Detekcja zakleszczenia (2)

Plan wykładu

Celem wykładu jest zaznajomienie studenta z kolejnymi algorytmami detekcji zakleszczenia. Jest on jest bezpośrednią kontynuacją poprzedniego wykładu, w którym zdefiniowane i omówione zostały niezbędne teoretyczne podstawy. Student zapozna się w trakcie jego trwania z trzema kolejnymi algorytmami detekcji zakleszczenia: detekcji zakleszczenia w środowisku asynchronicznym dla modelu k spośród r, dwufazowego algorytmu detekcji zakleszczenia i na koniec z algorytmem detekcji zakleszczenia dla modelu predykatowego.

Algorytm detekcji zakleszczenia w środowisku asynchronicznym (1)

Załóżmy teraz, że w rozważanym środowisku rozproszonym czas transmisji w niezawodnych kanałach FIFO jest skończony ale nieznany. Wyznaczenie stanu globalnego takiego systemu musi zatem uwzględnić stany kanałów. Przyjmując, tak jak poprzednio, że aplikacyjne przetwarzanie rozproszone stosuje wiadomości typu `REQUEST`, `GRANT` i `CANCEL`, stan globalny można przedstawić jako **kolorowany graf oczekiwanych potwierdzeń** $WFG^C = \langle \mathcal{P}, \mathcal{A} \rangle$ (kolorowany graf WFG). Wiadomości w kanałach są przy tym modelowane przez odpowiednie kolory łuków.

Algorytm detekcji zakleszczenia w środowisku asynchronicznym (2)

Wyróżnia się cztery kolory łuków: *Grey*, *Black*, *White* i *Translucent*.

- Łukowi $\mathcal{A}_{i,j} = \langle P_i, P_j \rangle$ w grafie przypisuje się kolor *Grey*, jeżeli proces P_i wysłał do P_j wiadomość typu `REQUEST`, a P_j jeszcze tej wiadomości nie odebrał, ani też P_i nie wysłał jeszcze wiadomości typu `CANCEL`.
- Łukowi $\langle P_i, P_j \rangle$ w grafie WFG^C nadaje się kolor *Black*, jeżeli P_j odebrał już wiadomość typu `REQUEST` od P_i, lecz P_j jeszcze nie wysłał w odpowiedzi wiadomości typu `GRANT` do P_i, ani też P_i nie wysłał jeszcze wiadomości typu `CANCEL` do P_j.
- Łukowi $\langle P_i, P_j \rangle$ przypisuje się kolor *White*, jeżeli P_j wysłał już wiadomość typu `GRANT` do P_i, lecz P_i jeszcze jej nie odebrał, ani też P_i nie wysłał jeszcze do P_j wiadomości typu `CANCEL`.
- Łukowi $\langle P_i, P_j \rangle$ nadaje się kolor *Translucent*, jeżeli P_i wysłał wiadomość typu `CANCEL` do P_j, lecz P_j jeszcze jej nie odebrał.

Algorytm detekcji zakleszczenia w środowisku asynchronicznym (3)

Załóżmy teraz, że każdy monitor Q_i zna odpowiadające mu zbiory oraz kolory wszystkich krawędzi incydentnych grafu WFG^C. Zauważmy, że w tym wypadku, w przeciwieństwie do środowiska synchronicznego rozważanego poprzednio, nie jest prawdziwa relacja:

$$P_i \in \mathcal{OUT}_j \Leftrightarrow P_j \in \mathcal{IN}_i.$$

Jak łatwo zauważyć, łuk *Grey* w skończonym czasie zmieni swój kolor na *Black* albo *Translucent* w wyniku odebrania wiadomości typu `REQUEST` przez adresata lub wysłania przez nadawcę wiadomości typu `CANCEL`. Kolor *Grey* występuje zatem tylko przejściowo. Uwzględniając ten fakt, można dokonać odwzorowania kolorowanego grafu WFG^C w tradycyjny graf czarno-biały WFG^{BW} przez uwzględnienie w tym ostatnim tylko łuków w kolorze *Black*. Wynikowy graf WFG^{BW} jest wystarczający do detekcji zakleszczenia, choć wykrycie tego stanu staje się możliwe dopiero, gdy żądania zostaną uwzględnione w grafie, a więc gdy łuki *Grey* zmienią kolor na

