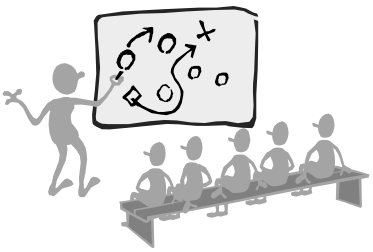


Podstawy przetwarzania rozproszonego

Literatura

1. **J. Bacon**, *Concurrent Systems - An Integrated Approach to Operating Systems*, Addison Wesley, 1992.
2. **J. Brzeziński,** *Ocena stanu globalnego w systemach rozproszonych,* OWN 2001
3. **K. M. Chandy, J. Misra**, *Parallel Program Design, A Foundation,* Addison Wesley, 1988.
4. **A. Gościński**, *Distributed Operating Systems, The Logical Design,* Addison Wesley, 1991.
5. **J.M. Helary, M. Raynal**, *Synchronization and Control of Distributed Systems and Programs,* John Wiley & Sons, 1990.
6. **K. Hwang, F. Briggs**, *Computer Architecture and Parallel Processing,* McGraw Hill, 1984.
7. **J.Jaja**, *An Introduction to Parallel Algorithms,* Addison Wesley, 1992.
8. **N. Lynch**, *Distributed Algorithms,* Morgan Kaufmann Publishers, Inc., 1996.
9. **S.J. Müllender**, *Distributed Systems,* ACM Press, 1993.
10. **M. Raynal**, *Ditributed Algorithms and Protocols,* John Wiley & Sons, 1988.
11. **M. Singhal, N.G. Shivaratri**, *Advanced Concepts in Operating Systems – Distributed, Database, and Multiprocessor Operating Systems,* McGraw Hill, 1994.
12. **A. Tanenbaum**, *Modern Operating Systems*, Prentice-Hall, 1992
13. **G.Tel**, *Introduction to Distributed Algorithms,* Cambridge University Press, 1994.

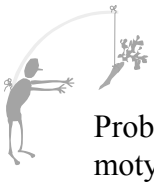
Charakterystyka systemów rozproszonych

Systemy rozproszone charakteryzują się:

- **dużą wydajnością**
  w sensie dużej mocy obliczeniowej i maksymalnej przepustowości, krótkiego czasu odpowiedzi;
- **dużą efektywnością inwestowania**
  w sensie kosztów niezbędnych do uzyskania wymaganej wydajności systemu;
- **wysoką sprawnością wykorzystania zasobów**
  w sensie stopnia wykorzystania zasobów, współczynnika jednoczesności;
- **skalowalnością**
  w znaczeniu możliwości ciągłego i nieograniczonego rozwoju systemu bez negatywnego wpływu na jego wydajność i sprawność;
- **wysoką niezawodnością**
  w sensie odporności na błędy;
- **otwartością funkcjonalną**
  w sensie łatwości realizacji nowych, atrakcyjnych usług komunikacyjnych, informatycznych i informacyjnych.

Problemy związane z konstrukcją systemów rozproszonych

- ❑ optymalne zrównoleglenie algorytmów przetwarzania
- ❑ ocena poprawności i efektywności algorytmów rozproszonych
- ❑ alokacja zasobów rozproszonych
- ❑ synchronizacja procesów
- ❑ ocena globalnego stanu przetwarzania
- ❑ realizacja zaawansowanych modeli przetwarzania
- ❑ niezawodność
- ❑ bezpieczeństwo



Motywy

Problematyka przetwarzania rozproszonego jest motywowana:

- ogromnym rzeczywistym zapotrzebowaniem na systemy rozproszone;
- dostępnością środków technicznych i praktyczną możliwością realizacji systemów rozproszonych;
- różnorodnością otwartych problemów związanych z konstrukcją i zarządzaniem systemami rozproszonymi

Systemy rozproszone – podstawowe pojęcia

Rozproszony system informatyczny obejmuje:

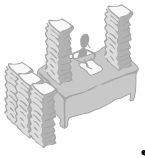
- ❖ środowisko przetwarzania rozproszonego
  węzły, łącza
- ❖ zbiór procesów rozproszonych
  zbiór procesów sekwencyjnych realizujących wspólne cele przetwarzania

Środowisko przetwarzania rozproszonego

Środowisko przetwarzania rozproszonego jest zbiorem $\mathcal{N}$ autonomicznych jednostek przetwarzających N_i (węzłów), zintegrowanych *siecią komunikacyjną* (*środowiskiem komunikacyjnym, łączami komunikacyjnymi, łączami transmisyjnymi*).

W środowisku tym komunikacja między węzłami możliwa jest tylko przez transmisję ***pakietów informacji*** (*wiadomości*, *komunikatów*) łączami komunikacyjnymi.

Jednostki przetwarzające realizują przetwarzanie z prędkością narzucaną przez *lokalne zegary*. Jeżeli zegary te są niezależne, to mówimy, że węzły działają ***asynchronicznie***. Jeżeli natomiast zegary te są zsynchronizowane lub istnieje wspólny zegar globalny dla wszystkich węzłów, to mówimy, że węzły działają ***synchronicznie***.



Węzły

Jednostka przetwarzająca $N_i \in \mathcal{N}$ (*węzeł*) jest elementem środowiska przetwarzania rozproszonego obejmującym:

- ❖ procesor
- ❖ lokalną pamięć operacyjną
- ❖ interfejs komunikacyjny

Procesor wykonuje automatycznie program zapisany w *pamięci lokalnej*, gdzie pamiętane są również dane. *Interfejs komunikacyjny* umożliwia węzłom dostęp do łączy i tym samym wzajemną wymianę informacji - *komunikatów* (ang. message passing, message exchange) oraz komunikację z użytkownikiem.

Łącza (1)

Łącze komunikacyjne jest elementem umożliwiającym transmisję informacji między interfejsami odległych węzłów. Wyróżnia się łącza jedno- i dwukierunkowe. Wyposażone są one w *bufory* o określonej *pojemności* (ang. links capacity).

Jeżeli łącze nie posiada buforów (jego pojemność jest równa zero), to mówimy o ***łączu niebuforowanym,*** w przeciwnym razie – o ***buforowanym***.

Zwykle kolejność odbierania komunikatów wysyłanych z danego węzła jest zgodna z kolejnością ich wysłania, wówczas łącze nazywamy ***łączem FIFO,*** w przeciwnym razie – ***nonFIFO***.

Łącza (2)

Łącza mogą gwarantować również, w sposób niewidoczny dla użytkownika, że żadna wiadomość nie jest tracona, duplikowana lub zmieniana – są to tzw. ***łącza niezawodne*** (ang. reliable, lossless, duplicate free, error free, uncorrupted, no spurious).

Czas transmisji w łączu niezawodnym (ang. transmission delay, in-transit time) może być ograniczony lub jedynie określony jako skończony lecz nieprzewidywalny. W pierwszym przypadku mówimy o ***transmisji synchronicznej*** lub ***z czasem deterministycznie ograniczonym*** (w szczególności równym zero), a w drugim – o ***transmisji asynchronicznej*** lub ***z czasem niedeterministycznym.***

Struktura środowiska przetwarzania

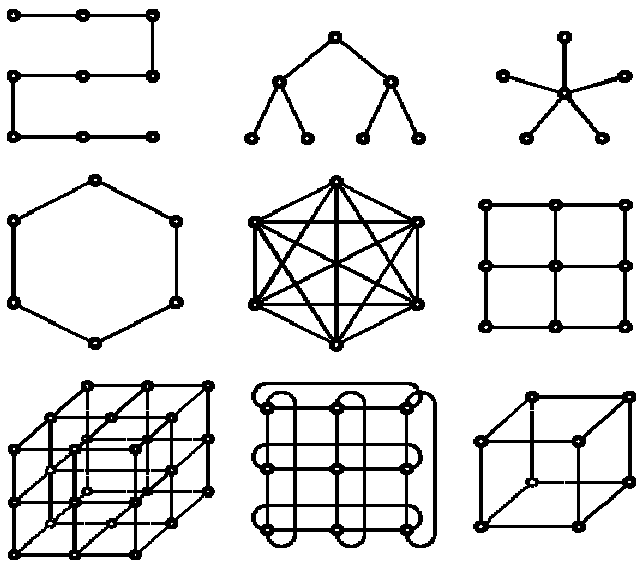
Struktura środowiska przetwarzania rozproszonego jest często przedstawiana jako graf:

$$\mathcal{G} = \langle \mathcal{V}, \mathcal{A} \rangle$$

w którym :

- wierzchołki grafu $V_i \in \mathcal{V}$ reprezentują jednostki przetwarzające $N_i \in \mathcal{N}$,
- krawędzie $(V_i, V_j) \in \mathcal{A}$, $\mathcal{A} \subseteq \mathcal{V} \times \mathcal{V}$, grafu niezorientowanego lub łuki $\langle V_i, V_j \rangle \in \mathcal{A}$ grafu zorientowanego, reprezentują odpowiednio łącza dwu- lub jednokierunkowe.

Przykłady topologii środowiska rozproszonego



Przetwarzanie rozproszone

Procesem rozproszonym (*przetwarzaniem rozproszonym*) nazywamy współbieżne i skoordynowane (ang. concurrent and coordinated) wykonanie w środowisku rozproszonym zbioru $\mathcal{P}$, procesów sekwencyjnych P_1 , P_2 , P_3 , ... , P_n , współdziałających w realizacji wspólnego celu przetwarzania.

Proces sekwencyjny

Nieformalnie, każdy ***proces sekwencyjny*** jest działaniem wynikającym z wykonywania w pewnym środowisku (kontekście) *programu sekwencyjnego* (*algorytmu sekwencyjnego*), który składa się z ciągu *operacji* (*instrukcji, wyrażeń*) atomowych (nieprzerywalnych).

Wyróżnia się dwie podstawowe klasy operacji:

- ***wewnętrzne*** (ang. internal)
  odnoszą się tylko do zmiennych lokalnych programu
- ***komunikacyjne*** (ang. communication)
  odnoszą się do środowiska i dotyczą komunikatów (ang. messages) oraz kanałów (ang. channels)

Komunikaty

Komunikat *(wiadomość)* jest dynamiczną strukturą danych:

$$M = \langle\, tag, mId, sId, rId, data \,\rangle$$

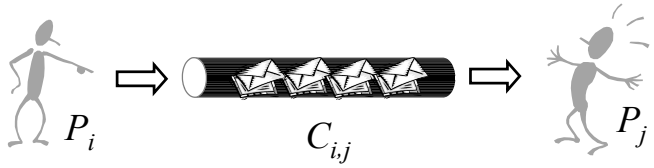
- identyfikator typu wiadomości
- identyfikator wiadomości
- identyfikator procesu *nadawcy* (ang. sender)
- identyfikator procesu *odbiorcy* (ang. receiver)
- dane

Kanały (1)

Kanał jest obiektem (zmienną) skojarzonym z uporządkowaną parą procesów $\langle P_i, P_j \rangle$, modelującym jednokierunkowe łącze transmisyjne.

Typem tego obiektu jest zbiór wiadomości, którego rozmiar nazywany jest ***pojemnością kanału***.

Kanał skojarzony z parą procesów $\langle P_i, P_j \rangle$, oznaczamy przez $C_{i,j}$ oraz nazywamy *kanałem incydentnym* z procesem P_i i z procesem P_j. Ponadto, kanał $C_{i,j}$ nazywamy *kanałem wyjściowym* procesu P_i oraz *kanałem wejściowym* procesu P_j.



Kanały (2)

Zbiór wszystkich kanałów incydentnych procesu P_i oznaczymy przez $\mathcal{C}_i$, a zbiór kanałów wejściowych i wyjściowych tego procesu odpowiednio przez $\mathcal{C}_i^{IN}$ i $\mathcal{C}_i^{OUT}$. Tak więc:

$$\mathcal{C}_i = \mathcal{C}_i^{IN} \cup \mathcal{C}_i^{OUT}$$

Ponadto przyjmujemy:

$\mathcal{P}_i^{IN} = \{P_j : \langle P_j, P_i \rangle \in \mathcal{C}_i^{IN}\}$

zbiór sąsiednich procesów wejściowych procesu P_i

$\mathcal{P}_i^{OUT} = \{P_j : \langle P_i, P_j \rangle \in \mathcal{C}_i^{OUT}\}$

zbiór sąsiednich procesów wyjściowych procesu P_i

Stan kanału (1)

Przez ***stan $L_{i,j}$ kanału $C_{i,j}$*** rozumieć będziemy zbiór, lub uporządkowany zbiór, wiadomości wysłanych przez P_i lecz jeszcze nie odebranych przez proces P_j.

