Czas wirtualny, złożoność algorytmów

Wykład prowadzą:

Jerzy Brzeziński
Jacek Kobusiński

UCZELNIA ONLINE

Plan wykładu

- Zegary logiczne
  - skalarne
  - wektorowe
- Kanały
  - FIFO
  - typu FC
- Rząd funkcji, funkcja kosztu
- Złożoność komunikacyjna i czasowa algorytmu
- Warunki poprawności algorytmu

Czas wirtualny (1)

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**).

Czas wirtualny (2)

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*).

Zegar logiczny - definicja

Zegar logiczny systemu rozproszonego jest funkcją $\mathcal{T}$: $\Lambda \to \mathcal{Y}$, odwzorowującą zbiór zdarzeń Λ w zbiór uporządkowany $\mathcal{Y}$, taką że:

$$(E \mapsto E') \Rightarrow (\mathcal{T}(E) < \mathcal{T}(E')) \qquad (4.1)$$

gdzie $<$ jest relacją porządku na zbiorze $\mathcal{Y}$.

Należy zauważyć, że w ogólności relacja odwrotna nie musi być spełniona, tzn. $\mathcal{T}(E) < \mathcal{T}(E') \nRightarrow E \mapsto E'$.

Zegary logiczne - 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

Zegar skalarny – definicja

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

Funkcja $\mathcal{T}(E)$ implementowana jest 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.

Alg. Lamporta (1)

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


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


Alg. Lamporta (2)

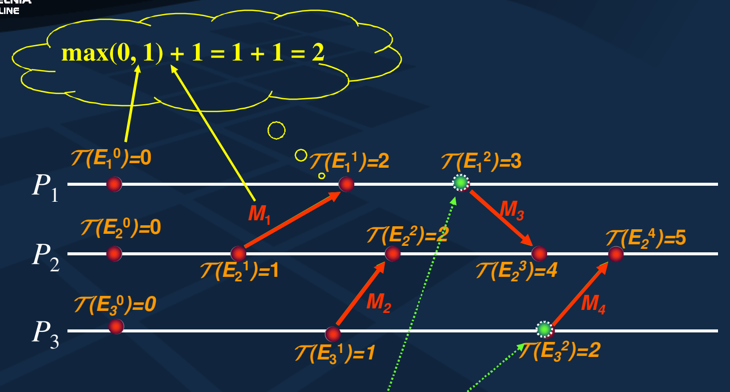
```
when e_send(P_i, P_j, msgOut: MESSAGE) do
  clock_i := clock_i + d
  pcktOut.clock := clock_i
  pcktOut.data := msgOut
  send(Q_i, Q_j, pcktOut)
end when
```


Alg. Lamporta (3)

```
when e_receive(Q_j, Q_i, pcktIn: PACKET) do
  clock_i := max(clock_i, pcktIn.clock) + d
  msgIn := pcktIn.data
  deliver(P_j, P_i, msgIn)
end when

when e_internal(P_i, *) do
   clock_i := clock_i + d
end when
```

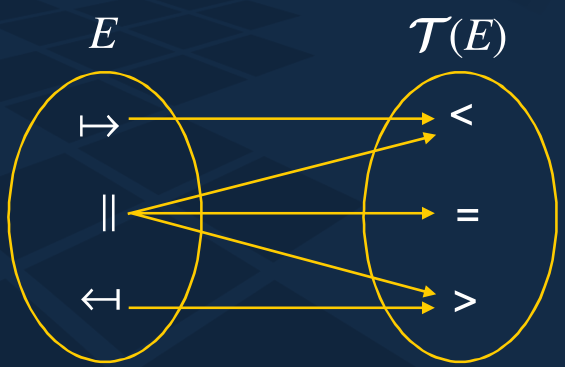

Przykład synchronizacji zegarów logicznych



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

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

Relacja między zb. zdarzeń i zb. wartości zegara skalarnego



Zegar wektorowy - definicja

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

Realizacja zegarów wektorowych

Funkcja $\mathcal{T}^V$ implementowana jest 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.