Black. To opóźnienie detekcji nie podważa jednak poprawności rozwiązania. Powyższa transformacja oznacza przyjęcie wstępnej hipotezy, że żądania odpowiadające łukom *Grey* i *White* są już potwierdzone. Oczywiście, jeżeli przy takim założeniu zostanie wykryte zakleszczenie, to zakleszczenie występuje tym bardziej w stanie opisanym przez graf kolorowany WFG^C.

Przedstawioną powyżej koncepcję implementuje algorytm zaprezentowany na kolejnych slajdach. Przyjęto w nim, że warunkiem uaktywnienia procesu jest spełnienie relacji:

$$expectNo_i - outGreyWhiteNo_i \leqslant 0,$$

gdzie $outGreyWhiteNo_i$ oznacza sumaryczną liczbę wyjściowych łuków o kolorze *Grey* lub *White* wierzchołka P_i, a $expectNo_i$ – liczność zbioru $\mathcal{D}_i = \mathcal{OUT}_i$ (liczbę łuków wyjściowych w grafie WFG^C).

Problem wyznaczania kolorów można rozwiązać przez zastosowanie odpowiednich liczników wiadomości wysłanych i odebranych. Dalej algorytm ten może być uzupełniony i rozwinięty o fazę wyznaczania grafu kolorowanego WFG^C, wykonywaną równolegle z samym algorytmem detekcji.

Algorytm Bacha, Touega dla środowiska asynchronicznego (1)

Algorytm ten podobnie jak algorytm dla środowiska synchronicznego wykorzystuje cztery typy komunikatów. Są to komunikaty typu `NOTIFY`, `DONE`, `CONFIRM` i `ACK`.

Algorytm Bacha, Touega dla środowiska asynchronicznego (2)

Najważniejsze zmienne wykorzystywane przez ten algorytm są następujące:

- $\mathcal{A}^C$ – zadany zbiór łuków grafu WFG^C
- $\mathcal{OUT}_i^{Black}$ – zadany zbiór łuków wyjściowych wierzchołka P_i w kolorze *Black* grafu WFG^C
- $\mathcal{IN}_i^{Black}$ – zadany zbiór łuków wejściowych wierzchołka P_i w kolorze *Black* grafu WFG^C
- $expectNo_i$ – liczność zbioru $\mathcal{D}_i = \mathcal{OUT}_i$ (liczbę łuków wyjściowych w grafie WFG^C)
- $outArcColour_i$ – tablica reprezentująca kolory łuków wyjściowych wierzchołka P_i
- $inArcColour_i$ – tablica reprezentująca kolory łuków wejściowych wierzchołka P_i
- $outGreyWhiteNo_i$ – sumaryczną liczbę wyjściowych łuków o kolorze *Grey* lub *White* wierzchołka P_i

Algorytm Bacha, Touega dla środowiska asynchronicznego (3)

Podobnie jak w przypadku algorytmu tych autorów przedstawionym dla środowiska synchronicznego, procedura `NotifyProc` polega na rozesłaniu do wszystkich procesów należących do zbioru procesów sąsiednich wyjściowych, czyli należących do zbioru warunkującego procesu P_i wiadomości typu `NOTIFY` i zebraniu potwierdzeń typu `DONE`. Ilustruje ona pierwszą fazę algorytmu (fazę powiadamiania), w której wszystkie monitory są informowane o rozpoczęciu detekcji. Należy zauważyć że wywołanie procedury `ConfirmProc`, realizującej drugą fazę algorytmu jest zagnieżdżone w fazie powiadamiania, która tym samym kończy się dopiero po zakończeniu fazy drugiej.

