VLP (*Virtual Loglan Processor*), łączenie maszyn wirtualnych w wieloprocesorowy wirtualny komputer loglanowski i kilka innych ważnych dla programisty narzędzi.

Moduł process

W Loglanie oprócz modułów klas występują moduły procesów.

Porównanie klas i procesów

Procesy, jakie można zadeklarować w programie loglanowskim, są klasami wyposażonymi w dodatkowe cechy.

- Na podstawie deklaracji klasy A

  unit A: **class** ... **end** A;

  można tworzyć obiekty tej klasy, np.

  aa:= **new** A(), ab:= **new** A().

  Obiekty klasy są pasywne, *nie* wykonują własnych instrukcji. Metody zadeklarowane w klasie mogą być wykonywane zdalnie przez instrukcje zawarte w innych modułach programu.

- Na podstawie deklaracji procesu P

  unit P: **process** ... **end** P;

  można tworzyć obiekty aktywne, np.

  pp:= **new** P(0,...), p2:= **new** P(7,...).

  Obiekt aktywny *może* wykonywać instrukcje własnego wątku. Ponadto, obiekt aktywny może wykonywać swoje metody na zlecenie innych obiektów aktywnych.

Składnia procesu

Wyróżniony rodzaj klas – **process** umożliwia zadeklarowanie wzorców, wg których będą tworzone obiekty aktywne.

```
unit A : B process(param)   – –  nagłówek, proces A dziedziczy z B!
  ⟨deklaracje⟩              – –  te wielkości są niedostępne z zewnątrz
begin                       – –  (oddziela deklaracje od instrukcji
  ⟨konstruktor⟩             – –  te instrukcje inicjalizują obiekt aktywny
  return;                   – –  oddziela instrukcje inicjalizacji od wątku
  ⟨wątek⟩                   – –  tu umieszcza się instrukcje wątku
end A
```

Ograniczenia i rozszerzenia

Poniżej podano ograniczenia i rozszerzenia modułu process.

- Pierwszy parametr musi być typu *integer*. Jego wartość określa numer procesora, na którym będzie alokowany obiekt aktywny.
- Proces może dziedziczyć z modułu B – klasy lub procesu. Może go rozszerzać.
- Deklaracje lokalne i parametry lokalne obiektów *a* lub *b* nie są widoczne z zewnątrz przez zdalny dostęp, np. wyrażenie *a.x* jest błędne. Inne obiekty aktywne mogą wywoływać metody danego obiektu aktywnego, ale semantyka tych poleceń to tzw. obce wołanie metody. Nie mylić z wywołaniem odległym.
- Można zdalnie wywoływać metody, ale semantyka tych poleceń to tzw. obce wołanie metody.

Obiekty aktywne

Tworzenie, alokacja i uruchamianie

Obiekty aktywne tworzy się, wykonując instrukcję:

aa := new A(*nr*, <inne parametry>).

Tworzony jest wtedy obiekt o nazwie *aa*, alokowany na maszynie o numerze *nr*. Dla przykładu, polecenie:

– a:= new A(0, xxx) spowoduje, że obiekt *a* będzie na tym samym procesorze co wątek wykonujący to polecenie (*współbieżność*);
– b:= new A(3, yyy) spowoduje, że obiekt *b* będzie alokowany na procesorze nr 3 (*rozproszenie*); obiekt ten jest na razie pasywny;
– resume(aa) umożliwi uruchomienie wykonywania wątku instrukcji w obiekcie *aa*.

