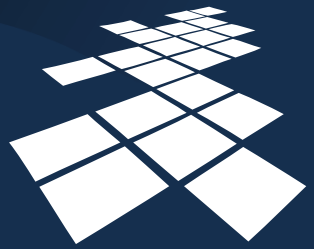
# Detekcja zakleszczenia (1)

Wykład prowadzą:

**Jerzy Brzeziński**
**Jacek Kobusiński**






## Plan wykładu

- Procesy aktywne i pasywne
- Definicja zakleszczenia
- Problem detekcji wystąpienia zakleszczenia
- Detekcja zakleszczenia dla modelu OR
- Detekcja zakleszczenia dla modelu AND
- Detekcja zakleszczenia w środowisku synchronicznym dla modelu „k spośród r”





## Wprowadzenie

Procesy tworzące przetwarzanie rozproszone komunikują się ze sobą za pomocą mechanizmu wymiany wiadomości, realizując wspólny cel przetwarzania. Jedne procesy wysyłają komunikaty zawierające żądania przydziału pewnych zasobów, inne – w odpowiedzi – przesyłają ewentualnie komunikaty potwierdzające przydział żądanych zasobów.





## Nieformalna definicja problemu

W przetwarzaniu rozproszonym może w ogólności wystąpić sytuacja, w której wszystkie procesy pewnego niepustego zbioru procesów oczekują na wiadomości (potwierdzające na przykład przydział zasobów) od innych procesów tego właśnie zbioru. Stan taki nazywany jest **zakleszczeniem rozproszonym** (ang. *distributed deadlock*).

## Procesy aktywne i pasywne

W każdej chwili proces może być w jednym z dwóch stanów: **aktywnym** albo **pasywnym**.

Proces aktywny może realizować przetwarzanie wykonując operacje odpowiadające zajściu zdarzeń wewnętrznych i komunikacyjnych.





## Warunek uaktywnienia

Warunek uaktywnienia jest wyrażony przez predykat:

$$ready_i(\mathcal{X}) \equiv (\mathcal{P}_i^A \supseteq \mathcal{X}) \wedge activate_i(\mathcal{X}) \qquad (5.1)$$





## Definicja problemu

Przez $deadlock(\mathcal{B})$ oznaczamy predykat stwierdzający, że w danej chwili $\tau$, niepusty zbiór procesów $\mathcal{B}$ jest zbiorem procesów zakleszczonych.





## Zakleszczenie w modelu jednostkowym

$$\begin{aligned} &deadlock(\mathcal{B}) \equiv \qquad (5.2)\\ &\quad (\mathcal{B} \subseteq \mathcal{P}) \wedge (\mathcal{B} \neq \emptyset) \wedge \\ &\quad (\forall P_i :: P_i \in \mathcal{B} :: (passive_i \wedge |\mathcal{D}_i| = 1 \wedge \\ &\qquad (\exists P_j :: P_j \in \mathcal{D}_i \cap \mathcal{B} :: (\neg\, in\text{-}transit_i[j] \wedge \neg\, available_i[j]\ ))))\end{aligned}$$

## Zakleszczenie w modelu AND

$$deadlock(\mathcal{B}) \equiv \quad (5.3)$$
$$(\mathcal{B} \subseteq \mathcal{P}) \wedge (\mathcal{B} \neq \emptyset) \wedge$$
$$(\forall P_i :: P_i \in \mathcal{B} :: (passive_i \wedge$$
$$(\exists P_j :: P_j \in \mathcal{D}_i \cap \mathcal{B} :: (\neg\, in\text{-}transit_i[j] \wedge \neg\, available_i[j]))))$$





## Zakleszczenie w modelu OR

$$deadlock(\mathcal{B}) \equiv \quad (5.4)$$
$$(\mathcal{B} \subseteq \mathcal{P}) \wedge (\mathcal{B} \neq \emptyset) \wedge$$
$$(\forall P_i :: P_i \in \mathcal{B} :: (passive_i \wedge$$
$$\mathcal{D}_i \subseteq \mathcal{B} \quad \wedge$$
$$(\forall P_j :: P_j \in \mathcal{D}_i :: (\neg\, in\text{-}transit_i[j] \wedge \neg\, available_i[j]))))$$