Alg. Matterna (1)

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

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

Alg. Matterna (2)

```
when e_send(P_i, P_j, msgOut: MESSAGE) do
  vClock_i[i] := vClock_i[i] + d
  pcktOut.vClock := vClock_i
  pcktOut.data := msgOut
  send(Q_i, Q_j, pcktOut)
end when
```


Alg. Matterna (3)

```
when e_receive(Q_j , Q_i, pcktIn: PACKET) do
 vClock_i[i] := vClock_i[i] + d
 for all k ∈ {1, 2, ..., n} do
   vClock_i[k] := max(vClock_i[k], pcktIn.vClock[k])
 end for
 msgIn := pcktIn.data
 deliver(P_j, P_i, msgIn)
end when

when e_internal(P_i, *) do
 vClock_i[i] := vClock_i[i] + d
end when
```


Zegary wektorowe (1)

Twierdzenie 4.1

W każdej chwili czasu rzeczywistego

$$\forall i,j :: vClock_i[i] \geqslant vClock_j[i] \tag{4.2}$$

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

Relacje na etykietach wektorowych

$$vClock_i = vClock_j \iff \forall_k\ vClock_i[k] = vClock_j[k]$$
$$vClock_i \neq vClock_j \iff \exists_k\ vClock_i[k] \neq vClock_j[k]$$

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

$$vClock_i < vClock_j \iff vClock_i \leqslant vClock_j \wedge vClock_i \neq vClock_j$$
$$vClock_i \not< vClock_j \iff \neg(vClock_i \leqslant vClock_j \wedge vClock_i \neq vClock_j)$$

$$vClock_i \parallel vClock_j \iff vClock_i \not< vClock_j \wedge vClock_j \not< vClock_i)$$

Zegary wektorowe (2)

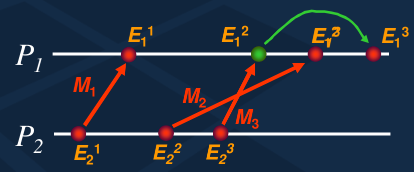
Twierdzenie 4.2

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

$$(E \mapsto E') \Leftrightarrow (\mathcal{T}^V(E) < \mathcal{T}^V(E')) \qquad (4.3)$$

Kanały FIFO

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



Alg. Müllender'a (1)

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

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


Alg. Müllender'a (2)

```
when e_send(P_i, P_j, msgOut: MESSAGE) do
  pcktOut.data:= msgOut
  seqNo_i[j]:= seqNo_i[j] + 1
  pcktOut.seqNo:= seqNo_i[j]
  send(Q_i, Q_j, pcktOut)
end when
```

Alg. Müllender'a (3)

```
when e_receive(Q_j, Q_i, pcktIn : PACKET)  do
  if pcktIn.seqNo = delivNo_i[j] + 1
   then
      msgIn:= pcktIn.data
      deliver(P_j, P_i, msgIn)
      delivNo_i[j]:= delivNo_i[j] + 1
      delivered_i:= True
    else
      delayBuf_i[j]:= delayBuf_i[j] ∪ {pcktIn}
      delivered_i:= False
    end if
```


Alg. Müllender'a (4)

```
  while delivered_i do
    delivered_i:= False
    for all pckt ∈ delayBuf_i[j] do
      if pckt.seqNo = delivNo_i[j] + 1 then
        msgIn:= pckt.data
        deliver(P_j, P_i, msgIn)
        delivNo_i[j]:= delivNo_i[j] + 1
        delivered_i:= True
        delayBuf_i[j]:= delayBuf_i[j]\{pckt}
      end if
    end for
  end while
end when
```


Cechy 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.