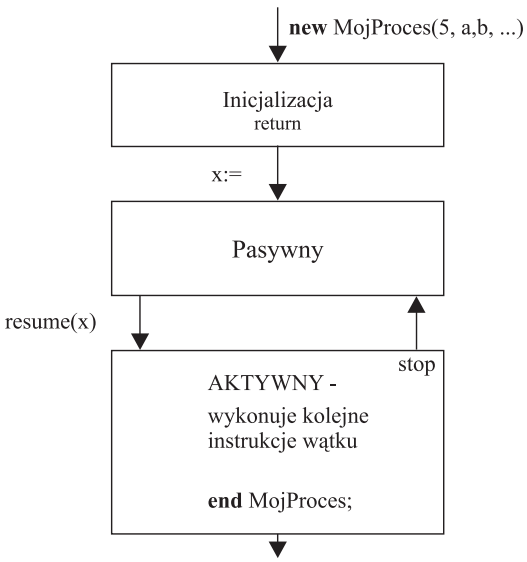
Stany obiektu

Po utworzeniu, nowy obiekt aktywny jest w stanie pasywnym, nie wykonuje żadnych instrukcji (rys. 1). Pewien obiekt aktywny może go uaktywnić, wykonując polecenie:

resume(c).

Warunkiem jest posiadanie odnośnika do obiektu, zwykle ma go obiekt aktywny, w którym wykonano polecenie:

c := new MojProces(parametry-Aktualne).



Rys. 1. *Scenariusz obiektu aktywnego*

Od tego momentu instrukcje obiektu aktywnego *c* są wykonywane współbieżnie z instrukcjami innych wątków. Proces *c* może zawiesić swój wątek – instrukcja stop. Po wykonaniu wszystkich instrukcji, czyli po osiągnięciu "end", obiekt aktywny jest zakończony i usuwany, ponieważ nic z nim nie można dalej zrobić. Jego pola i metody są niedostępne.

Ograniczenia. Parametrami aktualnymi nie mogą być obiekty klas, procedury lub funkcje. Mogą nimi być wielkości typów pierwotnych (integer, real, boolean, string, char) oraz obiekty aktywne procesów, a także metody procesów (procedury lub funkcje).

Maska. Każdy obiekt aktywny ma predefiniowany atrybut MASK – maska. Niech *p* będzie obiektem aktywnym typu A. Wartością atrybutu MASK w obiekcie *p* jest pewien podzbiór zbioru metod obiektu, zadeklarowanych w procesie A. Początkową wartością maski jest zbiór pusty $\emptyset$.

Stan maski obiektu aktywnego zmieniają instrukcje:

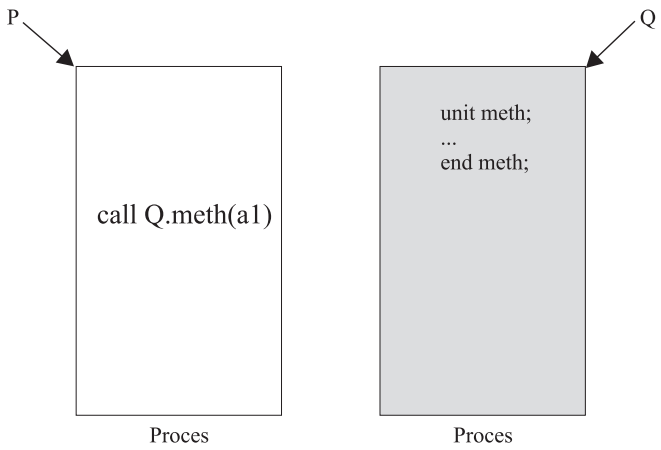
enable(m1, m2); $--\ MASK := MASK \cup \{m1, m2\}$,

disable(m2,m3); $--\ MASK := MASK \setminus \{m2, m3\}$.

Obiekt aktywny przyjmie zlecenie wykonania swej metody *m* od innego obiektu aktywnego, jeśli jest w stanie AKTYWNY i metoda *m* znajduje się w masce. W przeciwnym przypadku obiekt wzywający będzie oczekiwać.

Obce wołanie metody

Proces P wzywa proces Q do wykonania metody meth (rys. 2).