## Zakleszczenie w podstawowym modelu k spośród r

$$deadlock(\mathcal{B}) \equiv \quad (5.5)$$
$$(\mathcal{B} \subseteq \mathcal{P}) \wedge (\mathcal{B} \neq \emptyset) \wedge$$
$$(\forall P_i :: P_i \in \mathcal{B} :: (passive_i \wedge$$
$$(\exists \mathcal{B}_i :: \mathcal{B}_i \subseteq \mathcal{D}_i \cap \mathcal{B} ::$$
$$(|\mathcal{D}_i \setminus \mathcal{B}_i| < k_i \quad \wedge$$
$$(\forall P_j :: P_j \in \mathcal{B}_i :: (\neg\, in\text{-}transit_i[j] \wedge \neg\, available_i[j])))))$$





## Zakleszczenie w modelu OR – AND

$$deadlock(\mathcal{B}) \equiv \quad (5.6)$$
$$(\mathcal{B} \subseteq \mathcal{P}) \wedge (\mathcal{B} \neq \emptyset) \wedge$$
$$(\forall P_i :: P_i \in \mathcal{B} :: (passive_i \wedge$$
$$(\forall u :: 1 \leqslant u \leqslant q_i ::$$
$$(\exists P_j :: P_j \in \mathcal{D}_i^u \cap \mathcal{B} :: (\neg\, in\text{-}transit_i[j] \wedge \neg\, available_i[j])))))$$

## Zakleszczenie w modelu dysjunkcyjnym k spośród r

$$deadlock(\mathcal{B}) \equiv \qquad (5.7)$$
$$(\mathcal{B} \subseteq \mathcal{P}) \wedge (\mathcal{B} \neq \emptyset) \wedge$$
$$(\forall P_i :: P_i \in \mathcal{B} :: (passive_i \wedge (\forall\, u :: 1 \leqslant u \leqslant q_i ::$$
$$(\exists\, \mathcal{B}_i^u :: \mathcal{B}_i^u \subseteq \mathcal{D}_i^u \cap \mathcal{B} ::$$
$$(|\, \mathcal{D}_i^u \setminus \mathcal{B}_i^u \,| < k_i^u \;\wedge$$
$$(\forall P_j :: P_j \in \mathcal{B}_i^u :: (\neg\, in\text{-}transit_i[j] \wedge \neg\, available_i[j]\,)))))))$$





## Zakleszczenie w modelu predykatowym

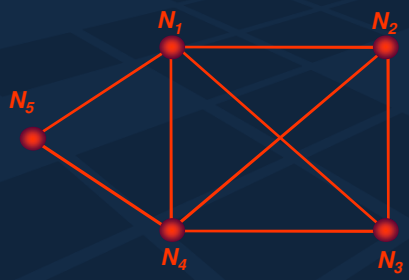
$$deadlock(\mathcal{B}) \equiv \qquad (5.8)$$
$$(\mathcal{B} \subseteq \mathcal{P}) \wedge (\mathcal{B} \neq \emptyset) \wedge$$
$$(\forall P_i :: P_i \in \mathcal{B} :: (passive_i \wedge \neg\, activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i \cup (\mathcal{P} \setminus \mathcal{B}))))$$





## Przykłady zakleszczeń

***Wait-For Graph (WFG):***




$N_1, N_2, N_3, N_4, N_5$ – węzły środowiska przetwarzania

$P_1, P_2, P_3, P_4, P_5$ – procesy wykonywane w odpowiednich węzłach

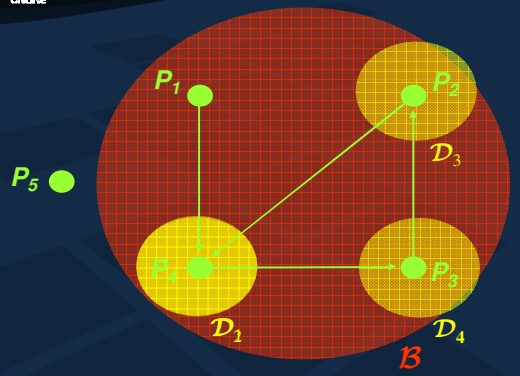
$P_i \longrightarrow P_j$ – łuk $\langle P_i, P_j \rangle$ reprezentujący fakt, że proces $P_i$ oczekuje na wiadomość od procesu $P_j$

- Każdy proces w grafie z łukiem wychodzącym jest pasywny.
- Procesy bez łuków wychodzących są aktywne.
- Zakładamy, że wszystkie kanały są puste.