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

Wyprzedzanie wiadomości

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 $send^t$

FIFO

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 $send^o$

nonFIFO

Implementacja kanałów *FC*

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

- selektywnego rozgłaszania
- liczników
- potwierdzeń
- …

Relacja binarna poprzedzania

$FL_{i,j}$ – stan kanału $FC_{i,j}$

$\prec^+_{i,j}$ – binarna **relacja poprzedzania** typu F, zdefiniowana na zbiorze $FL_{i,j}$:

$$M \prec^+_{i,j} M' \Leftrightarrow (M, M' \in FL_{i,j}) \wedge (M' \text{ nie może być odebrana przed } M) \qquad (4.4)$$

Bezpośrednie poprzedzanie

Jeżeli zachodzi predykat :

$$M \prec^{+}_{i,j} M' \wedge (\nexists M'' :: (M'' \neq M \wedge M'' \neq M' \wedge M \prec^{+}_{i,j} M'' \wedge M'' \prec^{+}_{i,j} M')) \qquad (4.5)$$

to mówimy, że M **bezpośrednio poprzedza** M' i fakt ten oznaczamy $M \prec_{i,j} M'$, czyli

$\prec_{i,j} := \{\langle M , M' \rangle : (M$ bezpośrednio poprzedza $M')\}$

Konstrukcja relacji poprzedzania (1)

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

Mechanizm sukcesywnej konstrukcji relacji $\prec_{i,j}$ poprzez stosowne uaktualnienie relacji przy wysyłaniu kolejnych wiadomości.

Konstrukcja relacji poprzedzania (2)

Niech $M_{i,j}{}^{b}$ oznacza ostatnią wiadomość typu *TF* lub *BF* wysłaną kanałem $FC_{i,j}$

- Jeżeli M jest typu OF i $M_{i,j}{}^{b} \neq \varnothing$, to

  $\prec_{i,j} \; := \; \prec_{i,j} \cup \{\langle M_{i,j}{}^{b}, M \rangle\}$ (4.6)

- Jeżeli M jest typu BF i $M_{i,j}{}^{b} \neq \varnothing$, to

  $\prec_{i,j} \; := \; \prec_{i,j} \cup \{\langle M_{i,j}{}^{b}, M \rangle\}$ (4.7)

  Następnie, $M_{i,j}{}^{b} := M$.

Konstrukcja relacji poprzedzania (3)

- Jeżeli M jest typu FF, to dla wszystkich M', takich że M' nie ma następnika w $\prec_{i,j}$,

  $\prec_{i,j} \; := \; \prec_{i,j} \cup \{\langle M', M \rangle\}$ (4.8)

- Jeżeli M jest typu TF, to dla wszystkich M', takich że M' nie ma następnika w $\prec_{i,j}$,

  $\prec_{i,j} \; := \; \prec_{i,j} \cup \{\langle M', M \rangle\}$ (4.9)

  Następnie, $M_{i,j}{}^{b} := M$.

Przekazywania informacji o relacji poprzedzania

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

Alg. Kearnsa, Campa i Ahuja (1)

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

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
```


Alg. Kearnsa, Campa i Ahuja (2)

```
procedure TryToDeliver(pcktIn: PACKET) do
  msgIn := pcktIn.data
  if pcktIn.type = OF ∨ pcktIn.type = BF
   then
    if pcktIn.waitForNo=0 ∨ pcktIn.waitForNo∈delivNo_i[j]
      then
      deliver(P_j, P_i, msgIn)
      delivNo_i[j]:= delivNo_i[j] ∪ {pcktIn.seqNo}
    end if
    else /* pcktIn.type = FF ∨ pcktIn.type = TF */
    if  ∀ k :: 1≤k ≤pcktIn.waitForNo :: k ∈ delivNo_i[j]
      then
      deliver(P_j, P_i, msgIn)
      delivNo_i[j]:= delivNo_i[j] ∪ {pcktIn.seqNo}
    end if
  end if
 end procedure
```


Alg. Kearnsa, Campa i Ahuja (3)

```
when e_send°(P_i, P_j, msgOut: MESSAGE) do
   pcktOut.type:= OF
   seqNo_i[j]:= seqNo_i[j] + 1
   pcktOut.seqNo:= seqNo_i[j]
   pcktOut.waitForNo:= backFP_i[j]
   pcktOut.data:= msgOut
   send(Q_i, Q_j, pcktOut)
end when
```

Alg. Kearnsa, Campa i Ahuja (4)