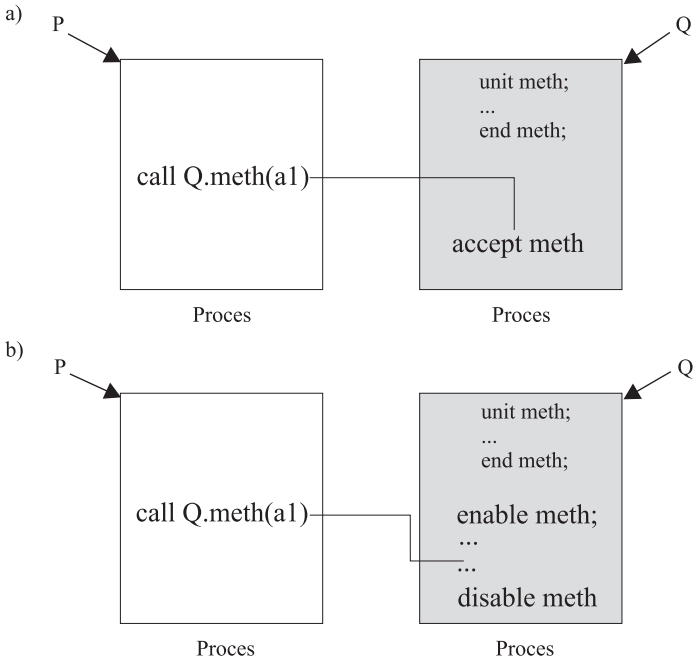


Rys. 2. *Obce wołanie metody*

- Proces wzywany Q jest aktywny, wykonuje swoje instrukcje: w przeciwnym przypadku nie będzie mógł zareagować, będzie w stanie pasywnym. Proces P będzie czekać na uaktywnienie procesu Q.
- Proces Q dowie się o nadejściu zlecenia od innego procesu po zakończeniu aktualnie wykonywanej instrukcji atomowej.
- Proces Q zastanowi się, czy ma chęć zaakceptowania zlecenia, zbada swoją MASKę (rys. 3).
- Jeśli odpowiedzi na powyższe pytania są pozytywne, to proces P przekazuje aktualne parametry procesowi Q, proces Q uruchamia metodę, zapamiętuje stan MASKi i zeruje ją $Mask := \emptyset$ (rys. 3a).
- Po zakończeniu wykonywania metody przekazuje te parametry, które były wskazane jako wyjście, procesowi P, odtwarza MASKę i wykonuje instrukcję return.
- Obliczenia procesów P i Q rozchodzą się.

Współpraca dwóch procesów może być *zsynchronizowana*. Występuje to wtedy, gdy proces właściciel metody *meth* wykona instrukcję **accept**. Wtedy procesy P i Q muszą się najpierw *spotkać*, a dopiero potem wspólnie wykonają parę instrukcji **call Q.meth**(...) w procesie P oraz **accept** ... w procesie Q. Obce wywołanie metody może zostać wykonane w trybie *niezsynchronizowanym*. Wtedy zrealizowanie tej metody może przerwać normalny tok wykonywania instrukcji wątku w obiekcie aktywnym Q, gdziekolwiek między instrukcją **enable meth** a instrukcją **disable meth**. Należy pamiętać, że maska może zawierać więcej niż jedną nazwę metody. Stwarza to dalsze ciekawe możliwości. Instrukcja accept może także dodawać kilka nazw metod do maski – tylko na czas wykonania tej instrukcji.

Warto też wspomnieć, że instrukcja return, kończąca wykonywanie metody, może mieć postać: return enable m1, m2 disable m3, m4.