## Przykład – model jednostkowy

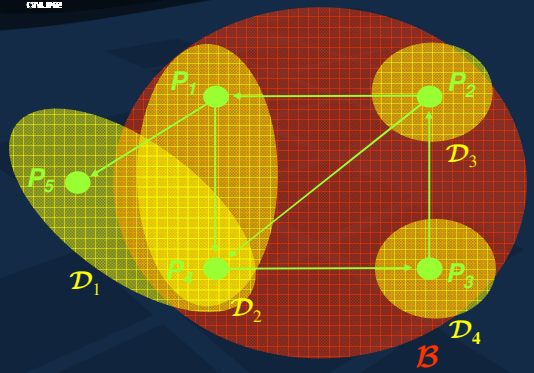



Zbiory warunkujące:

$\mathcal{D}_1 = \{P_4\}$,
$\mathcal{D}_2 = \{P_4\}$,
$\mathcal{D}_3 = \{P_2\}$,
$\mathcal{D}_4 = \{P_3\}$.

$deadlock(\mathcal{B})$ zachodzi dla $\mathcal{B} = \{P_1, P_2, P_3, P_4\}$

## Przykład – model AND




Zbiory warunkujące:

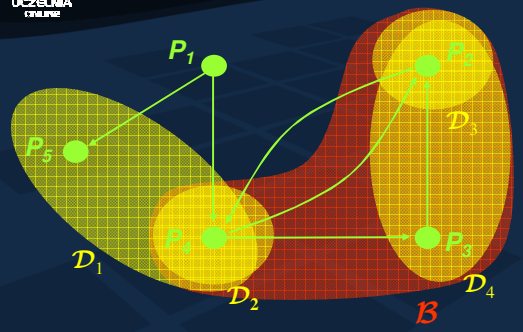
$\mathcal{D}_1 = \{P_4, P_5\}$,
$\mathcal{D}_2 = \{P_1, P_4\}$,
$\mathcal{D}_3 = \{P_2\}$,
$\mathcal{D}_4 = \{P_3\}$.

$deadlock(\mathcal{B})$ zachodzi dla $\mathcal{B} = \{P_1, P_2, P_3, P_4\}$





## Przykład – model OR




Zbiory warunkujące:

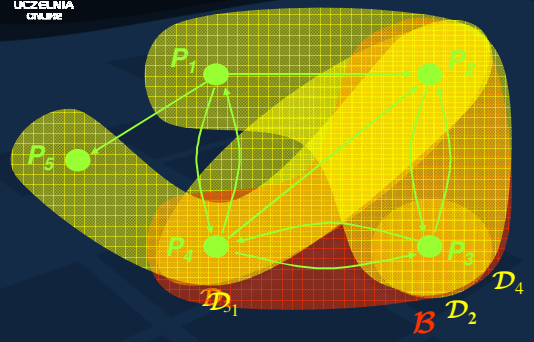
$\mathcal{D}_1 = \{P_4, P_5\}$,
$\mathcal{D}_2 = \{P_4\}$,
$\mathcal{D}_3 = \{P_2\}$,
$\mathcal{D}_4 = \{P_2, P_3\}$.

$deadlock(\mathcal{B})$ zachodzi dla $\mathcal{B} = \{P_2, P_3, P_4\}$





## Przykład – model k spośród r




Modele żądań:
dla $P_1$: 1 spośród 3
dla $P_2$: 1 spośród 1
dla $P_3$: 2 spośród 2
dla $P_4$: 2 spośród 3

Zbiory warunkujące:

$\mathcal{D}_1 = \{P_2, P_4, P_5\}$,
$\mathcal{D}_2 = \{P_3\}$,
$\mathcal{D}_3 = \{P_2, P_4\}$,
$\mathcal{D}_4 = \{P_1, P_2, P_3\}$.