```
when e_send^f(P_i, P_j, msgOut: MESSAGE) do
  pcktOut.type:= FF
  seqNo_i[j]:= seqNo_i[j] + 1
  pcktOut.seqNo:= seqNo_i[j]
  pcktOut.waitForNo:= seqNo_i[j] - 1
  pcktOut.data:= msgOut
  send(Q_i, Q_j, pcktOut)
end when
```


Alg. Kearnsa, Campa i Ahuja (5)

```
when e_send^b(P_i, P_j, msgOut: MESSAGE) do
    pcktOut.type:= BF
    seqNo_i[j]:= seqNo_i[j] + 1
    pcktOut.seqNo:= seqNo_i[j]
    pcktOut.waitForNo:= backFP_i[j]
    pcktOut.data:= msgOut
    send(Q_i, Q_j, pcktOut)
    backFP_i[j]:= seqNo_i[j]
 end when
```


Alg. Kearnsa, Campa i Ahuja (6)

```
when e_send^t(P_i, P_j, msgOut: MESSAGE) do
  pcktOut.type:= TF
  seqNo_i[j]:= seqNo_i[j] + 1
  pcktOut.seqNo:= seqNo_i[j]
  pcktOut.waitForNo:= seqNo_i[j] - 1
  pcktOut.data:= msgOut
  send(Q_i, Q_j, pcktOut)
  backFP_i[j]:= seqNo_i[j]
end when
```


Alg. Kearnsa, Campa i Ahuja (7)

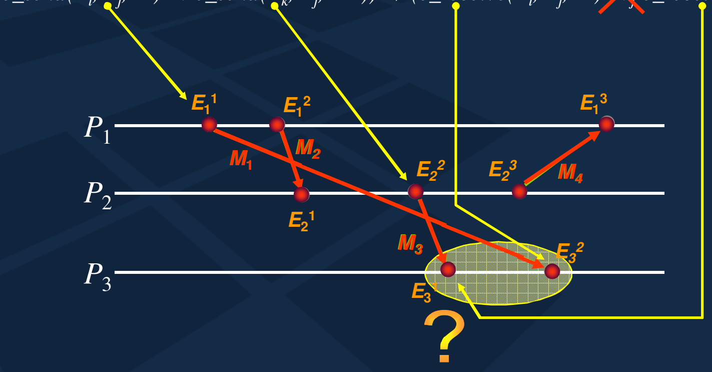
```
when e_receive(Q_j, Q_i, pcktIn: PACKET)  do
   TryToDeliver(pcktIn)
   if pcktIn.seqNo ∈ delivNo_i[j]
   then
        delivered_i:= True
   else
        delivered_i:= False
        delayBuf_i:= delayBuf_i ∪ {pcktIn}
   end if
...
```

Alg. Kearnsa, Campa i Ahuja (8)

```
while delivered_i do
  delivered_i := False
  for all pckt ∈ delayBuf_i[j] do
          TryToDeliver(pckt)
           if pckt.seqNo ∈ delivNo_i[j]
             then
               delivered_i := True
               delayBuf_i := delayBuf_i \ {pckt}
             end if
        end for
    end while
end when
```

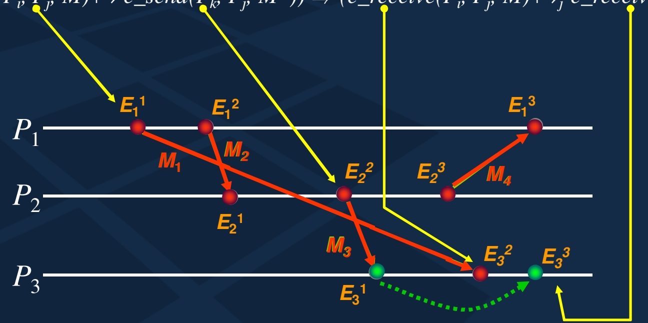

Środowisko zachowujące uporządkowanie przyczynowe

$$(e\_send(P_i, P_j, M) \mapsto e\_send(P_k, P_j, M')) \Rightarrow (e\_receive(P_i, P_j, M) \cancel{\mapsto_j} e\_receive(P_k, P_j, M')) \quad (4.10)$$



Środowisko zachowujące uporządkowanie przyczynowe

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



Alg. Birmana, Schipera i Stephensona (1)

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


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

Alg. Birmana, Schipera i Stephensona (2)

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


Alg. Birmana, Schipera i Stephensona (3)