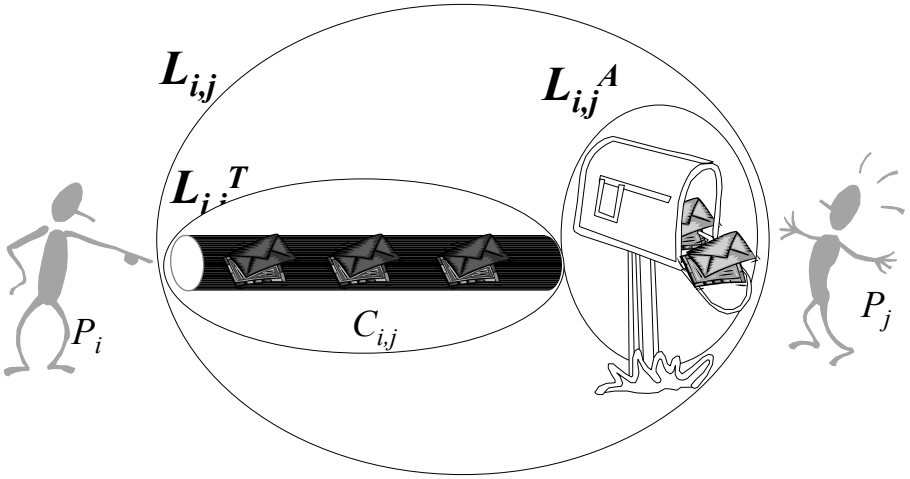
W celu modelowania w kanale opóźnień komunikacyjnych, w zbiorze wiadomości $L_{i,j}$ wyróżnia się dwa rozłączne podzbiory:

- ❑ zbiór wiadomości transmitowanych $L_{i,j}^T$ (ang. in-transit)
  zawiera wiadomości, które zostały wysłane przez proces P_i lecz nie dotarły jeszcze do węzła realizującego proces P_j i tym samym nie są dostępne (widoczne) dla P_j
- ❑ zbiór wiadomości bezpośrednio dostępnych $L_{i,j}^A$ (ang. available, arrived, ready)
  zawiera wiadomości wysłane przez P_i, które dotarły już (ang. arrived) do węzła procesu P_j i czekają w buforze kanału $C_{i,j}$ na ewentualne pobranie przez proces P_j.

Oczywiście, w każdej chwili

$$L_{i,j} = L_{i,j}^T \cup L_{i,j}^A$$

Stan kanału (2)



Predykaty opisujące stan kanału

$$\textbf{\textit{empty}}(C_{i,j}) \equiv L_{i,j} = \varnothing$$

$$\textbf{\textit{in-transit}}(C_{i,j}) \equiv L_{i,j}^{T} \neq \varnothing$$

$$\textbf{\textit{available}}(C_{i,j}) \equiv L_{i,j}^{A} \neq \varnothing$$

Indywidualne operacje komunikacyjne (1)

***send*(P_i, P_j, M)** – w procesie nadawcy P_i, operacja ta jest sparametryzowana dodatkowo przez pojedynczą wiadomość M i identyfikator procesu odbiorcy P_j (lub odpowiednio kanału $C_{i,j}$). Efektem wykonania tej operacji jest umieszczenie wiadomości M w kanale $C_{i,j}$, a więc wykonanie podstawienia $L_{i,j} := L_{i,j} \cup \{M\}$.

***receive*(P_i, P_j, inM)** – w procesie odbiorcy P_j, operacja ta jest sparametryzowana dodatkowo przez pojedynczą zmienną *inM* i identyfikator procesu P_i – *oczekiwanego nadawcy wiadomości*. Jeżeli kanał $C_{i,j}$ nie jest pusty i pewna wiadomość M jest bezpośrednio dostępna (*available*($C_{i,j}$) ma wartość *True*), to efektem wykonania tej operacji jest pobranie wiadomości M z kanału $C_{i,j}$, a więc wykonanie podstawienia $L_{i,j} := L_{i,j} \setminus \{M\}$ oraz $inM := M$.

Grupowe operacje komunikacyjne (2)

***send*(P_i, $\mathcal{P}_i^R$, M)** – w procesie nadawcy P_i, operacja ta jest sparametryzowana przez pojedynczą wiadomość M i zbiór procesów $\mathcal{P}_i^R \subseteq \mathcal{P}_i^{OUT}$, będących odbiorcami (adresatami) wiadomości M. Efektem wykonania tej operacji jest umieszczenie wiadomości we wszystkich kanałach $C_{i,j}$, takich że $P_j \in \mathcal{P}_i^R$, a więc podstawienie dla wszystkich tych kanałów: $L_{i,j} := L_{i,j} \cup \{M\}$.

***receive*($\mathcal{P}_j^S$, P_j, sInM)** – w procesie odbiorcy P_j, operacja ta jest sparametryzowana przez zmienną *sInM* (typu zbiór wiadomości) i zbiór procesów $\mathcal{P}_j^S \subseteq \mathcal{P}_j^{IN}$, będących *oczekiwanymi nadawcami* wiadomości. W ogólności, warunkiem wykonania operacji *receive*($\mathcal{P}_j^S$, P_j, *sInM*) jest jednoczesna dostępność wiadomości od wszystkich procesów $P_i \in \mathcal{P}_j^S$. Jeżeli warunek ten jest spełniony, to efektem wykonania operacji *receive*($\mathcal{P}_j^S$, P_j, *sInM*) jest atomowe pobranie wiadomości M_i od procesów $P_i \in \mathcal{P}_j^S$ i umieszczanie ich w *sInM*. Tym samym, dla każdego procesu $P_i \in \mathcal{P}_j^S$, wykonywane jest kolejno podstawienie $L_{i,j} := L_{i,j} \setminus \{M_i\}$ oraz $sInM := sInM \cup \{M_i\}$.

Rodzaje komunikacji

Kanały o niezerowej pojemności umożliwiają realizację następujących operacji komunikacji:

• ***nieblokowanych*** (***asynchronicznych***)

proces nadający przekazuje komunikat do kanału (bufora) i natychmiast kontynuuje swe działanie, a proces odbierający odczytuje stan kanału wejściowego, lecz nawet gdy kanał jest pusty, proces kontynuuje działanie;

• ***blokowanych*** (***synchronicznych***)

nadawca jest wstrzymywany do momentu, gdy wiadomość zostanie odebrana przez adresata, natomiast odbiorca – do momentu, gdy oczekiwana wiadomość pojawi się w jego buforze wejściowym.

W komunikacji *synchronicznej,* nadawca i odbiorca są blokowani aż odpowiedni odbiorca odczyta przesłaną do niego wiadomość (ang. rendezvous). W przypadku komunikacji *asynchronicznej*, nadawca lub odbiorca komunikuje się w sposób nieblokowany.

Model formalny procesu sekwencyjnego

Formalnie, proces sekwencyjny P_i może być opisany (modelowany) przez uporządkowaną czwórkę:

$$P_i = \langle \mathcal{S}_i, \mathcal{S}_i^0, \mathcal{E}_i, \mathcal{F} \rangle$$

gdzie

- $\mathcal{S}_i$ jest *zbiorem stanów* S_i procesu P_i,
- $\mathcal{S}_i^0$ jest *zbiorem stanów początkowych,* $\mathcal{S}_i^0 \subseteq \mathcal{S}_i$,
- $\mathcal{E}_i$ jest *zbiorem zdarzeń* procesu P_i,
- $\mathcal{F}$ jest *funkcją tranzycji*, taką że:
  $\mathcal{F}_i \subseteq \mathcal{S}_i \times \mathcal{E}_i \times \mathcal{S}_i$, a $\langle S, E, S' \rangle \in \mathcal{F}_i$,
  jeżeli zajście zdarzenia E w stanie S jest możliwe i prowadzi do zmiany stanu na S' .

Stan procesu

Stan $S_i(t)$ procesu w chwili t czasu lokalnego jest w ogólności zbiorem wartości wszystkich zmiennych lokalnych skojarzonych z procesem w chwili t oraz ciągów wiadomości wysłanych (wpisanych) do incydentnych kanałów wyjściowych i ciągów wiadomości odebranych z incydentnych kanałów wejściowych do chwili t.

Dla każdego t, $S_i(t) \in \mathcal{S}_i$. W celu uproszczenia notacji, zależność stanu od czasu można przyjąć za domyślną i jeśli nie prowadzi to do niejednoznaczności, oznaczać stan w pewnej chwili t przez S_i.

Zbiór $\mathcal{S}_i^0$ jest ***zbiorem stanów początkowych***, których wartości są zadawane wstępnie, bądź są wynikiem zajścia wyróżnionego *zdarzenia inicjującego* E_i^0.

Zdarzenie

Zdarzenie E_i^k odpowiada unikalnemu wykonaniu operacji atomowej, zmieniającemu stan S_i procesu i ewentualnie stan incydentnych z procesem kanałów $C_{i,j}$ lub $C_{j,i}$. Jeżeli operacja odpowiadająca zdarzeniu została wykonana, to powiemy, że *zdarzenie zaszło*.

Klasy zdarzeń

Wyróżnia się trzy podstawowe klasy zdarzeń:

- ❑ ***e_send***

- ❑ ***e_receive***

- ❑ ***e_internal***

Zdarzenie *e_send*

e_send (P_i, P_j, M)

zachodzi w procesie P_i, w wyniku wykonania przez ten proces operacji $send(P_i, P_j, M)$

e_send $(P_i, \mathcal{P}_i^R, M)$

zachodzi w procesie P_i, w wyniku wykonania przez ten proces operacji $send(P_i, \mathcal{P}_i^R, M)$

Zdarzenie *e_receive*

e_receive **(P_i, P_j, *M*)**

zachodzi w procesie P_j, gdy P_j wykonał operację *receive*($\mathcal{P}_j^S$, P_j, *sInM*), a odczytana do zmiennej lokalnej *sInM* wiadomość *M* pochodziła od procesu $P_i \in \mathcal{P}_j^S$

e_receive **($\mathcal{P}_j^S$, P_j, $\mathcal{M}_j^S$)**

zachodzi w procesie P_j, gdy P_j wykonał operację *receive*($\mathcal{P}_j^S$, P_j, *sInM*), a odczytane do zmiennej lokalnej *sInM* wiadomości $M_i \in \mathcal{M}_j^S$ pochodzą od procesów $P_i \in \mathcal{P}_j^S$

Zdarzenie *e_internal*

e_internal **(P_i, P_j, *M*)**

zachodzi gdy proces P_i wykonał operację, która nie zmienia stanu jego kanałów incydentnych. Do zdarzeń lokalnych zalicza się między innymi zdarzenia:

- *e_init*(P_i, S_i^k) – które nadaje procesowi P_i stan S_i^k (w szczególności stan początkowy)
- *e_stop*(P_i) – które kończy wykonywanie procesu

Dostępność wiadomości

Dostępność wiadomości utożsamiać można z zajściem zdarzeń w środowisku komunikacyjnym:

- *zdarzenie dostarczenia wiadomości M* ***e_deliver(P_i, P_j, M)***
- *zdarzenia nadejścia wiadomości M* ***e_arrive(P_i, P_j, M)***

Przez $\mathcal{P}_j^A$ oznaczać będziemy zbiór procesów, których wiadomości dotarły i są dostępne dla P_j.

Jeżeli proces odbiorcy P_j odczytuje skierowaną do niego wiadomość M, wykonując operację *receive(P_i, P_j, inM)*, wiadomość ta jest przepisywana (przemieszczana) z bufora wyjściowego łącza (bufora wyjściowego procesu P_i) do lokalnej zmiennej procesu *inM*. Jeśli przepisanie to nastąpiło, to mówimy, że ***wiadomość została odebrana*** lub, że zaszło ***zdarzenie odbioru e_receive (P_i, P_j, M)***

Funkcja tranzycji

Funkcja tranzycji $\mathcal{F}_i \subseteq \mathcal{S}_i \times \mathcal{E}_i \times \mathcal{S}_i$ opisuje reguły zmiany stanu S na S' w wyniku zajścia zdarzenia E. Elementy $\langle S, E, S' \rangle \in \mathcal{F}_i$ nazwiemy *tranzycjami* lub *krokami*. W zależności od zachodzącego zdarzenia E, tranzycję nazwiemy odpowiednio *tranzycją wejścia, wyjścia* lub *lokalną*.

Zdarzenia dopuszczalne i gotowe

Funkcja tranzycji dopuszcza możliwość zajścia zdarzenia E tylko w tych stanach S, dla których $\langle S, E, S' \rangle \in \mathcal{F}_i$. Dlatego też, w wypadku gdy $\langle S, E, S' \rangle \in \mathcal{F}_i$, powiemy że zdarzenie jest ***dopuszczalne*** (ang. allowed) w stanie S. Wprowadzimy też predykat *allowed*(E) oznaczający, że w danej chwili zdarzenie E jest dopuszczalne.

Oprócz czynnika wewnętrznego (stanu procesu), zajście zdarzenia może być dodatkowo uwarunkowane stanem kanałów wejściowych (środowiska). Jeśli zdarzenie może zajść ze względu na warunki zewnętrzne (stan kanałów), to powiemy że zdarzenie jest ***przygotowane*** lub ***gotowe*** (ang. ready). Fakt gotowości zdarzenia E w danej chwili wyrażać będzie predykat *ready*(E).

W tym kontekście wprowadzimy jeszcze predykat *enable*(E), oznaczający, że zdarzenie jest ***aktywne*** – czyli jednocześnie gotowe i dopuszczalne. Stąd też:

$$enable(E) \equiv ready(E) \wedge allowed(E)$$

Procesy zakończone, wstrzymane, aktywne, pasywne

Powiemy, że proces P_i jest w ***stanie końcowym*** $\boldsymbol{S_i^e}$, jeżeli zbiór zdarzeń dopuszczalnych w tym stanie jest pusty.

Jeżeli natomiast niepusty zbiór zdarzeń dopuszczalnych zawiera wyłącznie zdarzenia odbioru i żadne z tych zdarzeń nie jest aktywne (gotowe), to powiemy że proces jest ***wstrzymany*** (***zablokowany***).

Proces wstrzymany lub zakończony nazwiemy ***pasywnym***. Przez proces ***aktywny*** będziemy natomiast rozumieć proces, który nie jest pasywny.