$deadlock(\mathcal{B})$ zachodzi dla $\mathcal{B} = \{P_2, P_3, P_4\}$





## Klasyfikacja problemów detekcji zakleszczenia

- problem detekcji wystąpienia zakleszczenia
- problem detekcji zakleszczenia procesu
- problem detekcji zakleszczenia zbioru procesów
- problem detekcji maksymalnego zbioru procesów zakleszczonych

## Detekcja wystąpienia zakleszczenia

Czy istnieje w pewnej chwili zbiór $\mathcal{B}$, dla którego predykat *deadlock*($\mathcal{B}$) jest prawdziwy ?

Odpowiedź na to pytanie określa wartość predykatu:

$$dE \equiv (\exists \mathcal{B} :: deadlock(\mathcal{B})) \qquad (5.9)$$





## Detekcja wystąpienia zakleszczenia procesu

**Detekcja zakleszczenia procesu** $P_i$ sprowadza się do sprawdzenia czy prawdziwy jest predykat:

$$dP_i \equiv ( \exists \mathcal{B} :: deadlock(\mathcal{B}) ) \wedge P_i \in \mathcal{B} ) \qquad (5.10)$$





## Detekcja wystąpienia zakleszczenia zbioru procesów

**Detekcja zakleszczenia zbioru procesów** polega na znalezienia zbioru $\mathcal{B}^*$, dla którego prawdziwy jest następujący predykat:

$$deadlock(\mathcal{B}^*) \vee ((\mathcal{B}^* = \emptyset) \wedge ( \nexists \mathcal{B} :: deadlock(\mathcal{B}))) \qquad (5.11)$$





## Detekcja maksymalnego zbioru zakleszczonego

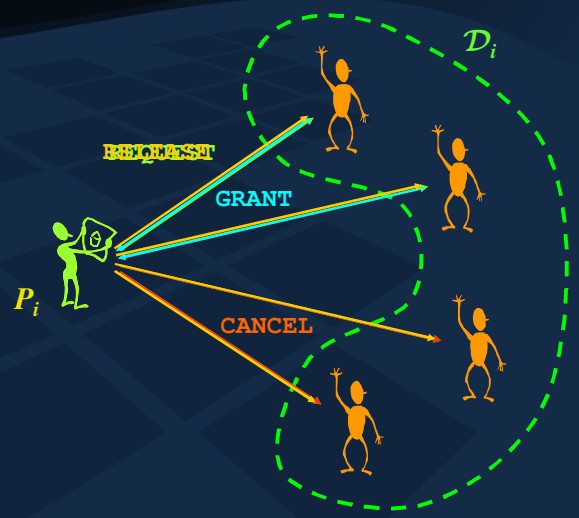
**Detekcja maksymalnego zbioru zakleszczonego**, sprowadza się do znalezienia takiego zbioru $\mathcal{B}^*$, dla którego spełniony jest warunek:

$$(deadlock(\mathcal{B}^*) \vee \mathcal{B}^* = \emptyset) \ \wedge \ (maxdead(\mathcal{B}^*)), \qquad (5.12)$$

*gdzie*

$$maxdead(\mathcal{B}^*) \equiv (\forall \mathcal{B} :: deadlock(\mathcal{B}) \Rightarrow (\mathcal{B} \subseteq \mathcal{B}^*)) \qquad (5.13)$$

## Model aplikacyjnego przetwarzania rozproszonego








## Alg. Chandy, Misra, Hass dla modelu AND (1)

```
type PROBE extends FRAME is
record of
   initIndex : INTEGER
end record

probeOut      : PROBE
granted_i     : array [1..n] of BOOLEAN := False
D_i           : set of PROCESS_ID
recvProbe_i   : array [1..n] of BOOLEAN := False
α_i           : INTEGER
k             : INTEGER
deadlockDetected_i  : BOOLEAN := False
```





## Alg. Chandy, Misra, Hass dla modelu AND (2)

```
1.  when e_start(Q_α , DeadlockDetection) do
2.   if passive_α
3.    then
4.        for all Q_k :: P_k ∈ D_α do
5.        probeOut.initIndex := α
6.        send(Q_α, Q_k, probeOut)
7.    end for
8.    end if
9.  end when
```





## Alg. Chandy, Misra, Hass dla modelu AND (3)

```
10. when e_activate(P_i) do
11.       for all k ∈ {1, 2, ... , n} do
12.          recvProbe_i[k]:= False
13.       end for
14. end when
```