```
when e_receive(Q_j, Q_i, pcktIn: PACKET) do
 wait until (vSentClock_i[j]=pcktIn.vSentClock[j]-1) ∧
  ∀k∈{1,2,...,n}\{j}::vSentClock_i[k]⩾pcktIn.vSentClock[k])
 if pckIn.data.rId = P_i
   then
   msgIn:= pcktIn.data
   deliver(P_j, P_i, msgIn)
 end if
 for all k ∈ {1, 2, ..., n} do
  vSentClock_i[k]:=max(vSentClock_i[k],pcktIn.vSentClock[k])
 end for
end when
```


Alg. Schipera, Egli, Sandoza (1)

```
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
```


Alg. Schipera, Egli, Sandoza (2)

```
msgIn          : MESSAGE
pcktOut        : PACKET
vSentClock_i   : array [1..n] of INTEGER
vP_i           : set of record of
                    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
```

Alg. Schipera, Egli, Sandoza (3)

```
procedure TryToDeliver(pcktIn: PACKET) do
  msgIn:= pcktIn.data
  if ∃⟨pId^M, vTS^M⟩ ∈ pcktIn.vM :: pId^M = P_i ∧ vTS^M ≰ vSentClock_i
     then
        delayBuf_i := delayBuf_i ∪ {pcktIn}
     else
         deliver(P_j, P_i, msgIn)
         delivered_i := True
  end if
```


Alg. Schipera, Egli, Sandoza (4)

```
if delivered_i then
 for all ⟨pId^M, vTS^M⟩ ∈ pcktIn.vM ∧ pId^M ≠ P_i do
  if ∀ ⟨pId, vTS⟩ ∈ vP_i :: pId ≠ pId^M then
      vP_i:= vP_i ∪ {⟨ pId^M, vTS^M ⟩}
     else
      for ( pId, vTS) ∈ vP_i ∧ pId = pId^M do
         for all k ∈ {1, 2, ..., n} do
            vTS_i^sup[k] := max(vTS[k], vTS^M[k])
         end for
         vP_i := vP_i \ {⟨pId, vTS⟩}
         vP_i := vP_i ∪ {⟨pId, vTS_i^sup⟩}
      end for
    end if
  end for
  for all k ∈ {1, 2, ..., n} do
   vSentClock_i[k] :=
                   max(vSentClock_i[k], pcktIn.vSentClock[k])
  end for
end if
end procedure
```


Alg. Schipera, Egli, Sandoza (5)

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


Alg. Schipera, Egli, Sandoza (6)