Przyjmujemy, że w każdej chwili t stan procesu P_i reprezentuje zmienna logiczna $\boldsymbol{passive_i}$, przyjmująca wartość *True*, gdy proces P_i jest pasywny, a wartość *False*, gdy proces ten jest aktywny.

Procesy aktywne i pasywne

Aktywny proces P_i ($passive_i$ = *False*) może wysyłać i odbierać wiadomości, wykonywać tranzycje lokalne, a więc potencjalnie może również spontanicznie (w dowolnej chwili) zmienić swój stan na pasywny.

W stanie ***pasywnym*** procesu P_i ($passive_i$ = *True*) dopuszczalne są natomiast co najwyżej zdarzenia odbioru.

Zmiana stanu procesu z pasywnego na aktywny uwarunkowana jest osiągnięciem gotowości przez choćby jedno z dopuszczalnych zdarzeń odbioru, czyli spełnieniem tak zwanego *warunku uaktywnienia*.

Warunek uaktywnienia (1)

Warunek uaktywnienia (ang. activation condition) procesu P_i związany jest ze zbiorem warunkującym $\mathcal{D}_i$, zbiorem $\mathcal{P}_i^A$, oraz predykatem $activate_i(\mathcal{X})$.

Zbiór warunkujący (ang. dependent set), jest sumą mnogościową zbiorów $\mathcal{P}_i^S$ wszystkich zdarzeń odbioru dopuszczalnych w danej chwili.

Predykat $\boldsymbol{activate_i(\mathcal{X})}$ zdefiniowany jest w sposób następujący:

- Jeżeli $\mathcal{X} = \mathcal{D}_i$, to $activate_i(\mathcal{X}) = True$
- Jeżeli $\mathcal{X} = \varnothing$, to $activate_i(\mathcal{X}) = False$
- Jeżeli $\mathcal{X} \subset \mathcal{D}_i$ i $\mathcal{X} \neq \varnothing$, to:
  $activate_i(\mathcal{X}) \equiv \exists \mathcal{X}' :: \mathcal{X}' \neq \varnothing \wedge \mathcal{X}' \subseteq \mathcal{X} \wedge (\mathcal{P}_i^A = \mathcal{X}' \Rightarrow (passive_i \rightsquigarrow \neg passive_i))$

gdzie

$passive_i \rightsquigarrow \neg passive_i$ oznacza, że pasywny proces P_i zmieni swój stan na aktywny w skończonym choć nieprzewidywalnym czasie.

Warunek uaktywnienia (2)

Warunek uaktywnienia **procesu** P_i formalnie wyraża predykat $ready_i(\mathcal{X})$:

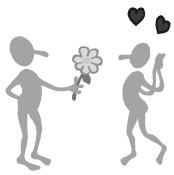
$$ready_i(\mathcal{X}) \equiv (\mathcal{P}_i^A \supseteq \mathcal{X}) \wedge activate_i(\mathcal{X})$$

Gdy proces jest uaktywniany, to wiadomości, których dostarczanie doprowadziło do spełnienia warunku uaktywnienia, są atomowo pobierane z buforów wejściowych i dalej przetwarzane.

Klasyczne warunki uaktywnienia – modele żądań

W praktyce, warunki uaktywnienia wyrażane są często w kategoriach tak zwanego modelu żądań (ang. request model), pozwalającego na zwięzłe sformułowanie warunków uaktywnienia specyficznych dla określonej klasy zastosowań. Podejście takie odpowiada rozwiązaniom, w których procesy w trakcie wykonywania kierują do innych procesów żądania przydziału ogólnie rozumianych zasobów, lub podjęcia określonych działań. Spełnienie żądania może być potwierdzone przez przesłanie stosownej wiadomości. Tym samym, istotą modelu żądań jest wysyłanie żądań w formie wiadomości i oczekiwanie na określony zbiór wiadomości potwierdzających realizację żądania.

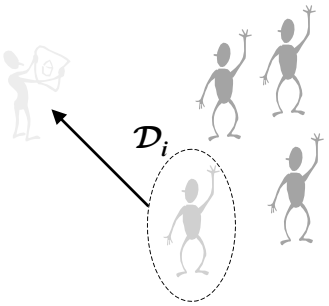
- Model jednostkowy
- Model *AND*
- Model *OR*
- Podstawowy model „*k spośród r*”
- Model *OR-AND*
- Dysjunkcyjny model „*k spośród r*”
- Model predykatowy



Model jednostkowy

W *modelu jednostkowym* warunkiem uaktywnienia pasywnego procesu jest przybycie wiadomości od jednego, ściśle określonego nadawcy.

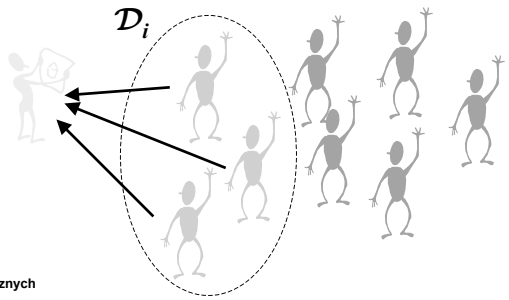
W tym przypadku $|\mathcal{D}_i| = 1$, dla każdego naturalnego i, $1 \leqslant i \leqslant n$. Model ten odpowiada szerokiej klasie systemów, w których procesy żądają kolejno po jednym tylko zasobie.



Model *AND*

W ***modelu AND*** proces pasywny staje się aktywnym, jeżeli dotarły wiadomości od wszystkich procesów tworzących zbiór warunkujący.

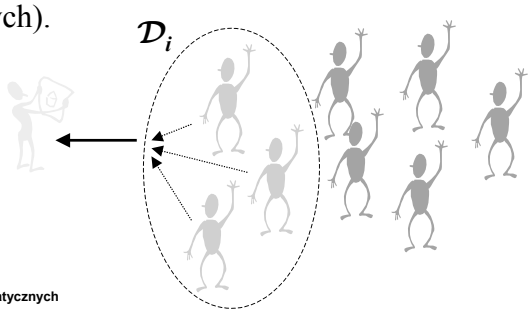
Model ten nazywany jest również *modelem zasobowym*. Jest on stosowany w rozwiązaniach, w których procesy ubiegają się o pewne wieloelementowe zbiory zasobów.



Model *OR*

W ***modelu OR*** do uaktywnienia procesu wystarczy jedna wiadomość od któregokolwiek z procesów ze zbioru warunkującego.

Model ten nazywany jest również *modelem komunikacyjnym*. Odpowiada on alternatywnym strukturom programowym dostępnym między innymi w językach *CSP* i *ADA*, a także – ubieganiu się o zasoby z pewnej puli zasobów równoważnych (alternatywnych).

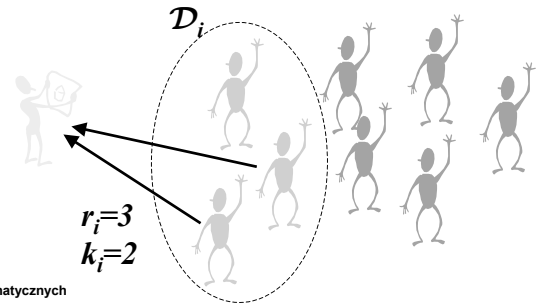


 Slajd 45

Podstawowy model *k spośród r*

W ***podstawowym modelu k spośród r***, z pasywnym procesem P_i skojarzony jest zbiór warunkujący $\mathcal{D}_i$, liczba naturalna k_i, $1 \leqslant k_i \leqslant |\mathcal{D}_i|$, oraz liczba $r_i = |\mathcal{D}_i|$. W modelu tym proces staje się aktywny tylko wówczas, gdy uzyska wiadomości od co najmniej k_i różnych procesów ze zbioru warunkującego $\mathcal{D}_i$.

Model ***k spośród r*** jest uogólnieniem modelu *AND* ($k_i=r_i$) oraz modelu *OR* ($k_i=1$).



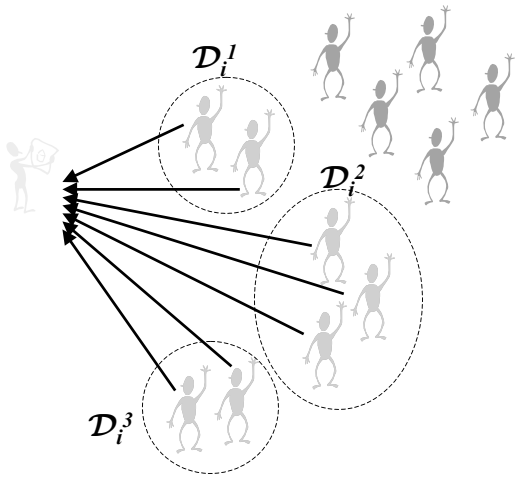
 Slajd 46

Model *OR-AND* (1)

W ***modelu OR-AND*** zbiór warunkujący $\mathcal{D}_i$ pasywnego procesu P_i jest zdefiniowany jako $\mathcal{D}_i^1 \cup \mathcal{D}_i^2 \cup ... \cup \mathcal{D}_i^{q_i}$, gdzie dla każdego naturalnego u, $1 \leqslant u \leqslant q_i$, $\mathcal{D}_i^u \subseteq \mathcal{P}$. Proces staje się aktywny po otrzymaniu wiadomości od każdego z procesów tworzących zbiór $\mathcal{D}_i^1$ lub od każdego z procesów tworzących zbiór $\mathcal{D}_i^2$ lub ... lub od każdego z procesów tworzących zbiór $\mathcal{D}_i^{q_i}$.

Model *OR-AND* jest ogólniejszy od podstawowego modelu *"k spośród r"*. Model ten redukuje się do modelu *"k spośród r"*, gdy podzbiory $\mathcal{D}_i^u$ odpowiadają wszystkim możliwym podzbiorom o liczności k zbioru warunkującego $\mathcal{D}_i$ o liczności r.

Model *OR-AND* (2)



Dysjunkcyjny model *k spośród r*

W *modelu dysjunkcyjnym k spośród r* z każdym pasywnym procesem P_i skojarzony jest zbiór warunkujący $\mathcal{D}_i=\mathcal{D}_i^1\cup\mathcal{D}_i^2\cup...\cup\mathcal{D}_i^{q_i}$, liczby naturalne $k_i^1, k_i^2, ..., k_i^{q_i}$, oraz liczby naturalne $r_i^1, r_i^2, ..., r_i^{q_i}$, gdzie dla każdego naturalnego u, $1\leqslant u\leqslant q_i$, $\mathcal{D}_i^u\subseteq\mathcal{P}$, $1\leqslant k_i^u\leqslant r_i^u=|\mathcal{D}_i^u|$. Proces staje się aktywny po otrzymaniu wiadomości od k_i^1 różnych procesów ze zbioru $\mathcal{D}_i^1$, lub k_i^2 wiadomości od różnych procesów ze zbioru $\mathcal{D}_i^2$, lub ... lub k_i^u wiadomości od różnych procesów ze zbioru $\mathcal{D}_i^{q_i}$.

Model ***dysjunkcyjny k spośród r*** redukuje się do:

- modelu *OR-AND,* gdy $k_i^u=|\mathcal{D}_i^u|$ dla każdego u;
- modelu podstawowego *k spośród r,* gdy $q_i=1$;
- modelu *AND,* gdy $q_i=1$ i $k_i^1=|\mathcal{D}_i^1|$;
- modelu *OR,* gdy $q_i=1$ i $k_i^1=1$

Model predykatowy

W *modelu predykatowym*, dla każdego pasywnego procesu P_i ze zbiorem warunkującym $\mathcal{D}_i$ określony jest predykat $activate_i(\mathcal{X})$, gdzie $\mathcal{X}\subseteq\mathcal{D}_i$.

Jak łatwo zauważyć, stosownie definiując predykat $activate_i(\mathcal{X})$ można oczywiście uzyskać wszystkie wcześniej omówione modele żądań.

Wykonanie procesu

Częściowym wykonaniem (ang. partial run) procesu $P_i=\langle \mathcal{S}_i, \mathcal{S}_i^0, \mathcal{E}_i, \mathcal{F}_i\rangle$ nazwiemy ciąg S_i^0, E_i^1, S_i^1, E_i^2, S_i^2, ..., S_i^s, E_i^{s+1}, S_i^{s+1}, składający się naprzemiennie ze stanów i zdarzeń, taki że dla każdego naturalnego u, $0 \leqslant u \leqslant s$, $\langle S_i^u, E_i^{u+1}, S_i^{u+1}\rangle \in \mathcal{F}_i$. Należy zaznaczyć, że każde częściowe wykonanie kończy się stanem.

Powiemy, że ***stan S_i' procesu P_i jest osiągalny ze stanu S_i***, co oznaczać będziemy $S_i \rightsquigarrow S_i'$, jeżeli istnieje częściowe wykonanie S_i^0, E_i^1, S_i^1, E_i^2, S_i^2, ..., S_i^s, E_i^{s+1}, S_i^{s+1} procesu P_i, takie że $S_i=S_i^0$, a $S_i'= S_i^{s+1}$.

Przez ***wykonanie*** (*realizację*) R_i procesu P_i (ang. run, execution) rozumieć będziemy natomiast częściowe wykonanie rozpoczynające się stanem początkowym $S_i^0 \in \mathcal{S}_i^0$. Przez $\mathcal{R}_i$ oznaczany jest zbiór wszystkich wykonań procesu P_i.