Rozpoczęcie drugiej fazy jest możliwe dopiero po spełnieniu warunku mówiącego iż różnica liczby elementów w zbiorze warunkującym i sumarycznej liczby wyjściowych łuków o kolorze *Gray* lub *White* jest mniejsza lub równa 0 ($expectNo_i$ - $outGreyWhiteNo_i \leqslant 0$).

Algorytm Bacha, Touega dla środowiska asynchronicznego (4)

Również procedura ConfirmProc jest analogiczna do odpowiedniej procedury w algorytmie dla środowiska synchronicznego. Jest ona odpowiedzialna za realizację drugiej fazy przetwarzania, w której monitor Q_i rozsyła do wszystkich procesów, które są połączone z procesem P_i łukiem koloru $Black$ (oznaczającym że żądanie zasobu zostało odebrane, ale nie jest jeszcze potwierdzone), wiadomość typu CONFIRM i czeka na otrzymanie od wszystkich potwierdzenia otrzymania tej wiadomości, czyli komunikatu typu ACK.

Algorytm Bacha, Touega dla środowiska asynchronicznego (5)

Algorytm rozpoczyna inicjator Q_a, którego proces aplikacyjny jest pasywny, a więc potencjalnie zakleszczony, wywołując procedurę NotifyProc.

Algorytm kończy się, gdy nie ma już możliwości aktywowania kolejnych procesów i żadne nowe wiadomości typu CONFIRM nie zostaną przesłane. Jeśli pomimo zakończenia algorytmu, dla części procesów nadal nie są potencjalnie spełnione warunki uaktywnienia to uznaje się te procesy za procesy zakleszczone.

Algorytm Bacha, Touega dla środowiska asynchronicznego (6)

Jeśli przesyłanie wiadomości typu CONFIRM między monitorami jest możliwe, to należy interpretować to jako istnienie możliwości uaktywnienia procesu. W takim przypadku, odpowiedni monitor symuluje możliwe zachowanie uaktywnionego procesu, wywołując procedurę ConfirmProc. Po jej zakończeniu wysyła potwierdzenie odebrania tego komunikatu.

Algorytm Bacha, Touega dla środowiska asynchronicznego (7)

Odebranie wiadomości typu NOTIFY przez proces, do którego nie dotarła jeszcze wiadomość tego typu, skutkuje wywołaniem procedury NotifyProc.

Dwufazowy algorytm detekcji zakleszczenia

Kshemkalyani i Singhal przedstawili algorytm detekcji zakleszczenia rozproszonego w środowisku z niezawodnymi kanałami FIFO dla modelu *k spośród r*. Algorytm ten składa się z dwóch faz wymiany wiadomości: fazy inicjacji (ang. *outward sweep*) i fazy detekcji (ang. *inward sweep*). W fazie **inicjacji**, algorytm konstruuje spójny obraz globalnego rozproszonego grafu oczekiwanych potwierdzeń WFG. W fazie **detekcji**, wykonywana jest natomiast tak zwana **redukcja** utworzonego wcześniej grafu WFG, w celu stwierdzenia wystąpienia zakleszczenia. Realizacja obu powyższych faz może nakładać się w czasie.

Dwufazowy algorytm detekcji zakleszczenia dla *modelu k spośród r* (1)

Dwufazowy algorytm detekcji zakleszczenia dla modelu k spośród r wykorzystuje trzy typy komunikatów. Pierwszy z nich FLOOD jest odpowiednikiem żądania zasobu. Za jego pomocą w fazie inicjacji jest konstruowany graf *WFG*. Z kolei detekcja zakleszczenia dokonywana jest z wykorzystaniem wiadomości typu ECHO w trakcie fazy detekcji. Ostatni z typów, to typ SHORT. Wiadomości tego typu służą do stwierdzenia momentu zakończenia algorytmu detekcji.