## Alg. Chandy, Misra, Hass dla modelu AND (4)

```
15. when e_receive(Q_j, Q_i, probeIn: PROBE) do
16.   α_i := probeIn.InitIndex
17.   if passive_i ∧ (¬recvProbe_i[α_i]) ∧ (¬granted_i[j])
18.   then
19.     recvProbe_i[α_i] := True
20.     if α_i = i
21.     then
22.       deadlockDetected_i := True
23.       decide(deadlockDetected_i)
24.     else
25.       probeOut.initIndex := α_i
26.       for all Q_k :: P_k ∈ D_i do
27.         send(Q_i, Q_k, probeOut)
28.       end for
29.     end if
30.   end if
31.   end when
```





## Detekcja zakleszczenia dla modelu *OR* (1)

Algorytm detekcji zakleszczenia dla modelu *OR* opiera się na **przetwarzaniu dyfuzyjnym** (ang. *query computation*).





## Detekcja zakleszczenia dla modelu *OR* (2)

**Twierdzenie 5.2**
Jeżeli inicjator $Q_\alpha$ rozpoczyna detekcję w chwili, gdy jego proces aplikacyjny $P_\alpha$ jest zakleszczony, to $Q_\alpha$ stwierdzi zakleszczenie procesu $P_\alpha$ w skończonym czasie (algorytm detekcji zakończy się).





## Detekcja zakleszczenia dla modelu *OR* (3)

**Twierdzenie 5.2**
Jeżeli inicjator $Q_\alpha$ deklaruje, że jego proces aplikacyjny $P_\alpha$ jest zakleszczony, to $P_\alpha$ należy do pewnego zbioru procesów zakleszczonych w chwili zakończenia algorytmu.

## Alg. Chandy, Misra, Hass dla modelu OR (1)

```
type CONTROL extends FRAME is
   record of
     initIndex : INTEGER
     queryNo   : INTEGER
   end record

type QUERY extends CONTROL

type REPLY extends CONTROL
```





## Alg. Chandy, Misra, Hass dla modelu OR (2)

```
queryOut            : QUERY
replyOut            : REPLY
D_i                 : set of PROCESS_ID
maxQueryNo_i        : array [1..n] of INTEGER := 0
engager_i           : array [1..n] of INTEGER := 0
QRBalance_i         : array [1..n] of INTEGER := 0
contPassive_i       : array [1..n] of BOOLEAN := False
queryNo_i           : INTEGER
α_i                 : INTEGER
deadlockDetected_i  : BOOLEAN := False
```





## Alg. Chandy, Misra, Hass dla modelu OR (3)

```
1.  when e_start(Q_α, DeadlockDetection) do
2.    if passive_α
3.     then
4.       maxQueryNo_α[α] := maxQueryNo_α[α] + 1
5.      contPassive_α[α] := True
6.      queryOut.queryNo := maxQueryNo_α[α]
7.      queryOut.initIndex := α
8.      for all Q_k :: P_k ∈ D_α do
9.       send(Q_α, Q_k, queryOut)
10.     end for
11.     QRBalance_α[α] := | D_α |
12.   end if
13. end when
```





## Alg. Chandy, Misra, Hass dla modelu OR (4)

```
14. when e_receive(Q_j, Q_i, queryIn : QUERY) do
15.  if passive_i
16.   then
17.     queryNo_i := queryIn.queryNo
18.     α_i := queryIn.initIndex
19.     if queryNo_i > maxQueryNo_i[α_i]
20.       then
21.         maxQueryNo_i[α_i] := queryNo_i
22.         engager_i[α_i] := j
23.         contPassive_i[α_i] := True
24.         queryOut.queryNo := queryNo_i
25.         queryOut.initIndex := α_i
26.         for all Q_k :: P_k ∈ D_i do
27.           send(Q_i, Q_k, queryOut)
28.         end for
29.         QRBalance_i[α_i] := |D_i|
```

## Alg. Chandy, Misra, Hass dla modelu OR (5)

```
30.   else
31.     if contPassive_i[α_i] ∧ queryNo_i = maxQueryNo_i[α_i]
32.     then
33.       replyOut.queryNo := queryNo_i
34.       replyOut.initIndex := α_i
35.       send(Q_i, Q_j, replyOut)
36.     end if
37.   end if
38.  end if
39.  end when
```