Jeżeli istnieje wykonanie procesu P_i, takie że stan S jest stanem końcowym, to stan ten nazwiemy ***osiągalnym*** lub ***spójnym*** (ang. reachable, consistent).

Historia procesu

Każdemu wykonaniu R_i procesu P_i odpowiada ciąg stanów S_i^0, S_i^1, S_i^2, ..., S_i^s, S_i^{s+1} nazywany ***śladem wykonania (realizacji) procesu***, oraz ciąg zdarzeń E_i^0, E_i^1, E_i^2, ..., E_i^s, E_i^{s+1} nazywany ***historią wykonania (realizacji) procesu***.

W historii procesu występuje zawsze ***zdarzenie wstępne E_i^0*** = $e\_init(P_i, S_i^0)$. Historię E_i^0, E_i^1, E_i^2, ..., E_i^s procesu oznaczamy przez H_i^s a zbiór historii przez $\mathcal{H}_i^s$. Stąd, $H_i^s = E_i^0, E_i^1, E_i^2, ..., E_i^s$ oraz $H_i^s \in \mathcal{H}_i^s$.

Stan lokalny a historia wykonania

Ponieważ zdarzenia odpowiadają wykonaniu operacji atomowych, można przyjąć, że zachodzą one (są wykonywane) natychmiastowo (w czasie zerowym). Wówczas, zajście dwóch kolejnych zdarzeń E_i^s, E_i^{s+1} procesu P_i określa interwał czasu między tymi zdarzeniami. Stan lokalny procesu P_i w każdej chwili tego interwału jest zdefiniowany przez częściową historię H_i^s, w tym przez ostatnie zdarzenie częściowej historii E_i^s. Oznaczmy ten stan przez S_i^s.

Stan S_i^s możemy utożsamić z lokalną historią H_i^s i powiedzieć, że zdarzenie E_i^s należy do stanu lokalnego S_j^l wtedy i tylko wtedy, gdy $i = j$ oraz $s \leqslant l$.

Proces rozproszony (1)

Proces rozproszony $\boldsymbol{\Pi}$, będący współbieżnym wykonaniem zbioru $\boldsymbol{\mathcal{P}} = \{P_1, P_2, ..., P_n\}$ procesów sekwencyjnych P_i, opisuje uporządkowana czwórka $\Pi = \langle \boldsymbol{\Sigma}, \boldsymbol{\Sigma}^0, \boldsymbol{\Lambda}, \boldsymbol{\Phi} \rangle$, gdzie:

- $\boldsymbol{\Sigma}$ – jest zbiorem *stanów globalnych* Σ procesu rozproszonego, $\boldsymbol{\Sigma} \subseteq \boldsymbol{\mathcal{S}}_1 \times \boldsymbol{\mathcal{S}}_2 \times ... \times \boldsymbol{\mathcal{S}}_n$,
- $\boldsymbol{\Sigma}^0$ – jest zbiorem *stanów początkowych*, $\boldsymbol{\Sigma}^0 \subseteq \boldsymbol{\mathcal{S}}_1^0 \times \boldsymbol{\mathcal{S}}_2^0 \times ... \times \boldsymbol{\mathcal{S}}_n^0$,
- $\boldsymbol{\Lambda}$ – jest zbiorem *zdarzeń,* $\boldsymbol{\Lambda} = \boldsymbol{\mathcal{E}}_1 \cup \boldsymbol{\mathcal{E}}_2 \cup ... \cup \boldsymbol{\mathcal{E}}_n$;
- $\boldsymbol{\Phi}$ – jest *funkcją tranzycji*, taką że $\boldsymbol{\Phi} \subseteq \boldsymbol{\Sigma} \times \boldsymbol{\Lambda} \times \boldsymbol{\Sigma}$.

Proces rozproszony (2)

Zbiór stanów globalnych $\boldsymbol{\Sigma}$ jest podzbiorem iloczynu kartezjańskiego $\mathcal{S}_1 \times \mathcal{S}_2 \times ... \times \mathcal{S}_n$ zbiorów stanów lokalnych procesów składowych przetwarzania rozproszonego. Stan globalny procesu rozproszonego Π w chwili τ czasu globalnego (czasu postrzeganego przez teoretycznego zewnętrznego obserwatora), oznaczony przez $\Sigma(\tau)$, jest więc uporządkowanym zbiorem stanów lokalnych wszystkich procesów składowych w chwili τ :

$$\Sigma(\tau) = \langle S_1(\tau), S_2(\tau), ..., S_n(\tau) \rangle$$

Zbiór globalnych stanów początkowych $\boldsymbol{\Sigma^0}$ jest podzbiorem iloczynu kartezjańskiego $\mathcal{S}_1^{\,0} \times \mathcal{S}_2^{\,0} \times ... \times \mathcal{S}_n^{\,0}$ zbiorów stanów początkowych procesów składowych.

Zbiór zdarzeń globalnych $\boldsymbol{\Lambda}$ jest sumą mnogościową $\mathcal{E}_1 \cup \mathcal{E}_2 \cup ... \cup \mathcal{E}_n$ zbiorów zdarzeń procesów składowych.

Globalna funkcja tranzycji $\boldsymbol{\Phi}$ jest złożeniem funkcji tranzycji procesów składowych, a więc jest zbiorem trójek $\langle \Sigma, E, \Sigma' \rangle$, dla których istnieje takie k, $1 \leqslant k \leqslant n$, że $\Sigma = \langle S_1, S_2, ... , S_k, ... , S_n \rangle$, $\Sigma' = \langle S_1, S_2, ... , S_k', ... , S_n \rangle$, oraz $\langle S_k, E, S_k' \rangle \in \mathcal{F}_k$.

Wykonanie częściowe, stan osiągalny

Częściowe wykonanie procesu rozproszonego $\Pi = \langle \boldsymbol{\Sigma}, \boldsymbol{\Sigma^0}, \boldsymbol{\Lambda}, \boldsymbol{\Phi} \rangle$ utożsamia się z ciągiem $\Sigma^0, E^1, \Sigma^1, E^2, ... , \Sigma^s, E^{s+1}, \Sigma^{s+1}$, składającym się naprzemiennie ze stanów i zdarzeń, takim że dla każdego u, $0 \leqslant u \leqslant s$, $\langle \Sigma^u, E^{u+1}, \Sigma^{u+1} \rangle \in \boldsymbol{\Phi}$.

Powiemy, że stan Σ' procesu jest ***osiągalny*** ze stanu Σ, co oznaczymy przez $\boldsymbol{\Sigma \rightsquigarrow \Sigma'}$, jeżeli istnieje częściowe wykonanie $\Sigma^0, E^1, \Sigma^1, E^2, ... , \Sigma^s, E^{s+1}, \Sigma^{s+1}$ procesu Π, takie że $\Sigma = \Sigma^0$ a $\Sigma' = \Sigma^{s+1}$.

Przez ***wykonanie*** (***realizację***) Υ procesu Π rozumieć będziemy natomiast częściowe wykonanie rozpoczynające się stanem początkowym $\Sigma^0 \in \boldsymbol{\Sigma^0}$.

Oznaczmy przez $\boldsymbol{\Upsilon}$ zbiór wszystkich możliwych wykonań (realizacji) procesu Π. Jeżeli istnieje wykonanie procesu Π, takie że Σ jest stanem końcowym, to stan ten nazwiemy ***globalnym stanem osiągalnym*** (***spójnym***) procesu rozproszonego Π (ang. reachable, consistent).

Każdemu wykonaniu $\Upsilon \in \boldsymbol{\Upsilon}$ procesu Π, odpowiada pewien ciąg stanów $\Sigma^0, \Sigma^1, \Sigma^2, ... , \Sigma^s, \Sigma^{s+1}$, nazywany ***śladem wykonania (realizacji) procesu*** $\boldsymbol{\Pi}$, oraz ciąg zdarzeń $E^0, E^1, E^2, ... , E^s, E^{s+1}$, nazywany ***historią wykonania (realizacji) procesu***. Historię $E^0, E^1, E^2, ... , E^s$, oznaczamy przez Ξ^s, a zbiór historii – przez $\boldsymbol{\Xi}$.

Topologia przetwarzania rozproszonego

Proces rozproszony jest często przedstawiany jako graf:

$$\mathcal{G} = \langle \mathcal{V}, \mathcal{A} \rangle$$

w którym :

- wierzchołki grafu $V_i \in \mathcal{V}$ reprezentują procesy składowe $P_i \in \mathcal{P}$ przetwarzania rozproszonego,
- krawędzie $(V_i, V_j) \in \mathcal{A}$, $\mathcal{A} \subseteq \mathcal{V} \times \mathcal{V}$, grafu niezorientowanego lub łuki $\langle V_i, V_j \rangle \in \mathcal{A}$ grafu zorientowanego, reprezentują kanały $C_{i,j}$ odpowiednio dwu lub jednokierunkowe.

Tak zdefiniowany graf nazywany jest ***grafem procesu rozproszonego*** lub ***topologią przetwarzania*** (*topologią procesu rozproszonego)*.

Relacja poprzedzania zdarzeń

Oznaczmy przez $\mapsto$ ***relację poprzedzania*** (ang. happen before, causal precedence, happened before) zdefiniowaną na zbiorze Λ w następujący sposób:

$E_i^k \mapsto E_j^l \Leftrightarrow$

1) $i = j \ \wedge \ k < l$, lub
2) $i \neq j$ oraz E_i^k jest zdarzeniem *e_send*(P_i, P_j, M) wysłania wiadomości M, a zdarzenie E_j^l jest zdarzeniem *e_receive*(P_i, P_j, M) odbioru tej samej wiadomości, lub
3) istnieje sekwencja zdarzeń $E^0, E^1, E^2, \ldots, E^s$, taka że $E^0 = E_i^k$, $E^s = E_j^l$ i dla każdej pary $\langle E^u, E^{u+1} \rangle$ gdzie $0 \leqslant u \leqslant s\text{-}1$, zachodzi 1) albo 2).

Relacja poprzedzania jest ***antysymetryczna*** i ***przechodnia,*** a więc jest relacją *częściowego porządku*.

Relacja poprzedzania lokalnego

Przez $\mapsto_i$ oznaczamy ***relację poprzedzania lokalnego*** zdarzeń procesu, taką że: $E_i^k \mapsto_i E_j^l$ wtedy i tylko wtedy, gdy $i=j$ oraz $k<l$ (lub gdy $i=j$ oraz $E_i^k \rightarrow E_j^l$).

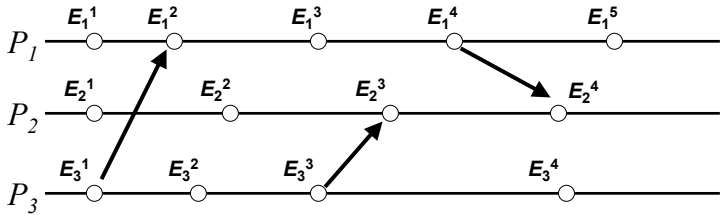
Zdarzenia E_i^k i E_j^l nazywamy ***przyczynowo-zależnymi***, jeżeli:

$$E_i^k \mapsto E_j^l \quad \text{albo} \quad E_j^l \mapsto E_i^k .$$

W przeciwnym razie zdarzenie te nazwiemy ***przyczynowo-niezależnymi*** lub *współbieżnymi* (ang. concurrent, causally independent), co będziemy oznaczać przez $E_i^k \parallel E_j^l$.

Diagramy przestrzenno-czasowe

Realizację przetwarzania rozproszonego można przedstawić graficznie w postaci ***diagramu przestrzenno-czasowego*** (ang. space-time diagram), w którym osie reprezentują upływ czasu globalnego, a punkty na osiach – zdarzenia.



$E_1^1 \parallel E_2^1$, $E_1^1 \parallel E_2^2$, $E_1^1 \parallel E_3^1$, $E_1^1 \parallel E_3^4$,
$E_3^1 \mapsto E_1^2$, $E_3^3 \mapsto E_2^3$, $E_1^4 \mapsto E_2^4$, $E_3^1 \mapsto E_1^4$,
$E_3^1 \mapsto E_2^4$, $E_3^2 \mapsto E_2^4$

Relacja poprzedzania stanów lokalnych

Przez analogię do relacji na zbiorze zdarzeń, można zdefiniować częściowy porządek na zbiorze stanów wszystkich procesów $P_i \in \mathcal{P}$ w sposób następujący:

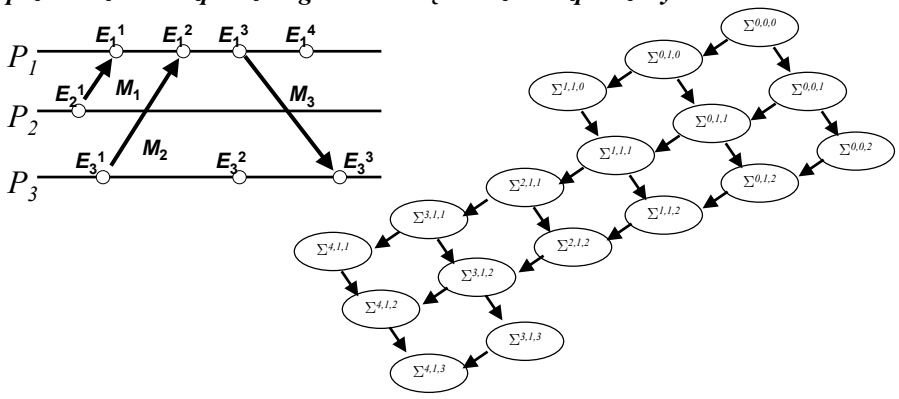
$$S_i^k \mapsto S_j^l \Leftrightarrow \begin{cases} E_i^{k+1} \mapsto E_j^l, \textit{ lub} \\ E_i^{k+1} = E_j^l \end{cases}$$

Relacja powyższa oznacza, że stan jednego procesu poprzedza przyczynowo stan innego, wtedy i tylko wtedy, gdy zdarzenie rozpoczynające drugi stan zależy przyczynowo bądź jest tożsame ze zdarzeniem kończącym pierwszy stan.