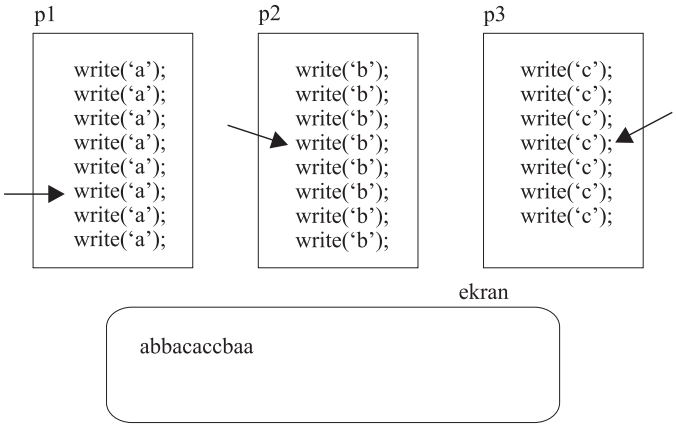
Rys. 3. *Wykonywanie zleceń: a) synchroniczne z akceptacją zlecenia; b) asynchroniczne bez akceptacji, przerywające pracę procesu Q*

Przykład 1

Utworzone są trzy obiekty aktywne typu *pisarz*. Każdy z nich drukuje 55 razy powierzoną mu literę. Drukowanie jest *niezsynchronizowane*.

```
program first;
    unit pisarz: process(node: integer, c: character);
        var i: integer;
    begin
        return;
        for i := 1 to 55 do write(c) od; writeln;
    end pisarz;
    var p1, p2, p3: pisarz;
begin
    p1 := new pisarz (0, 'a'); (* tworzymy trzech pisarzy *)
    p2 := new pisarz (0, 'b');
    p3 := new pisarz (0, 'c');
    resume(p1); (* i puszczamy ich w ruch)
    resume(p2);
    resume(p3); (* teraz są 4 wątki aktywne *)
    ...
end first
```

Wątki konkurują o wspólny zasób – ekran. W efekcie na ekranie jest widoczna mieszanina liter ‘a’, ‘b’ i ‘c’ (rys. 4).



Rys. 4. *Nieskoordynowane drukowanie, uwidaczniające konflikt w dostępie procesów do wspólnego zasobu ekran (na ekranie jeden z możliwych przeplotów liter a, b i c)*

Przykład 2

W tym przykładzie wykorzystano obce wołania metod i maskę oraz pokazano, jak zaimplementować pojęcie semafora (można to zresztą zrobić wieloma innymi sposobami).

```
program second;
    unit pisarz: process(node: integer, c: char, s: semafor);
        var i: integer;
    begin
        return;
        call s.P;
        for i := 1 to 55 do write(c); od; writeln;
        call s.V;
    end pisarz;
    unit semafor: process(node: integer);
        unit P: procedure; end P;
        unit V: procedure; end V;
    begin
        return;
        do accept P; accept V od;
    end semafor;
    var s: semafor, p1, p2, p3: pisarz;
begin
    s := new semafor(0); resume(s);
    p1 := new pisarz(0,‘a’,s); resume(p1);
    p2 := new pisarz(0,‘b’,s); resume(p2);
    p3 := new pisarz(0,‘c’,s); resume(p3);
end second
```

W tym przykładzie pojawia się obiekt aktywny *s* typu semafor. Jego wątkiem są powtarzające się dwie instrukcje *accept P* i potem *accept V*. Natomiast P i V są metodami zadeklarowanymi w procesie semafor, a więc obiekt *s*, po jego uruchomieniu instrukcją *resume(s)*, będzie najpierw oczekiwać na wezwanie do wykonania metody P na rzecz jakiegoś procesu *pisarz* (ponieważ wykonanie instrukcji *accept* polega na oczekiwaniu aż nadejdzie wezwanie do wykonania pewnej metody z maski; maska w tym przypadku zawiera tylko jedną nazwę: nazwę metody P). Potem obiekt *s* będzie oczekiwać na wezwanie do wykonania metody V. Z tekstu modułu *pisarz* widać, że będzie to ten sam *pisarz*, który poprzednio wezwał obiekt aktywny *s* do wykonania metody P. Widać też, że żaden inny *pisarz* nie może wykonać metody P w semaforze *s*, dopóki poprzedni *pisarz* nie zakończy pisania i nie zwolni semafora, wykonując instrukcję s.V. Na ekranie pojawią się trzy wiersze złożone z liter a, b lub c odpowiednio, bez przeplotów. Zostało zapewnione wzajemne wykluczanie się *pisarzy* w dostępie do ekranu.