Dwufazowy algorytm detekcji zakleszczenia dla *modelu k spośród r* (2)

Najistotniejsze zmienne wykorzystywane przez ten algorytm są następujące:

- $inState_i$ – tablica, której j-ty element określa zbiór procesów P_k, których monitory Q_k przesłały wiadomości typu `FLOOD` odebrane już przez Q_i w ramach ostatniego procesu detekcji zakleszczenia, zainicjowanego przez monitor Q_j.
- $outState_i$ – tablica, której j-ty element określa zbiór warunkujący procesu P_i, w chwili otrzymania wiadomości typu `FLOOD` w ramach ostatniego procesu detekcji zakleszczenia, zainicjowanego przez monitor Q_j.
- $expectNoState_i$ – liczba k wymaganych przez aplikację wiadomości potwierdzeń, w chwili otrzymania wiadomości `FLOOD` w ramach ostatniego procesu detekcji, zainicjowanego przez monitor Q_j
- $latestInitClock_i$ – czas logiczny rozpoczęcia ostatniej detekcji przez Q_i
- $latestBlockClock_i$ – moment w którym ostatnio zablokowany był proces P_i
- $weight_i$ – waga będąca w dyspozycji inicjatora Q_i procesu detekcji zakleszczenia

Dwufazowy algorytm detekcji zakleszczenia dla *modelu k spośród r* (3)

Procedura `FILL` jest procedurą pomocniczą, której celem jest uproszczenie zapisu algorytmu. W efekcie wykonania tej procedury, wypełniony zostaje podany jako argument pakiet, Poszczególne pola tego pakietu są inicjowane wartościami przekazanymi jako kolejne argumenty.

Kiedy proces P_i staje się pasywny, jego monitor Q_i zapamiętuje stan lokalny procesu oraz skalarny czas logiczny $latestBlockClock_i$ odpowiadający momentowi przyjęcia po raz ostatni przez P_i stanu pasywnego. Następnie Q_i może zainicjować detekcję zakleszczenia (wówczas Q_i oznaczymy przez Q_α) wysyłając wiadomość typu `FLOOD` do monitorów wszystkich procesów należących do $\mathcal{OUT}_i = \mathcal{D}_i$ w chwili $latestBlockClock_i$. Ponadto, inicjator Q_α ma zmienną $weight_\alpha$, w której zapamiętuje zwrócone wagi.

Dwufazowy algorytm detekcji zakleszczenia dla *modelu k spośród r* (4)

Niezależnie, gdy monitor Q_i otrzyma pierwszą wiadomość typu `FLOOD` przez kanał reprezentowany w grafie WFG, zapamiętuje stan procesu P_i związany z postrzeganym przez niego aktualnym obrazem grafu WFG ($\mathcal{OUT}_i$, $\mathcal{IN}_i$), $latestBlockClock_i$, oraz proces od którego otrzymał wiadomość typu `FLOOD`. Stan ten obejmuje aktualne wartości zbioru warunkującego $\mathcal{OUT}_i = \mathcal{D}_i$, wymaganej do uaktywnienia liczby k_i wiadomości `GRANT`, czasu logicznego inicjacji danego procesu detekcji. Stan zapamiętywany jest odpowiednio w zmiennych: $outState_i[\alpha]$, $expectNoState_i[\alpha]$ oraz $latestInitClock_i[\alpha]$.

Jeśli w tym czasie proces P_i jest pasywny, Q_i wysyła wiadomość typu `FLOOD` do monitorów wszystkich procesów odpowiedniego zbioru warunkującego $\mathcal{OUT}_i$, w celu zagwarantowania, że wszystkie monitory osiągalne w WFG z węzła inicjatora, będą uczestniczyły w tworzeniu i zapamiętywaniu spójnego obrazu grafu WFG.

Dwufazowy algorytm detekcji zakleszczenia dla *modelu k spośród r* (5)

Jeżeli natomiast proces P_i jest aktywny w chwili otrzymania wiadomości typu FLOOD (nie ma żadnych łuków wychodzących w grafie WFG), to Q_i inicjuje redukcję łuków wejściowych wierzchołka P_i grafu WFG, przez przesłanie w odpowiedzi wiadomości typu ECHO.