Stany lokalne, dla których nie zachodzi ani relacja $S_i^k \mapsto S_j^l$ ani też relacja $S_j^l \mapsto S_i^k$, nazywamy ***współbieżnymi***.

Graf stanów osiągalnych przetwarzania rozproszonego

Zbiór częściowo uporządkowany $\langle \Lambda, \mapsto \rangle$ może być przedstawiony w postaci grafu zorientowanego, w którym wierzchołki odpowiadają stanom Σ, a łuki $\langle \Sigma^k, \Sigma^l \rangle$ oznaczają istnienie zdarzenia dopuszczalnego takiego, że $\langle \Sigma^k, E, \Sigma^l \rangle \in \Phi$. Graf taki, będziemy nazywać ***grafem stanów osiągalnych przetwarzania rozproszonego*** lub ***siatką obliczeń rozproszonych***.



Niedeterminizm przetwarzania (1)

W kontekście grafu stanów osiągalnych przetwarzania rozproszonego, każda realizacja przetwarzania rozproszonego jest pewną ścieżką w tym grafie. Istnienie wielu różnych ścieżek ilustruje ***niedeterminizm*** przetwarzania rozproszonego, oznaczający że dla danego stanu może istnieć wiele stanów następnych.

Lokalne zdarzenie procesu jest ***niedeterministyczne***, gdy jego zajście może być zastąpione przez zajście innego zdarzenia i wybór ten nie jest przewidywalny. Jeżeli przykładowo sekwencyjne wykonanie procesu może być w każdej chwili zmienione w wyniku zajścia przerwania zewnętrznego, to wszystkie zdarzenia tego procesu są niezdeterminowane.

Niedeterminizm przetwarzania (2)

Przetwarzanie nazywamy ***zdeterminowanym*** jeżeli wszystkie zdarzenia są zdeterminowane. W przeciwnym wypadku, przetwarzanie nazywamy ***niedeterministycznym***. W ramach niedeterministycznego przetwarzania rozproszonego wyróżnia się podklasę przetwarzania ***quasi deterministycznego*** (ang. quasi-deterministic, piece-wice deterministic, event-driven), w której niedeterminizm jest wyłącznie konsekwencją niedeterminizmu operacji odbioru.

Należy zauważyć, że w ogólności istnieje wiele różnych diagramów przestrzenno-czasowych, którym odpowiada taki sam zbiór częściowo uporządkowany $\langle \Lambda, \mapsto \rangle$. Diagramy takie nazywa się ***diagramami równoważnymi***.

Konfiguracja spójna

Iloczyn kartezjański $\boldsymbol{\mathcal{S}}_1 \times \boldsymbol{\mathcal{S}}_2 \times ... \times \boldsymbol{\mathcal{S}}_n$ będziemy oznaczać przez $\boldsymbol{\Gamma}$ i nazywać ***zbiorem konfiguracji*** (*obrazów stanu globalnego*) procesu rozproszonego $\Pi = \langle \boldsymbol{\Sigma}, \boldsymbol{\Sigma}^0, \boldsymbol{\Lambda}, \boldsymbol{\Phi} \rangle$. Stąd, konfiguracja $\Gamma \in \boldsymbol{\Gamma}$ jest wektorem $\langle S_1^{k1}, S_2^{k2}, ..., S_n^{kn} \rangle$ stanów lokalnych (historii lokalnych) wszystkich procesów $P_1, P_2, ..., P_n$, takim że dla każdego u, $1 \leqslant u \leqslant n$, $S_u^{ku} \in \boldsymbol{\mathcal{S}}_u$. Łatwo zauważyć, że $\boldsymbol{\Gamma}$ zawiera zbiór stanów osiągalnych procesu Π.

Konfigurację Γ nazwiemy ***konfiguracją spójną*** lub ***obrazem spójnym*** jeżeli $\forall E \; \forall E'$ zachodzi:

$$(E' \in \Gamma \wedge E \mapsto E') \Rightarrow (E \in \Gamma)$$

Twierdzenie
Konfiguracja $\Gamma = \langle S_1^{k1}, S_2^{k2}, ... , S_n^{kn} \rangle$, reprezentująca stan osiągalny przetwarzania rozproszonego Π jest konfiguracją spójną.

Twierdzenie
Jeżeli $\Gamma = \langle S_1^{k1}, S_2^{k2}, ... , S_n^{kn} \rangle$ jest konfiguracją spójną, w której lokalne stany składowe S_u^{ku} są osiągalne w pewnej realizacji przetwarzania rozproszonego Π, to istnieje stan osiągalny $\Sigma(\tau)$, taki że dla każdego, $1 \leqslant u \leqslant n$, $S_u(\tau) = S_u^{ku}$.

Odcięcie spójne (1)

Oznaczmy przez $\sigma_1^{k1}, \sigma_2^{k2}, ..., \sigma_n^{kn}$ ciąg wybranych punktów (zdarzeń pozornych) odcinków czasu odpowiadających stanom $S_1^{k1}, S_2^{k2}, ..., S_n^{kn}$ poszczególnych procesów. Linię łamaną łączącą punkty $\sigma_1^{k1}, \sigma_2^{k2}, ..., \sigma_n^{kn}$ nazywać będziemy ***linią odcięcia***. Linia odcięcia dzieli zbiór zdarzeń na ***przeszłość*** (te zdarzenia, które zaszły przed linią odcięcia) i ***przyszłość*** (te zdarzenia, które zaszły po linii odcięcia).

Odcięciem $\boldsymbol{\Psi}$ zbioru zdarzeń $\boldsymbol{\Lambda}$ nazwiemy skończony zbiór $\Psi \subseteq \boldsymbol{\Lambda}$, taki że:

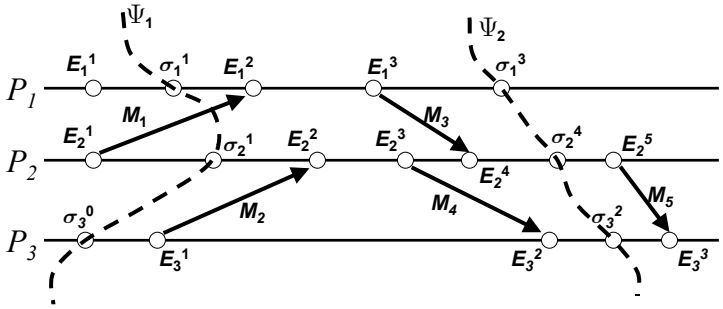
$$(E' \in \Psi \wedge E \mapsto_i E') \Rightarrow (E \in \Psi)$$

Powiemy, że odcięcie Ψ_2 jest ***późniejsze*** od odcięcia Ψ_1, jeżeli $\Psi_1 \subseteq \Psi_2$.

Odcięcie Ψ zbioru zdarzeń $\boldsymbol{\Lambda}$ nazwiemy ***odcięciem spójnym***, gdy:

$$(E' \in \Psi \wedge E \mapsto E') \Rightarrow (E \in \Psi)$$

Odcięcie spójne – przykład



Odcięcie spójne (2)

Twierdzenie

Dla odcięcia spójnego z linią odcięcia σ_1^{k1} , σ_2^{k2} , ... , σ_n^{kn} żadna para stanów (zdarzeń) odpowiadających linii odcięcia nie jest wzajemnie zależna.

Twierdzenie

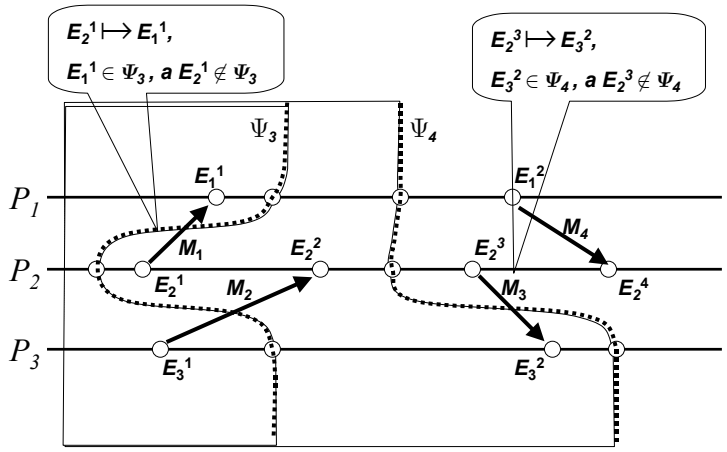
Dla każdego diagramu przestrzenno-czasowego z odcięciem spójnym określonym przez linię odcięcia σ_1^{k1} , σ_2^{k2} , ... , σ_n^{kn} istnieje równoważny diagram przestrzenno-czasowy, w którym linię odcięcia σ_1^{k1} , σ_2^{k2} , ... , σ_n^{kn} tworzą zdarzenia równoczesne w sensie czasu globalnegoτ.

Zgodnie z definicją, każdemu odcięciu Ψ opisanemu przez linię odcięcia σ_1^{k1} , σ_2^{k2} , ..., σ_n^{kn} odpowiada konfiguracja $\Gamma = \langle S_1^{k1}, S_2^{k2}, \ldots, S_n^{kn} \rangle$.

Twierdzenie

Niech Γ będzie konfiguracją a Ψ odpowiadającym jej odcięciem. Konfiguracja Γ jest konfiguracją spójną, wtedy i tylko wtedy, gdy Ψ jest odcięciem spójnym.

Odcięcie niespójne – przykład



Predykaty globalne i ich własności

Przez ***predykat globalny $\vartheta(\Sigma)$*** będziemy rozumieć predykat zdefiniowany na zbiorze osiągalnych stanów globalnych przetwarzania rozproszonego.

Predykaty opisują właściwości przetwarzania w poszczególnych stanach. Szczególne znaczenie mają w praktyce ***predykaty stabilne***, których zajście w pewnym stanie globalnym Σ implikuje, że dla każdego stanu Σ' osiągalnego ze stanu Σ, predykat ten jest również prawdziwy. Innymi słowy, predykat jest nazywany stabilnym, gdy spełniany jest następujący warunek:

$$(\vartheta(\Sigma) \wedge (\Sigma \rightsquigarrow \Sigma')) \Rightarrow \vartheta(\Sigma')$$

gdzie $\Sigma \rightsquigarrow \Sigma'$ oznacza, że stan Σ' jest osiągalny ze stanu Σ.

Predykaty, które nie spełniają tego warunku nazwiemy ***predykatami niestabilnymi***.

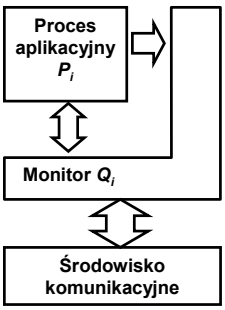
Przykładami predykatów stabilnych są predykaty definiujące stan zakleszczenia, zakończenia przetwarzania, utraty znacznika, przekroczenia czasu obliczeń czy czasu transmisji itp.

Predykaty *possibly* i *definitely*

Predykat possibly(ϑ) zachodzi wtedy i tylko wtedy, gdy istnieje wykonanie $\Upsilon \in \boldsymbol{\Upsilon}$ zawierające stan globalny Σ , dla którego zachodzi predykat $\vartheta(\Sigma)$.

Predykat definitely(ϑ) jest prawdziwy wtedy i tylko wtedy, gdy w każdym możliwym wykonaniu osiągalny jest stan Σ, dla którego zachodzi $\vartheta(\Sigma)$.

Monitory



Ocena wartości predykatów globalnych wymaga w pierwszej kolejności obserwacji (monitorowania) stanów lokalnych procesów składowych. W tym celu przyjmiemy, że z każdym procesem P_i skojarzony jest ***proces monitora*** $\boldsymbol{Q_i}$.

- ❑ Monitor może odczytywać (obserwować) zmienne lokalne procesu.
- ❑ Monitor może obserwować i kontrolować zdarzenia komunikacyjne
- ❑ Monitor nie ma natomiast możliwości zmiany stanu procesu przez przypisanie jego zmiennym lokalnym nowych wartości.