Ten przykład nie jest najprostszym rozwiązaniem, ale umożliwił objaśnienie działania obcego wołania metody w innym procesie w wariancie synchronicznym.

Prostsze rozwiązanie zawierałoby proces ekran z metodą drukuj, wypisującą wiersz złożony z 55 znaków. Proces semafor staje się zbędny.

Wieloprocesorowy wirtualny komputer loglanowski

Programy sekwencyjne napisane w Loglanie są wykonywane na loglanowskiej maszynie wirtualnej. Jest to oprogramowanie tworzące środowisko do wykonywania kodu, utworzonego podczas kompilacji programu napisanego w Loglanie [11].

W środowisku systemu Linux maszyna wirtualna objawia się jako okienko. Jedna maszyna wirtualna może obsługiwać wiele obiektów aktywnych – procesów. Są to tzw. obliczenia współbieżne.

Do połączenia maszyn wirtualnych dochodzi, gdy:

– działają co najmniej dwie maszyny wirtualne (na różnych komputerach);
– w okienku jednej z nich zostanie wybrany z listy poleceń *Connect* i podany nr IP drugiego komputera;
– tę operację można powtarzać lub zaprogramować łączenie większej liczby maszyn wirtualnych w pliku konfiguracyjnym loglanowskiej maszyny wirtualnej (przy założeniu stałej konfiguracji).

W terminalu maszyny wirtualnej można:

– wykonywać program;
– zakończyć (*kill*) program, który nas nie słucha;
– łączyć (*connect*) i rozłączać (*disconnect*) maszyny wirtualne;
– wezwać edytor programów i kompilować programy.

Java

Współbieżność w Javie jest wyposażona w wiele mechanizmów, instrukcji, typów itd.

Podstawę współbieżności w Javie stanowią:

1) **tworzenie wątków**: polega ono na deklaracji klasy implementującej interfejs Runnable; wymaga to stworzenia w tej klasie metody *run*;

2) **tworzenie zadań**: w najprostszej postaci zadanie tworzy się, przekazując obiekt z metodą *run*, nowo tworzonemu obiektowi klasy Thread;
3) **uruchomienie zadań**: w tym celu w każdym obiekcie klasy Thread należy wykonać metodę *start*;
4) **synchronizacja**: synchronizacja wątków wykorzystuje monitory; monitorem jest obiekt klasy, w której pewna metoda została opatrzona kwalifikatorem *synchronized*.

Reszta, choć bardzo obszerna, nie wnosi do programowania współbieżnego niczego istotnie nowego. Często jest wprowadzana pod pretekstem zwiększania efektywności.

Przykłady programów współbieżnych w Javie

Przykład 3

Ta klasa implementuje wątek o zadaniu podobnym do zadania opisanego w procesie *pisarz* (przykład 1 i 2).

```
import java.util.concurrent.*;
public class Pisarz extends Thread {
protected char znak;
protected int licznik =25;
public Pisarz(char c) {
    this.znak = c;
}
public void run () {
    try{
        while (licznik– > 0) {
            System.out.print(znak);
            Thread.yield();
            TimeUnit.MILLISECONDS.sleep((int)(100*Math.random())); }
        };
    }
        catch(InterruptedException e) {System.err.println(“Przerwany”);}
        System.out.println(“stop”+znak);
    }
}
```

Przykład 4

W tym przykładzie pokazano, jak w Javie tworzy się wątki i je uruchamia.

```
public class MorePisarze {
    public static void main(String[] args) {
    for (int i=0; i<15; i++)
        new Thread(new Pisarz((char)(65+i))).start();
    System.out.println(”Oczekiwanie na start”);
    }
}
```

Obliczenia rozproszone w Javie

Java nie opisuje, jak przeprowadzać obliczenia rozproszone. Nie ma na ten temat słowa ani w specyfikacji języka [3], ani w podręcznikach Javy [2]. Natomiast w dokumentacji rozwiązań stowarzyszonych z Javą można znaleźć opis mechanizmu RMI, umożliwiającego współpracę obiektów podklas interfejsu Runnable, alokowanych na różnych komputerach połączonych siecią [4].