```
when e_receive(Q_j, Q_i, pcktIn: PACKET) do
  delivered_i := False
  TryToDeliver(pcktIn)
  tmpBuf_i := ∅
  while delivered_i do
    tmpBuf_i := tmpBuf_i ∪ delayBuf_i
    delayBuf_i := ∅
    delivered_i := False
    for all pckt ∈ tmpBuf_i do
      tmpBuf_i := tmpBuf_i \ {pckt}
      TryToDeliver(pckt)
    end for
  end while
end when
```

Funkcje kosztu – oznaczenia

Δ^{δ}_A zbiór wszystkich poprawnych danych wejściowych δ algorytmu *A*,

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

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 $\mathcal{Z}^*_A(\delta)$ stosuje się zwykle jego oszacowanie w funkcji rozmiaru zadania $\mu = \mathcal{W}(\delta)$.

Funkcja kosztu - definicja

Funkcją kosztu wykonania algorytmu nazywać będziemy odwzorowanie

$$\mathcal{Z}_A : \Delta^{\mu}_A \rightarrow \mathbb{R} \qquad (4.11)$$

Gdzie Δ^{μ}_A jest zbiorem wszystkich poprawnych danych wejściowych o rozmiarze μ algorytmu A.

Funkcje kosztu wykonania algorytmów

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

$$\mathcal{Z}_A(\mu) = \sup\{ \mathcal{Z}^*_A(\delta) : \delta \in \Delta^{\delta}_A \wedge \mathcal{W}(\delta) = \mu \} \qquad (4.12)$$

Rząd funkcji (1)

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:

$$f = O(g) \qquad (4.13)$$

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:

$$| f(n) | < c \, | g(n) | \qquad (4.14)$$

Rząd funkcji (2)

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

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

$$f = \Theta(g) \tag{4.15}$$

jeżeli $f = O(g) \wedge g = O(f)$.

Rząd funkcji (3)

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

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

$$f = \Omega(g) \tag{4.16}$$

jeżeli $g = O(f)$.

Złożoność czasowa

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) jest stały
- kroki wykonywane są synchronicznie
- czas transmisji wiadomości jest stały

Czasy przetwarzania lokalnego i transmisji

W analizie złożoności czasowej algorytmów rozproszonych przyjmuje się też na ogół, że

- czas przetwarzania lokalnego (wykonania każdego kroku) jest pomijalny (zerowy)
- czas transmisji jest jednostkowy

Złożoność komunikacyjna

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 złożoność **pakietową** i **bitową**.

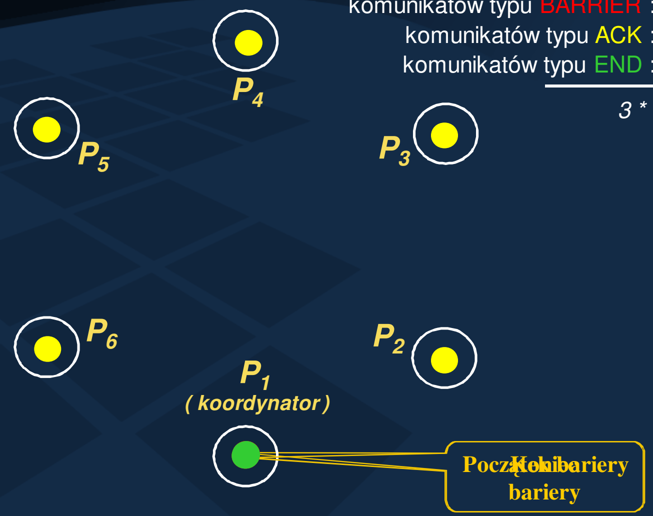
Przykład (1) - bariera

- Topologia połączeń – graf w pełni połączony
- Koordynator rozgłasza komunikat początku bariery
- Wszyscy uczestnicy odbierają komunikat i odsyłają potwierdzenia
- Po otrzymaniu potwierdzeń od wszystkich procesów, koordynator rozsyła komunikat końca bariery

Przykład (1) - rysunek

komunikatów typu BARRIER : n-1
komunikatów typu ACK : n-1
komunikatów typu END : n-1

$3 * (n$-1)



Przykład (2) - bariera

- Topologia połączeń – pierścień logiczny
- Koordynator wysyła komunikat początku bariery
- Wszyscy uczestnicy odbierają komunikat i przesyłają go dalej
- Po otrzymaniu komunikatu rozpoczynającego operację bariery, koordynator przesyła komunikat końca bariery

Przykład (2) - rysunek

komunikatów typu BARRIER : *n*
komunikatów typu END : *n*

$2 * n$

Warunki poprawności

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

- właściwość **bezpieczeństwa** (ang. *safety*, *consistency*)
- właściwość **żywotności (postępu)** (ang. *liveness*, *progress*)