Konwencja zapisu algorytmów (1)

Podstawową strukturą, z której pośrednio lub bezpośrednio wywodzą się struktury wszystkich innych komunikatów jest FRAME. Obejmuje ona wspólne atrybuty komunikatów. Zakłada się przy tym, że struktura komunikatu jest identyfikowana przez jej nazwę, w związku z czym nie musi być ona jawnie wyróżniany jako atrybut. Z punktu widzenia języka, FRAME jest typem danych zdefiniowanym w sposób następujący:

```
type FRAME is record of
    tag: ... /* pole identyfikatora typu */
    mId: ... /* pole identyfikatora wiadomości */
    sId: ... /* pole identyfikatora nadawcy */
    rId: ... /* pole identyfikatora odbiorcy */
end record
```

Struktura komunikatów aplikacyjnych, nazwana MESSAGE, jest wywiedziona ze struktury FRAME, a jej precyzyjna definicja zawarta jest w programie aplikacji rozproszonej i nie jest istotna z punktu widzenia prezentowanych dalej algorytmów. Przyjmiemy zatem tylko, że szkielet definicji struktury MESSAGE ma następującą postać:

```
type MESSAGE extends FRAME is record of
    ...
end record
```


Konwencja zapisu algorytmów (2)

Komunikat kontrolny ma strukturę, zwaną CONTROL, która jest wywiedziona ze struktury FRAME. Precyzyjna definicja struktury CONTROL jest zależna od algorytmu, a jej szkielet jest następujący:

```
type CONTROL extends FRAME is record of
    ...
end record
```

Sygnał jest komunikatem, który nie przenosi żadnych dodatkowych danych. Jego struktura, nazwana SIGNAL, jest zatem „pustym" rozwinięciem struktury FRAME. Ewentualne dalsze rozwinięcia struktury nie wnoszą przy tym żadnego nowego atrybutu, a jedynie zmieniają wartość niejawnego identyfikatora typu – *tag*. Stąd:

```
type SIGNAL extends FRAME is record of
end record
```

Szkielet definicji struktury pakietu, nazwanej PACKET, ma postać przedstawioną poniżej. Właściwa struktura przesyłanych pakietów, zdefiniowana jest w ramach konkretnego algorytmu. Tak więc:

```
type PACKET extends FRAME is record of
    ...
    data: MESSSAGE
end record
```


Czas wirtualny

Zegary realizowane w systemach asynchronicznych mają stanowić aproksymację czasu rzeczywistego. Aproksymacja taka uwzględnia jedynie zachodzące w systemie zdarzenia i dlatego czas ten nazywany jest ***czasem wirtualnym*** (***logicznym***).

W odróżnieniu od czasu rzeczywistego upływ czasu wirtualnego nie jest więc autonomiczny, a zależy od występujących w systemie zdarzeń i stąd określone wartości czasu wirtualnego mogą nigdy nie wystąpić.

Czas wirtualny wyznacza się za pomocą ***zegarów logicznych*** (ang. logical clocks).

Zegary logiczne

Ogólnie mówiąc, ***zegarem logicznym systemu rozproszonego*** nazywamy pewien abstrakcyjny mechanizm, który każdemu zdarzeniu $E \in \Lambda$ przyporządkuje wartość $\mathcal{T}(E)$ (czas wirtualny) z przeciwdziedziny $\mathcal{Y}$.

Formalnie, zegar logiczny jest więc funkcją $\mathcal{T}$: $\Lambda \rightarrow \mathcal{Y}$, odwzorowującą zbiór zdarzeń Λ w zbiór uporządkowany $\mathcal{Y}$, taką że:

$$(E \mapsto E') \Rightarrow (\mathcal{T}(E) < \mathcal{T}(E'))$$

gdzie $<$ jest relacją porządku na zbiorze $\mathcal{Y}$. Należy zauważyć, że w ogólności relacja odwrotna nie musi być spełniona, tzn. $\mathcal{T}(E) < \mathcal{T}(E') \not\Rightarrow E \mapsto E'$.

W konsekwencji definicji, zegar logiczny posiada następujące właściwości:

- Jeżeli zdarzenie E zachodzi przed E' w tym samym procesie, to wówczas wartość zegara logicznego odpowiadającego zdarzeniu E jest mniejsza od wartości zegara odpowiadającego zdarzeniu E' .
- W przypadku przesyłania wiadomości M, czas logiczny przyporządkowany zdarzeniu nadania wiadomości M jest zawsze mniejszy niż czas logiczny przyporządkowany zdarzeniu odbioru tej wiadomości.

Zegary skalarne

Jeżeli przeciwdziedzina $\mathcal{Y}$ funkcji zegara logicznego jest zbiorem liczb naturalnych $\mathbb{N}$ lub rzeczywistych $\mathbb{R}$, to zegar nazywany jest *zegarem skalarnym*.

Realizacja zegarów skalarnych

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

L.Lamport zaproponował realizację skalarnego zegara logicznego, w której funkcja $\mathcal{T}(E)$ implementowana była przez zmienne naturalne $clock_i$, $1 \leqslant i \leqslant n$, skojarzone z procesami P_i (monitorami Q_i).

Wartość zmiennej $clock_i$ reprezentuje w każdej chwili wartość funkcji $\mathcal{T}(E_i^k)$ odnoszącą się do ostatniego zdarzenia E_i^k jakie zaszło w procesie P_i, a tym samym reprezentuje upływ czasu logicznego w tym procesie.

Realizacja zegarów skalarnych (deklaracje, typy komunikatów)

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

TYPY KOMUNIKATÓW

type PACKET **extends** FRAME **is record of**
clock : INTEGER
data : MESSAGE
end record

DEKLARACJE

msgIn : MESSAGE
pcktOut : PACKET
$clock_i$: INTEGER
d : INTEGER

 Slajd 79

Realizacja zegarów skalarnych (akcje)

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

AKCJE

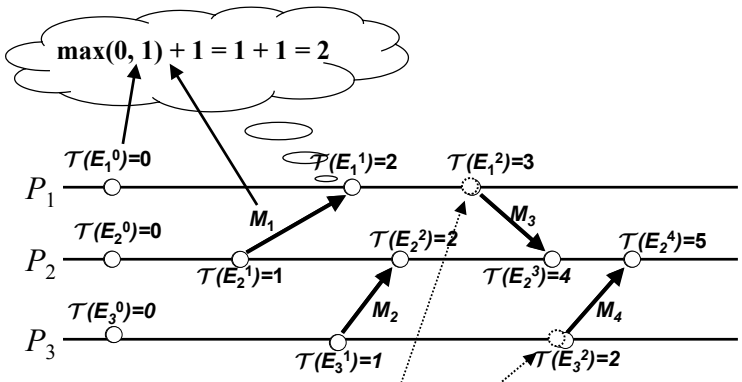
1. **when** *e_send*(P_i, P_j, *msgOut* : MESSAGE) **do**
2. $clock_i := clock_i + d$
3. $pcktOut.clock := clock_i$
4. *pcktOut.data* := *msgOut*
5. **send**(Q_i, Q_j, *pcktOut*)
6. **end when**

7. **when** *e_receive*(Q_j, Q_i, *pcktIn* : PACKET) **do**
8. $clock_i := \max(clock_i, pcktIn.clock) + d$
9. *msgIn* := *pcktIn.data*
10. **deliver**(P_j, P_i, *msgIn*)
11. **end when**

12. **when** *e_internal*(P_i, *) **do**
13. $clock_i := clock_i + d$
14. **end when**

 Slajd 80

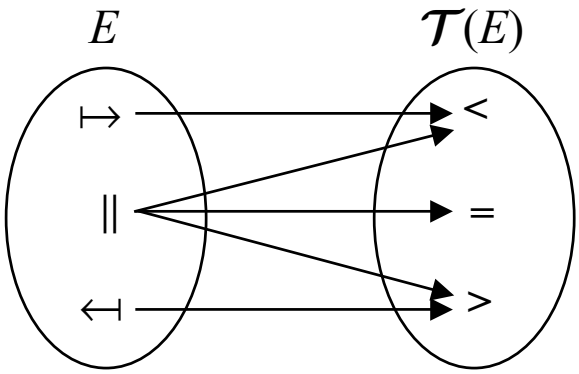
Przykład synchronizacji skalarnych zegarów logicznych



☑ $(E \mapsto E') \Rightarrow (\mathcal{T}(E) < \mathcal{T}(E'))$

☒ $(\mathcal{T}(E) < \mathcal{T}(E')) \Rightarrow (E \mapsto E')$

Relacja między zbiorem zdarzeń i zbiorem wartości zegara skalarnego



Zegary wektorowe (1)

Zegarem wektorowym jest zegar logiczny, dla którego przeciwdziedzina funkcji $\mathcal{T}$, oznaczanej dalej dla odróżnienia przez $\mathcal{T}^V$, jest zbiorem n-elementowych wektorów liczb naturalnych lub rzeczywistych.

Funkcja $\mathcal{T}^V$ implementowana jest przez zmienne tablicowe $vClock_i$, gdzie $1 \leqslant i \leqslant n$, skojarzone jest z poszczególnymi procesami.

Realizacja zegarów wektorowych

Mattern i Fidge (niezależnie) zaproponowali realizację zegarów wektorowych, w której funkcja $\mathcal{T}^V$ została zaimplementowana przez zmienne tablicowe $vClock_i$, $1 \leqslant i \leqslant n$, skojarzone z poszczególnymi procesami. Zmienna $vClock_i$ jest tablicą $[1..n]$ liczb naturalnych, odpowiadającą pewnej aproksymacji czasu globalnego z perspektywy procesu P_i.

W efekcie aktualna wartość tablicy $vClock_i$ odpowiada w każdej chwili wartości funkcji $\mathcal{T}^V(E_i^k)$ odnoszącej się do ostatniego zdarzenia, jakie zaszło w procesie P_i.

Realizacja zegarów wektorowych (typy komunikatów, deklaracje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

DEKLARACJE

msgIn : `MESSAGE`
pcktOut : `PACKET`
$vClock_i$: array [1..*n*] of `INTEGER`
d : `INTEGER`
k : `INTEGER`

TYPY KOMUNIKATÓW

type `PACKET` **extends** `FRAME` **is record of**
vclock : array [1..*n*] of `INTEGER`
data : `MESSAGE`
end record

Realizacja zegarów wektorowych (akcje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

AKCJE

1. **when** *e_send* (P_i, P_j, *msgOut* : `MESSAGE`) **do**
2. $vClock_i[i] := vClock_i[i] + d$
3. $pcktOut.vclock := vClock_i$
4. $pcktOut.data := msgOut$
5. **send**(Q_i, Q_j, *pcktOut*)
6. **end when**
7. **when** *e_receive* (Q_j , Q_i, *pcktIn* : `PACKET`) **do**
8. $vClock_i[i] := vClock_i[i] + d$
9. **for all** $k \in \{1, 2, ..., n\}$ **do**
10. $vClock_i[k] := \max(vClock_i[k], pcktIn.vclock[k])$
11. **end for**
12. $msgIn := pcktIn.data$
13. **deliver**(P_j, P_i, *msgIn*)
14. **end when**
15. **when** *e_internal* (P_i, *) **do**
16. $vClock_i[i] := vClock_i[i] + d$
17. **end when**

Zegary wektorowe (2)

Twierdzenie

W każdej chwili czasu rzeczywistego

$$\forall i,j :: vClock_i[i] \geqslant vClock_j[i]$$

gdzie zmienna $vClock_i[i]$ reprezentuje skalarny czas lokalny procesu P_i, a zmienna $vClock_j[i]$, $j \neq i$, aktualne wyobrażenie procesu P_i o bieżącym skalarnym czasie lokalnym procesu P_j.

 Slajd 87

Zegary wektorowe (3)

Relacje na etykietach wektorowych:

$$vClock_i = vClock_j \quad \Leftrightarrow \quad \forall_k\ vClock_i[k] = vClock_j[k]$$
$$vClock_i \neq vClock_j \quad \Leftrightarrow \quad \exists_k\ vClock_i[k] \neq vClock_j[k]$$

$$vClock_i \leqslant vClock_j \quad \Leftrightarrow \quad \forall_k\ vClock_i[k] \leqslant vClock_j[k]$$
$$vClock_i \not\leqslant vClock_j \quad \Leftrightarrow \quad \exists_k\ vClock_i[k] > vClock_j[k]$$

$$\mathbf{vClock_i < vClock_j} \quad \Leftrightarrow \quad \mathbf{vClock_i \leqslant vClock_j \ \wedge\ vClock_i \neq vClock_j}$$
$$vClock_i \not< vClock_j \quad \Leftrightarrow \quad \neg(vClock_i \leqslant vClock_j \ \wedge\ vClock_i \neq vClock_j)$$

$$vClock_i \parallel vClock_j \quad \Leftrightarrow \quad vClock_i \not< vClock_j \ \wedge\ vClock_j \not< vClock_i)$$

 Slajd 88

Zegary wektorowe (4)

Twierdzenie

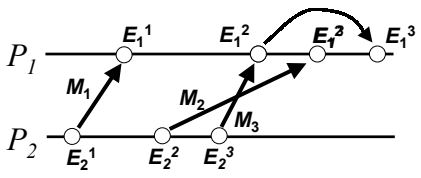
Niech $\mathcal{T}^V(E)$ oraz $\mathcal{T}^V(E')$ będą wartościami zegarów wektorowych zdarzeń E i E'. Wówczas:

$$(E \mapsto E') \Leftrightarrow (\mathcal{T}^V(E) < \mathcal{T}^V(E'))$$

Kanały *FIFO*

W ogólności, każda wiadomość jest przetwarzana (interpretowana) w pewnym kontekście, określonym często przez wiadomości wcześniej odebrane. Brak właściwego kontekstu, spowodowany na przykład zmianą kolejności przesyłanych wiadomości, może prowadzić do niewłaściwej interpretacji wiadomości i w efekcie do błędów przetwarzania.

Kanały gwarantujące porządek odbioru wiadomości zgodny z kolejnością wysyłania będziemy nazywać ***kanałami FIFO*** (ang. First-In-First-Out).