Nie jest to ten sam mechanizm co współbieżność!

Obiekty tworzy się na różnych komputerach połączonych siecią i zgłasza je do rejestracji w odpowiednio wybranym komputerze, RMI Registry.

Program na komputerze A może zapytać w rejestracji o obiekt X i otrzymać (sieciowy) odnośnik *r* do tego obiektu. (*Trzeba jednak wiedzieć, o co zapytać*). Wykorzystując ten odnośnik, można wywoływać metody zdalnego obiektu, np. r.meth(). Należy podkreślić, że metodę meth wykonuje *proces wzywający*. Obiekt zdalny pełni rolę serwera-monitora.

Z tego opisu, zresztą bardzo uproszczonego, widać, jak odmiennie wygląda programowanie rozproszone w Javie i w Loglanie.

W jednej z niewielu książek poruszających temat współbieżności w Javie [2] poświęcono temu zagadnieniu obszerny (150 str.) rozdział (jednak mechanizmu RMI tam nie omówiono).

Pod koniec tego rozdziału autor zauważa: „*masz zapewne wrażenie, że wielowątkowość w Javie to narzędzie skomplikowane i trudne w użyciu*" i dalej „*istnieje inny model – obiektów aktywnych*". Obiekty aktywne w jego rozumieniu są uboższe od procesów – obiektów aktywnych w Loglanie, m.in. o obce wołanie metod, maski itp.

Porównanie języków

Programując w Javie, trzeba nauczyć się dwóch mechanizmów: pierwszego, aby programować wątki wykonywane współbieżnie na jednym komputerze, drugiego, zwanego RMI, aby programować wątki wykonywane na różnych komputerach połączonych siecią.

Do oceny przydatności obu języków do obliczeń będą użyteczne, wyszczególnione dalej, elementy programowania współbieżnego i rozproszonego.

Programowanie współbieżne w Javie:

– deklaruj klasy pochodne klasy Thread;
– twórz obiekty tych klas, czyli wątki;
– uruchamiaj wątki – start();
– wątki mogą mieć pamięć dzieloną;
– twórz obiekty – monitory, stosując słowo *synchronized*;
– i wiele innych ...

Współbieżność w Loglanie:

– zadeklaruj procesy;
– twórz obiekty aktywne procesów (na tym samym procesorze);
– pole pamięci procesu jest prywatne;

– proces może udostępniać (*enable*) swe metody lub je chować (*disable*) przed innymi procesami;
– proces A może wzywać proces B do wykonania jego metody p:

call B.p(...),

na rzecz procesu B; jest to obce wywołanie tej metody.

Lista oferowanych (czy zawsze potrzebnych?) instrukcji, pojęć itp. programowania współbieżnego jest w Javie 4–5 razy dłuższa niż w Loglanie.

Programowanie współbieżne w obu językach ma podobne walory z małą przewagą Loglanu, w którym model obliczeń jest bardziej zwięzły i o większej sile programowania.

Rozproszony program w Javie wymaga utworzenia:

– definicji interfejsów dla odległych usług;
– implementacji odległych usług;
– plików z namiastkami klienta (*Stub*) i namiastkami serwera (*Skeleton*);
– serwera, na którym będą dostępne odległe usługi;
– usługi RMI Naming, która pozwala klientowi odnaleźć odległe usługi;
– serwera FTP (*File Transfer Protocol*) lub HTTP (*Hyper Text Transfer Protocol*);
– programu klienta, który potrzebuje odległych usług.