## Alg. Chandy, Misra, Hass dla modelu OR (6)

```
40.  when e_receive(Q_j, Q_i, replyIn : REPLY) do
41.   if passive_i
42.   then
43.     queryNo_i:= queryIn.queryNo
44.     α_i:= queryIn.initIndex
45.     if maxQueryNo_i[α_i]=queryNo_i ∧ contPassive_i[α_i]
46.     then
47.       QRBalance_i[α_i]:= QRBalance_i[α_i] - 1
48.       if QRBalance_i[α_i] = 0
49.       then
50.         if α_i=i
51.         then
52.           deadlockDetected_i:= True
53.           decide(deadlockDetected_i)
```





## Alg. Chandy, Misra, Hass dla modelu OR (7)

```
54.       else
55.         k:= engager_i[α_i]
56.         replyOut.queryNo:= queryNo_i
57.         replyOut.initIndex:= α_i
58.         send(Q_i, Q_k, replyOut)
59.       end if
60.     end if
61.    end if
62.   end if
63.  end when
```





## Alg. Chandy, Misra, Hass dla modelu OR (8)

```
64.  when e_activate(P_i) do
65.    for all k ∈ {1, 2, ... , n} do
66.      contPassive_i[k] := False
67.    end for
68.  end when
```

# Alg. Bracha, Toueg'a (1)

```
type NOTIFY  extends SIGNAL
type DONE    extends SIGNAL
type CONFIRM extends SIGNAL
type ACK     extends SIGNAL
```





# Alg. Bracha, Toueg'a (2)

```
notifyOut            : NOTIFY
doneOut, doneIn      : DONE
confirmOut           : CONFIRM
ackOut, ackIn        : ACK
𝒜                    : set of record of
                         predId : PROCESS_ID
                         succId : PROCESS_ID
                       end record := {⟨P_i,P_j⟩ : P_j ∈𝒟_i }
𝒪𝒰𝒯_i                : set of PROCESS_ID := {P_j: ⟨P_i,P_j⟩∈ 𝒜}
ℐ𝒩_i                 : set of PROCESS_ID := {P_j: ⟨P_i,P_j⟩∈ 𝒜}
confirmNo_i          : INTEGER := 0
notified_i           : BOOLEAN := False
confirmed_i          : BOOLEAN := False
expectNo_i           : INTEGER
deadlockDetected_i   : BOOLEAN := False
```





# Alg. Bracha, Toueg'a (3)

```
1.  procedure NOTIFYPROC() do
2.    notified_i := True
3.    for all Q_k ∈ 𝒪𝒰𝒯_i do
4.      send( Q_i , Q_k , notifyOut)
5.    end for
6.    if expectNo_i = 0
7.    then
8.      CONFIRMPROC()
9.    end if
10.   for all Q_k ∈ 𝒪𝒰𝒯_i do
11.     receive(Q_k , Q_i , doneIn)
12.   end for
13. end procedure
```





# Alg. Bracha, Toueg'a (4)

```
14. procedure CONFIRMPROC() do
15.   confirmed_i := True
16.   for all Q_k ∈ ℐ𝒩_i do
17.     send(Q_i, Q_k, confirmOut)
18.   end for
19.   for all Q_k ∈ ℐ𝒩_i do
20.     receive(Q_k, Q_i, ackIn)
21.   end for
22. end procedure
```

# Alg. Bracha, Toueg'a (5)

```
23. when e_start(Q_α, DeadlockDetection) do
24.   NOTIFYPROC()
25.   if confirmNo_α < expectNo_α
26.   then
27.       deadlockDetected_α := True
28.       decide(deadlockDetected_α)
29.   end if
30. end when
```





# Alg. Bracha, Toueg'a (6)

```
31. when e_receive(Q_j, Q_i, confirmIn: CONFIRM) do
32.   confirmNo_i := confirmNo_i + 1
33.   if ¬confirmed_i ∧ (confirmNo_i ⩾ expectNo_i)
34.   then
35.     CONFIRMPROC()
36.   end if
37.   send(Q_i, Q_j, ackOut)
38. end when
```