Własności kanałów *FIFO*

Właściwości kanałów *FIFO*:

- ✓ są pewnym mechanizmem synchronizacji wymaganym przez wiele aplikacji,
- ✓ ułatwiają znalezienie rozwiązania i konstrukcję algorytmów rozproszonych dla wielu problemów,
- ✓ ograniczają, w porównaniu z kanałami *nonFIFO*, współbieżność komunikacji, a tym samym efektywność przetwarzania.

Realizacja kanałów *FIFO*

- ➲ Wstęp
- ➲ Typy komunikatów
- ➲ Deklaracje
- ➲ Procedury
- ➲ Akcje

TYPY KOMUNIKATÓW

type PACKET **extends** FRAME **is record of**
seqNo : INTEGER
data : MESSAGE
end record

DEKLARACJE

msgIn : MESSAGE
pcktOut : PACKET
$delayBuf_i$: **set of** PACKET := ∅
$seqNo_i$: **array** [1..*n*] **of** INTEGER := 0
$delivNo_i$: **array** [1..*n*] **of** INTEGER := 0
$delivered_i$: BOOLEAN

AKCJE

```
1.  when e_send(P_i, P_j, msgOut : MESSAGE) do
2.        pcktOut.data := msgOut
3.        seqNo_i[j] := seqNo_i[j] + 1
4.        pcktOut.seqNo := seqNo_i[j]
5.        send( Q_i, Q_j, pcktOut )
6.  end when
7.  when e_receive(Q_j, Q_i, pcktIn : PACKET) do
8.    if pcktIn.seqNo = delivNo_i[j] + 1
9.      then
10.       msgIn := pcktIn.data
11.       deliver( P_j, P_i, msgIn )
12.       delivNo_i[j] := delivNo_i[j] + 1
13.       delivered_i := True
14.     else
15.       delayBuf_i := delayBuf_i ∪ { pcktIn }
16.       delivered_i := False
17.   end if
```

P_k jest nadawcą *pckt*

```
18. while delivered_i do
19.     delivered_i := False
20.     for all pckt ∈ delayBuf_i do
21.       if pckt.seqNo = delivNo_i[k] + 1
22.        then
23.          msgIn := pckt.data
24.          deliver(P_k, P_i, msgIn )
25.          delivNo_i[k] := delivNo_i[k] + 1
26.          delivered_i := True
27.          delayBuf_i := delayBuf_i \{pckt}
28.       end if
29.     end for
30.   end while
31. end when
```

Kanały typu *FC*

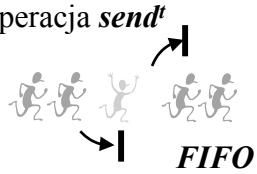
Kanały typu FC (ang. Flush Channels), łączą zalety kanałów *FIFO* i *nonFIFO*, (pewien stopień synchronizacji i współbieżnej komunikacji).

mechanizmy (operacje) komunikacji	zdarzenia	wiadomości
$send^t$ (ang. two-way-flush send)	e_send^t	M^t
$send^f$ (ang. forward-flush send)	e_send^f	M^f
$send^b$ (ang. backward-flush-send)	e_send^b	M^b
$send^o$ (ang. ordinary send)	e_send^o	M^o

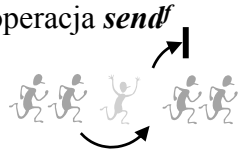
Powiemy, że wiadomość M' ***wyprzedza*** wiadomość M w kanale $C_{i,j}$, jeżeli wiadomość M została wysłana przez P_i wcześniej niż M', lecz proces P_j najpierw odebrał wiadomość M'.

Typy wiadomości w kanałach *FC*

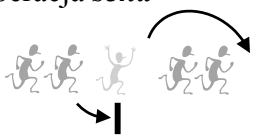
Wiadomość M^t typu ***TF***
(ang. two-way-flush-send)
operacja $send^t$



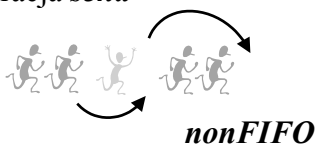
Wiadomość M^f typu ***FF***
(ang. forward-flush-send)
operacja $send^f$



Wiadomość M^b typu ***BF***
(ang. backward-flush-send)
operacja $send^b$



Wiadomość M^o typu ***OF***
(ang. ordinary-send)
operacja $send^o$



Implementacja kanałów *FC*

Kanały typu *FC* mogą być implementowane z użyciem różnych mechanizmów:

- selektywnego rozgłaszania,
- liczników,
- potwierdzeń
- itp.

Relacja poprzedzania $\prec^{+}_{i,j}$

$FL_{i,j}$ – zbiór wiadomości przesłanych kanałem $FC_{i,j}$

$\prec^{+}_{i,j}$ – binarna ***relacja poprzedzania*** *typu F*, zdefiniowana na zbiorze $FL_{i,j}$:
$M \prec^{+}_{i,j} M' \Leftrightarrow (M, M' \in FL_{i,j}) \wedge (M'$ nie może być odebrana przed $M)$

Jeżeli ponadto zachodzi predykat:
$M \prec^{+}_{i,j} M' \wedge (\nexists\, M'' :: (M'' \neq M \wedge M'' \neq M' \wedge M \prec^{+}_{i,j} M'' \wedge M'' \prec^{+}_{i,j} M'))$

to mówimy, że M ***bezpośrednio poprzedza*** M' i fakt ten oznaczamy $M \prec_{i,j} M'$, czyli
$\prec_{i,j} := \{\langle M, M' \rangle : \langle M, M' \rangle \in \prec^{+}_{i,j} \wedge (M$ bezpośrednio poprzedza $M')\}$

Znajomość relacji $\prec_{i,j}$, pozwala na prostą implementację kanałów *FC*, w której wiadomości M' nie są przekazywane procesom aplikacyjnym jeśli nie zostały wcześniej przekazane wszystkie wiadomości M poprzedzające M' w relacji $\prec_{i,j}$.

Konstrukcja relacji $\prec_{i,j}$ (1)

W celu implementacji kanałów *FC* należy rozwiązać problem efektywnego wyznaczenia relacji $\prec_{i,j}$ i przekazywania istotnych jej elementów do monitora odbiorcy.

W tym celu można zaproponować mechanizm sukcesywnej konstrukcji relacji $\prec_{i,j}$ poprzez stosowne uaktualnienie relacji przy wysyłaniu kolejnych wiadomości. W mechanizmie tym, istotne znaczenie ma wiadomość typu *TF* lub *BF* ostatnio wysłana kanałem $FC_{i,j}$, oznaczona przez $M_{i,j}^{\ b}$. Zauważmy, że wszystkie wiadomości wysłane po $M_{i,j}^{\ b}$, powinny być również po niej odebrane.

Początkowo $M_{i,j}^{\ b} = \varnothing$. W tym kontekście, konstrukcja relacji $\prec_{i,j}$ realizowana jest stosownie do typu wysłanej wiadomości, w sposób następujący:

Konstrukcja relacji $\prec_{i,j}$ (2)

- Jeżeli *M* jest typu *OF* i $M_{i,j}^{\ b} \neq \varnothing$, to
  $\prec_{i,j} := \prec_{i,j} \cup \{\langle M_{i,j}^{\ b}, M\rangle\}$

- Jeżeli *M* jest typu *BF* i $M_{i,j}^{\ b} \neq \varnothing$, to
  $\prec_{i,j} := \prec_{i,j} \cup \{\langle M_{i,j}^{\ b}, M\rangle\}$
  Następnie, $M_{i,j}^{\ b} := M$.

- Jeżeli *M* jest typu *FF*, to dla wszystkich *M′*, takich że *M′* nie ma następnika w $\prec_{i,j}$,
  $\prec_{i,j} := \prec_{i,j} \cup \{\langle M', M\rangle\}$

- Jeżeli *M* jest typu *TF*, to dla wszystkich *M′*, takich że *M′* nie ma następnika w $\prec_{i,j}$,
  $\prec_{i,j} := \prec_{i,j} \cup \{\langle M', M\rangle\}$
  Następnie, $M_{i,j}^{\ b} := M$.

Przekazywania monitorowi informacji o relacji $\prec_{i,j}$

W tym celu można zaproponować przesyłanie wiadomości aplikacyjnych w pakietach, zawierających dodatkowo:

- typ wiadomości (*OF, BF, FF*, *TF*),
- numer sekwencyjny wiadomości *M* (*seqNo* = $seqNo_i$[j]),
- numer sekwencyjny *waitForNo* wiadomości bezpośrednio poprzedzającej *M*.

Realizacja kanałów typu *FC*

Realizacja kanałów typu *FC* (deklaracje,typy komunikatów)

TYPY KOMUNIKATÓW

type PACKET **extends** FRAME **is record of**
  type : **enum** {*OF, BF, FF, TF*}
  seqNo : INTEGER
  waitForNo : INTEGER
  data : MESSAGE
end record

DEKLARACJE

msgIn : MESSAGE
pcktOut : PACKET
$delayBuf_i$: **array** [1..*n*] **of set of** PACKET := ∅
$seqNo_i$: **array** [1..*n*] **of** INTEGER := 0
$backFP_i$: **array** [1..*n*] **of** INTEGER := 0
$delivNo_i$: **array** [1..*n*] **of set of** INTEGER := ∅
$delivered_i$: BOOLEAN
k : INTEGER

Realizacja kanałów typu *FC* (procedury)

PROCEDURY

```
1.  procedure TryToDeliver(pcktIn : PACKET) do
2.   msgIn := pcktIn.data
3.   if pcktIn.type = OF ∨ pcktIn.type = BF
4.    then
5.      if pcktIn.waitForNo = 0 ∨ pcktIn.waitForNo ∈ delivNo_i[j]
6.        then
7.          deliver(P_j, P_i, msgIn)
8.          delivNo_i[j] := delivNo_i[j] ∪ {pcktIn.seqNo}
9.      end if
10.    else pcktIn.type = FF ∨ pcktIn.type = TF
11.     if ∀ k:: 1⩽k ⩽pcktIn.waitForNo :: k ∈ delivNo_i[j]
12.       then
13.         deliver(P_j, P_i, msgIn)
14.         delivNo_i[j] := delivNo_i[j] ∪ {pcktIn.seqNo}
15.     end if
16.   end if
17.  end procedure
```

Realizacja kanałów typu *FC* (akcje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje I
➲ Akcje II

AKCJE

```
1.  when e_send^o
      (P_i, P_j, msgOut : MESSAGE)) do
2.    pcktOut.type := OF
3.    seqNo_i[j] := seqNo_i[j] + 1
4.    pcktOut.seqNo := seqNo_i[j]
5.    pcktOut.waitForNo := backFP_i[j]
6.    pcktOut.data := msgOut
7.    send(Q_i, Q_j, pcktOut)
8.  end when

9.  when e_send^b
      (P_i, P_j, msgOut : MESSAGE)) do
10.   pcktOut.type := BF
11.   seqNo_i[j] := seqNo_i[j] + 1
12.   pcktOut.seqNo := seqNo_i[j]
13.   pcktOut.waitForNo := backFP_i[j]
14.   pcktOut.data := msgOut
15.   send(Q_i, Q_j, pcktOut)
16.   backFP_i[j] := seqNo_i[j]
17. end when

18. when e_send^f
      (P_i, P_j, msgOut : MESSAGE)) do
19.   pcktOut.type := FF
20.   seqNo_i[j] := seqNo_i[j] + 1
21.   pcktOut.seqNo := seqNo_i[j]
22.   pcktOut.waitForNo := seqNo_i[j] - 1
23.   pcktOut.data := msgOut
24.   send(Q_i, Q_j, pcktOut)
25. end when
26. when e_send^t
      (P_i, P_j, msgOut : MESSAGE)) do
27.   pcktOut.type := TF
28.   seqNo_i[j] := seqNo_i[j] + 1
29.   pcktOut.seqNo := seqNo_i[j]
30.   pcktOut.waitForNo := seqNo_i[j] - 1
31.   pcktOut.data := msgOut
32.   send(Q_i, Q_j, pcktOut)
33.   backFP_i[j] := seqNo_i[j]
34. end when
```


Realizacja kanałów typu *FC* (akcje II)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje I
➲ Akcje II

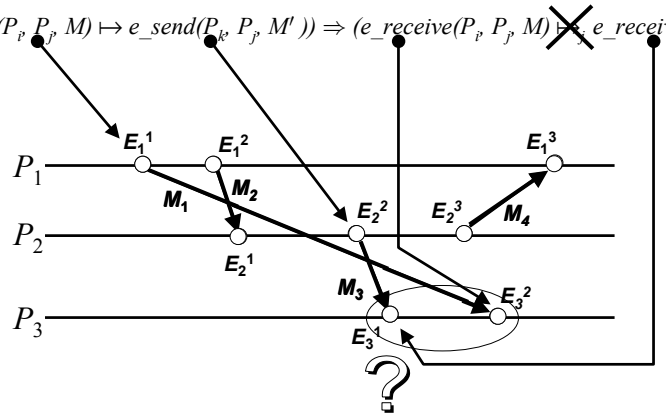
AKCJE

```
35.  when e_receive( Q_j, Q_i, pcktIn : PACKET ) do
36.    TryToDeliver(pcktIn)
37.    if pcktIn.seqNo ∈ delivNo_i[j]
38.     then
39.       delivered_i := True
40.     else
41.       delivered_i := False
42.       delayBuf_i := delayBuf_i ∪ {pcktIn}
43.    end if
44.    while delivered_i do
45.      delivered_i := False
46.      for all pckt ∈ delayBuf_i[j] do
47.         TryToDeliver(pckt)
48.         if pckt.seqNo ∈ delivNo_i[j]
49.                then
50.                  delivered_i := True
51.                  delayBuf_i := delayBuf_i \ {pckt}
52.          end if
53.      end for
54.    end while
55.  end when
```