Rozproszenie obliczeń w Loglanie wymaga:

– połączenia kilku maszyn wirtualnych w sieć za pomocą poleceń *Connect*;
– utworzenia obiektów aktywnych i ich alokacji na wybranych węzłach;
– uruchomienia tych obiektów, podobnie jak w programowaniu współbieżnym.

Porównując oba języki programowania, należy pamiętać też o niżej podanych uwagach, dotyczących Loglanu.

W Loglanie można napisać program P i zmieniając go w minimalnym stopniu, otrzymać jego wersje: współbieżną i rozproszoną. Wystarczy w tym celu:

– zadeklarować funkcję o nazwie, np. *nrKomputera*, typu integer;
– zadbać, aby we wszystkich wyrażeniach postaci **new** Proces(*a*, inneParametry), generujących obiekty aktywne, pierwszy parametr *a* miał postać: *nrKomputera*().

Wersję współbieżną programu P można otrzymać, gdy funkcja *nrKomputera* jest zadeklarowana tak:

```
unit nrKomputera: function(): integer;
begin
    result:= 0
end nrKomputera;
```

W efekcie wszystkie procesy tego programu będą wykonywane na tym samym procesorze.

Wersję rozproszoną otrzymuje się, gdy funkcja *nrKomputera* zwraca wielkości różne od zera. Oczywiście można też uzyskać wersję, w której współbieżność i rozproszenie są skombinowane w jakiś inny wybrany sposób, funkcja zwraca czasem zero, a czasem wartość różną od zera.

Taka transformacja programu nie byłaby możliwa w Javie.

Zakończenie

Status loglanowskiego systemu VLP wciąż jest *eksperymentalny*. Można go pobrać i zainstalować, ale do zastosowań komercyjnych brakuje mu "fontann i wodotrysków". Trzeba by lepiej wykorzystywać istniejące biblioteki, np. graficzne. I mimo że nadal nie ma pełnej implementacji rozpraszania/współbieżności na platformę Windows, to Loglan świetnie nadaje się do dydaktyki programowania współbieżnego i rozproszonego.

Ciekawe, czy programiści wykorzystują wątki w Javie do programowania współbieżnego i/lub rozproszonego. Popularność Javy powinna ich do tego zachęcać, jednak skomplikowane mechanizmy współbieżności i rozpraszania obliczeń niewątpliwie zniechęcają.

Trzeba wspomnieć o innych aspektach projektu badawczego Loglan:

– pracom programistycznym towarzyszyły prace teoretyczne, część wyników można znaleźć w monografii [7] w rozdziale: „Problemy i teorie zainspirowane przez Loglan" (część z nich była tematem rozpraw doktorskich);
– niektóre wyniki uzyskane podczas prac nad Loglanem znajdują teraz swe zastosowanie podczas analizy problemów dziedziczenia w Javie, np. [6].

Warto zastanowić się, jakie są **perspektywy Loglanu**. Pesymistyczna jest taka, że kiedyś, te dzisiaj zapoznane narzędzia, zostaną odkryte na nowo! Prawdopodobnie w nowym języku programowania i nie w Polsce.

Z analizy przeprowadzonej w tym artykule wynikają następujące **zadania** i **problemy badawcze**, których realizacja stworzy pozytywne perspektywy Loglanu.

- Zbudowanie pakietów implementujących loglanowski model obliczeń rozproszonych (*alien call*, łączenie maszyn wirtualnych itp.) dla trzech popularnych języków programowania obiektowego:
  – Java,
  – C++,
  – Ada95.
- Przeniesienie VLP na platformę Windows (obecna wersja Loglanu na Windows nie zawiera VLP, dziś powinno to być łatwiejsze, bo działa już na platformie Windows środowisko graficzne KDE (*K Desktop Environment*) z systemu Linux).
- Zmodyfikowanie maszyny wirtualnej Javy i C++ tak, aby zawrzeć w nich koncepcje przyśpieszające *alien call* oraz łączenie takich maszyn.
- Stworzenie nowej loglanowskiej wtyczki do środowiska Eclipse.
- Opracowanie nowszej wersji języka Loglan i zbudowanie kompilatorów dla niej (por. [5]).
- Zrealizowanie rozproszonych aplikacji wykorzystujących VLP.