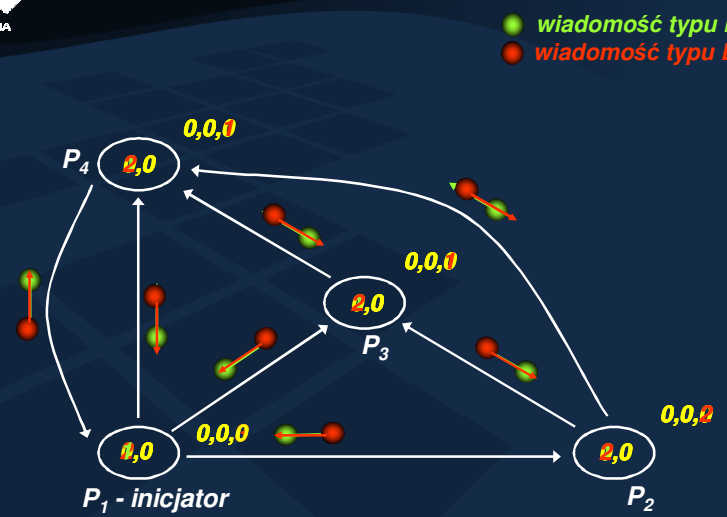
# Alg. Bracha, Toueg'a (7)

```
39. when e_receive(Q_j, Q_i, notifyIn: NOTIFY) do
40.   if ¬notified_i
41.   then
42.     NOTIFYPROC()
43.   end if
44.   send(Q_i ,Q_j ,doneOut)
45. end when
```

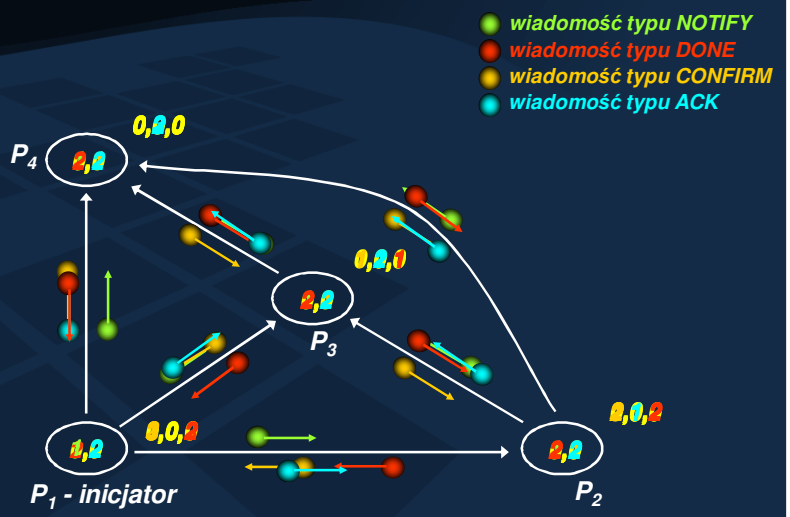




# Przykład działania algorytmu (1)



## Przykład działania algorytmu (2)








## Złożoność czasowa algorytmu detekcji zakleszczenia w środowisku synchronicznym dla modelu *k spośród r* (1)

Graf niezorientowany odpowiadający grafowi $WFG$ scharakteryzowany jest przez: średnicę grafu $d$, najdłuższą ścieżkę w grafie $l$.

Złożoność całego algorytmu to:

- złożoność fazy potwierdzania
- złożoność powiadomienia inicjatora fazy potwierdzania,
- złożoność odpowiadającą przesłaniu wiadomości typu DONE od inicjatora fazy potwierdzania do inicjatora detekcji.





## Złożoność czasowa algorytmu detekcji zakleszczenia w środowisku synchronicznym dla modelu *k spośród r* (2)

- przesłanie wiadomości typu `CONFIRM` może zabrać $l$ jednostek czasu
- transmisja wiadomości typu `ACK` również zabiera w najgorszym wypadku $l$ jednostek czasu
- przesłanie wiadomości typu `NOTIFY` (równolegle do monitorów wszystkich procesów tworzących zbiór $\mathcal{OUT}_i$) wymaga co najwyżej $d$ kroków.
- przesłanie wiadomości typu `DONE` po zakończeniu fazy potwierdzania wymaga co najwyżej $d$ kroków.

W efekcie złożoność czasowa algorytmu wynosi $2d + 2l$.