Środowisko komunikacyjne zachowujące uporządkowanie przyczynowe wiadomości (1)

Środowisko komunikacyjne zachowujące uporządkowanie przyczynowe wiadomości zapewnia dla wszystkich procesów i wiadomości, że:

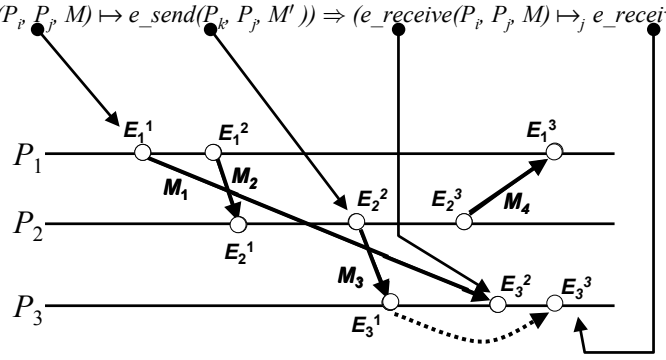
$(e\_send(P_i, P_j, M) \mapsto e\_send(P_k, P_j, M')) \Rightarrow (e\_receive(P_i, P_j, M) \mapsto_j e\_receive(P_k, P_j, M'))$



Środowisko komunikacyjne zachowujące uporządkowanie przyczynowe wiadomości (2)

Środowisko komunikacyjne zachowujące uporządkowanie przyczynowe wiadomości zapewnia dla wszystkich procesów i wiadomości, że:

$(e\_send(P_i, P_j, M) \mapsto e\_send(P_k, P_j, M')) \Rightarrow (e\_receive(P_i, P_j, M) \mapsto_j e\_receive(P_k, P_j, M'))$



Realizacja środowiska zachowującego uporządkowanie przyczynowe wiadomości z rozgłaszaniem

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

Birman, Schiper i Stephenson zaproponowali algorytm gwarantujący zachowanie uporządkowania przyczynowego wiadomości przy wykorzystaniu mechanizmu rozgłaszania.

- Wstęp
- Typy komunikatów
- Deklaracje
- Procedury
- Akcje

TYPY KOMUNIKATÓW

type PACKET **extends** FRAME **is record of**
vSentClock : **array** [1..*n*] **of** INTEGER
data : MESSAGE
end record

DEKLARACJE

msgIn : MESSAGE
pcktOut : PACKET
$vSentClock_i$: **array** [1..*n*] **of** INTEGER
k : INTEGER

Realizacja środowiska zachowującego uporządkowanie przyczynowe wiadomości z rozgłaszaniem (akcje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

AKCJE

```
1.  when e_send(P_i, P_j, msgOut : MESSAGE) do
2.      vSentClock_i[i] := vSentClock_i[i] + 1
3.      pcktOut.vSentClock := vSentClock_i
4.      pcktOut.data := msgOut
5.      send(Q_i, 𝒬 \ {Q_i}, pcktOut)
6.  end when

7.  when e_receive(Q_j, Q_i, pcktIn : PACKET) do
8.      wait until (vSentClock_i[j] = pcktIn.vSentClock[j] - 1) ∧
9.          (∀ k ∈ {1, 2, ..., n} \ {j} :: vSentClock_i[k] ⩾ pcktIn.vSentClock[k])
10.     if pckIn.data.rId = P_i
11.       then
12.         msgIn := pcktIn.data
13.         deliver(P_j, P_i, msgIn)
14.     end if
15.     for all k ∈ {1, 2, ..., n} do
16.         vSentClock_i[k] := max(vSentClock_i[k], pcktIn.vSentClock[k])
17.     end for
18. end when
```


Realizacja środowiska zachowującego uporządkowanie przyczynowe wiadomości bez mechanizmu rozgłaszania

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

Środowisko komunikacyjne zachowujące uporządkowanie przyczynowe wiadomości bez konieczności rozgłaszania wszystkich pakietów realizuje algorytm Schipera – Egli – Sandoza.

Realizacja środowiska zachowującego uporządkowanie przyczynowe wiadomości bez mechanizmu rozgłaszania (typy komunikatów)

TYPY KOMUNIKATÓW

```
type PACKET extends FRAME is record of
  vSentClock : array [1..n] of INTEGER
  vM         : set of record of
                 pId : PROCESS_ID
                 vTS : array [1..n] of INTEGER
               end record
  data : MESSAGE
end record
```


Realizacja środowiska zachowującego uporządkowanie przyczynowe wiadomości bez mechanizmu rozgłaszania (deklaracje)

DEKLARACJE

```
msgIn          : MESSAGE
pcktOut        : PACKET
vSentClock_i   : array [1..n] of INTEGER
vP_i           : set of record
                   pId : PROCESS_ID
                   vTS : array [1..n] of INTEGER
                 end record
delivered_i    : BOOLEAN
delayBuf_i     : set of PACKET := ∅
tmpBuf_i       : set of PACKET
vTS_i^sup      : array [1..n] of INTEGER
```

Realizacja środowiska zachowującego uporządkowanie przyczynowe wiadomości bez mechanizmu rozgłaszania (procedury)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

PROCEDURY

```
1.  procedure TryToDeliver
                (pcktIn : PACKET) do
2.    msgIn := pcktIn.data
3.    if ∀ ⟨ pId^M, vTS^M ⟩∈ pcktIn.vM ::
          pId^M ≠ P_i
4.      then
5.          deliver(P_j, P_i, msgIn)
6.          delivered_i := True
7.    end if
8.    if  ∃ ⟨ pId^M, vTS^M ⟩∈ pcktIn.vM ::
9.        pId^M = P_i ∧ vTS^M ≮ vSentClock_i
10.     then
11.       delayBuf_i := delayBuf_i ∪ {pcktIn}
12.     else
13.         deliver(P_j, P_i, msgIn)
14.         delivered_i := True
15.   end if
16. if delivered_i
17.  then
18.  for all ⟨pId^M,vTS^M⟩∈ pcktIn.vM∧ pId^M≠P_i do
19.   if ∀ ⟨pId, vTS⟩∈ vP_i :: pId ≠ pId^M
20.    then
21.     vP_i := vP_i ∪ {⟨ pId^M, vTS^M ⟩}
22.    else
23.     for ( pId, vTS) ∈ vP_i ∧ pId = pId^M do
24.       for all k ∈ {1, 2, ..., n} do
25.         vTS_i^sup[k] := max(vTS[k], vTS^M[k])
26.       end for
27.       vP_i := vP_i \ {⟨pId, vTS⟩}
28.       vP_i := vP_i ∪ {⟨pId, vTS_i^sup⟩}
29.     end for
30.    end if
31.  end for
32.  for all k ∈ {1, 2, ..., n} do
33.    vSentClock_i[k] :=
        max( vSentClock_i[k], pcktIn.vSentClock[k])
34.  end for
35. end if
36. end procedure
```


Realizacja środowiska zachowującego uporządkowanie przyczynowe wiadomości bez mechanizmu rozgłaszania (akcje)

➲ Wstęp
➲ Typy komunikatów
➲ Deklaracje
➲ Procedury
➲ Akcje

AKCJE

```
1.  when e_send
        (P_i, P_j, msgOut : MESSAGE) do
2.    vSentClock_i[i] := vSentClock_i[i] + 1
3.    pcktOut.vSentClock := vSentClock_i
4.    pcktOut.vM := vP_i
5.    pcktOut.data := msgOut
6.    send(Q_i, Q_j, pcktOut)
7.    for ⟨pId, vTS⟩ ∈ vP_i ∧ pId = P_j do
8.      vP_i := vP_i \ {⟨pId, vTS⟩}
9.    end for
10.   vP_i := vP_i ∪ {(P_j, vSentClock_i)}
11. end when

12. when e_receive
        (Q_j, Q_i, pcktIn : PACKET) do
13.   delivered_i := False
14.   TryToDeliver(pcktIn)
15.   tmpBuf_i := ∅
16.   while delivered_i do
17.     tmpBuf_i := tmpBuf_i ∪ delayBuf_i
18.     delayBuf_i := ∅
19.     delivered_i := False
20.     for all pckt ∈ tmpBuf_i do
21.       tmpBuf_i := tmpBuf_i \ {pckt}
22.       TryToDeliver(pckt)
23.     end for
24.   end while
25. end when
```

Funkcje kosztu wykonania algorytmów (1)

Δ^{δ}_{A} – zbiór wszystkich poprawnych danych wejściowych δ algorytmu A,

$\mathcal{Z}^{*}_{\mathrm{A}}(\delta)$ – ***koszt wykonywania*** algorytmu A dla danych δ, gdzie $\delta \in \Delta^{\delta}_{A}$ i $\mathcal{Z}^{*}_{\mathrm{A}} : \Delta^{\delta}_{A} \rightarrow \mathbb{R}$

μ – rozmiar danych wejściowych δ (rozmiar zadania), taki że $\mu = \mathcal{W}(\delta)$, gdzie $\mathcal{W} : \Delta^{\delta}_{A} \rightarrow \mathbb{N}$, jest zadaną funkcją.

W praktyce, zamiast kosztu $\mathcal{Z}^{*}_{\mathrm{A}}(\delta)$ stosuje się zwykle jego oszacowanie w funkcji rozmiaru zadania $\mu = \mathcal{W}(\delta)$.

Funkcją kosztu wykonania algorytmu nazywać będziemy odwzorowanie:

$$\mathcal{Z}_{\mathrm{A}} : \Delta^{\mu}_{A} \rightarrow \mathbb{R}$$

Funkcje kosztu wykonania algorytmów (2)

Najczęściej stosowane jest odwzorowanie tak zwane ***pesymistyczne*** (najgorszego przypadku), zdefiniowane w sposób następujący:

$$\mathcal{Z}_{\mathrm{A}}(\mu) = \sup\{\mathcal{Z}^{*}_{A}(\delta) : \delta \in \Delta^{\delta}_{A} \wedge \mathcal{W}(\delta) = \mu\}$$

Funkcja ta odwzorowuje zbiór rozmiarów danych wejściowych algorytmu (podzbiór zbioru liczb naturalnych) w zbiór liczb rzeczywistych. Definicja każe interpretować tę funkcję jako miarę złożoności algorytmu dla przypadku najbardziej niekorzystnego, stąd określenie pesymistyczna funkcja kosztu.

Rząd funkcji

Niech f i g będą dowolnymi funkcjami odwzorowującymi $\mathbb{N}$ w $\mathbb{R}$.

Mówimy, że ***funkcja f jest co najwyżej rzędu funkcji g***, co zapisujemy:

$$f = O(g)$$

jeżeli istnieje stała rzeczywista $c > 0$ oraz $n_0 \in \mathbb{N}$ takie, że dla każdej wartości $n > n_0$, $n \in \mathbb{N}$ zachodzi:

$$|f(n)| < c\,|g(n)|$$

Mówimy, że ***funkcja f jest dokładnie rzędu g***, co zapisujemy:

$$f = \Theta(g)$$

jeżeli $f = O(g) \wedge g = O(f)$.

Mówimy, że ***funkcja f jest co najmniej rzędu funkcji g***, co zapisujemy:

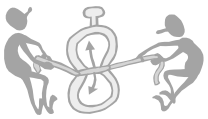
$$f = \Omega(g)$$

jeżeli $g = O(f)$.

Złożoność czasowa algorytmów rozproszonych

W wypadku algorytmów rozproszonych, ***złożoność czasowa*** jest funkcją kosztu wykonania, wyrażoną przez liczbę kroków algorytmu do jego zakończenia przy założeniu, że:

- czas wykonywania każdego kroku (operacji, zdarzenia) jest stały
- kroki wykonywane są synchronicznie
- czas transmisji wiadomości jest stały



W analizie złożoności czasowej algorytmów rozproszonych przyjmuje się też na ogół, że czas przetwarzania lokalnego (wykonania każdego kroku) jest pomijalny (zerowy), a czas transmisji jest jednostkowy. Będziemy stosować taką postać definicji.

Złożoność komunikacyjna algorytmów rozproszonych

Złożoność komunikacyjna jest funkcją kosztu wykonania algorytmu wyrażaną przez:

- liczbę pakietów (wiadomości) przesyłanych w trakcie wykonywania algorytmu do jego zakończenia
- sumaryczną długość (w bitach) wszystkich wiadomości przesłanych w trakcie wykonywania algorytmu.

W konsekwencji wyróżniamy zatem tak zwaną złożoność ***pakietową*** i ***bitową***.

Warunki poprawności algorytmów rozproszonych

Analizę poprawności algorytmu rozproszonego (procesu rozproszonego) dekomponuje się zwykle na analizę jego bezpieczeństwa i żywotności.

- Właściwość ***bezpieczeństwa*** (ang. safety, consistency)
  algorytm rozproszony nazywamy *bezpiecznym*, jeśli nigdy nie dopuszcza do niepożądanego stanu lub inaczej, gdy zawsze utrzymuje proces rozproszony w stanie pożądanym.
- Właściwość ***żywotności – postępu*** (ang. liveness, progress)
  algorytm rozproszony nazywamy *żywotnym* jeśli zapewnia, że każde pożądane zdarzenie w końcu zajdzie.