Warto włączyć się w te prace.

Podziękowania

Pragnę podziękować Panu docentowi Andrzejowi Hildebrandtowi za zachętę do napisania tego artykułu. Dziękuję też Pani profesor Grażynie Mirkowskiej i Panu profesorowi Andrzejowi Szałasowi za wiele wnikliwych uwag, które pozwoliły ulepszyć tę pracę.

Bibliografia

[1] Ciesielski B.: *Implementacja procesów rozproszonych w LOGLANie*. Praca magisterska. Warszawa, Instytut Informatyki Uniwersytetu Warszawskiego, 1988

[2] Eckel B.: *Thinking in Java*. Gliwice, Helion, 2006

[3] Gosling J., Joy B., Steel G., Bracha G.: *Java Language Specification*. 3 wyd., Sun Microsystems, 2005, http://java.sun.com/docs/books/jls/

[4] *Java Remote Method Invocation*, http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html

[5] Kreczmar A., Salwicki A., Warpechowski M.: *Loglan'88. Report on the Programming Language*. *LNCS*, vol. 414. Heidelberg, Springer-Verlag, 1990

[6] Langmaack H., Salwicki A., Warpechowski M.: *A deterministic algorithm for identifying direct superclasses in Java*. Fundamenta Informaticae, vol. 85, pp. 343–357, 2008

[7] Mirkowska G., Salwicki A.: *Algorithmic Logic*. Warszawa–Dordrecht, PWN & J. Reidel Publ., 1987

[8] Petermann U., Szałas A.: *A note on PCI distributed processes communicating by interrupts*. SIGPLAN Notices, vol. 20, no. 3, pp. 37–46, 1985

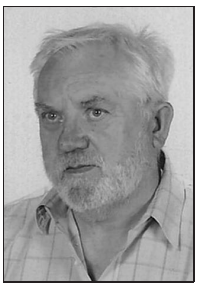
[9] Szałas A., Szczepańska D.: *Exception handling in parallel computations*. SIGPLAN Notices, vol. 20, no. 10, pp. 95–104, 1985

[10] Szałas A., Warpechowska J.: *Loglan*. Warszawa, WNT, 1991

[11] Świda O.: *Oprogramowanie VLP – Virtual Loglan Processor*, http://duch.mimuw.edu.pl/~salwicki/vlp26

[12] Świda O.: *Rozproszone środowisko programowania obiektowego*. Rozprawa doktorska. Warszawa, Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, 1996

Andrzej Salwicki



Prof. dr hab. Andrzej Salwicki (1938) – absolwent Wydziału Matematyki i Fizyki Uniwersytetu Warszawskiego (1960), doktorat (1969), habilitacja (1974), pracownik naukowy Instytutu Maszyn Matematycznych PAN (1959–1964); pracownik naukowo-dydaktyczny Instytutu Informatyki Uniwersytetu Warszawskiego (1964–1992, 2000–2006), profesor Universite de Pau (1990–1998), profesor w Instytucie Łączności w Warszawie (od 2007); inicjator programu badawczego logika algorytmiczna (w 1968 r. – osiem lat wcześniej przed logiką dynamiczną w MIT, USA); inicjator programu badawczego Loglan – język programowania obiektowego (w 1977 r. – kilkanaście lat wcześniej przed Javą); autor kilku książek i wielu artykułów naukowych; promotor 17 doktorów (w większości dziś profesorów w Polsce, USA, Kanadzie, Meksyku i Niemczech); zainteresowania naukowe: logika algorytmiczna i jej zastosowania w inżynierii oprogramowania, programowanie obiektowe i rozproszone.
e-mail: A.Salwicki@itl.waw.pl