Jeżeli wiadomość typu FLOOD dotrze do wierzchołka już zredukowanego, monitor po prostu odpowiada wysłaniem wiadomości typu ECHO.

Dwufazowy algorytm detekcji zakleszczenia dla *modelu k spośród r* (6)

Wiadomość typu ECHO realizuje redukcję wierzchołków i łuków grafu WFG, symulując w fazie detekcji przesyłanie wiadomości aplikacyjnych typu GRANT. Jeśli wiadomość typu ECHO dociera do węzła już zredukowanego, to jej waga jest wysłana wprost do inicjatora w wiadomości SHORT.

Dwufazowy algorytm detekcji zakleszczenia dla *modelu k spośród r* (7)

Dla modelu żądań *k spośród r*, wierzchołek grafu jest redukowany, gdy odpowiedni monitor Q_i otrzyma k_i wiadomości typu ECHO. Po redukcji wierzchołka, monitor wysyła wiadomości typu ECHO do wszystkich monitorów odpowiadających poprzednikom w grafie WFG, podtrzymując tym samym fazę detekcji. Wysłane wiadomości typu ECHO mogą z kolei spowodować redukcję kolejnych wierzchołków itd. Na koniec, monitor inicjujący Q_α stwierdza wystąpienie zakleszczenia, jeżeli nie został on zredukowany do chwili zakończenia algorytmu. Zakleszczone procesy reprezentowane są przy tym przez wszystkie nie zredukowane wierzchołki grafu WFG.

Jeśli wiadomość typu ECHO nie powoduje redukcji węzła to jej waga jest wysłana wprost do inicjatora w wiadomości SHORT.

W ogólności, redukcja spójnego obrazu grafu WFG może się rozpocząć w wierzchołku nie będącym liściem grafu, przed zapamiętaniem całego grafu. Ma to miejsce, gdy wiadomość typu ECHO dotrze do monitora Q_i i zainicjuje redukcję odpowiadającego mu wierzchołka grafu WFG przed nadejściem wiadomości typu FLOOD wszystkimi kanałami wejściowymi reprezentowanymi w grafie, a więc przed zapamiętaniem w całości lokalnej części grafu.

Tak więc dwa działania związane z tworzeniem obrazu spójnego grafu WFG i redukcją tego grafu, realizowane są współbieżnie i żadna dodatkowa synchronizacja nie jest potrzebna.

Dwufazowy algorytm detekcji zakleszczenia dla *modelu k spośród r* (8)

Zauważmy, że w prezentowanym algorytmie pojawia się problem stwierdzenia zakończenia algorytmu detekcji. W celu rozwiązania tego problemu, zastosowano **technikę wag**, wprowadzając dodatkowo wiadomości typu SHORT. W chwili gdy detekcja jest inicjowana, waga o wartości 1 jest równomiernie dzielona i dołączana do wiadomości typu FLOOD wysyłanych przez inicjatora.

Jeżeli odbierana jest przez monitor Q_i pierwsza wiadomość typu FLOOD, waga tej wiadomości jest ponownie równomiernie dzielona i wysyłana wraz z propagowanymi dalej wiadomościami typu FLOOD. Wagi wszystkich kolejnych wiadomości typu FLOOD, są przesyłane od razu wprost do inicjatora, z wykorzystaniem wiadomości typu SHORT. Kiedy wiadomość typu FLOOD dotrze do liścia grafu, jej waga jest zwracana wraz z wiadomością typu ECHO.

W przypadku odebrania wiadomości typu SHORT należy sprawdzić, czy nie jest to wiadomość nieaktualna. Jeśli tak jest w istocie to wiadomość jest ignorowana. W przeciwnym przypadku aktualizowana jest waga procesu inicjatora poprzez dodanie wagi zawartej w odebranym komunikacie.

Algorytm gwarantuje, że niezmienna, równa 1, pozostaje stale suma wag w wiadomościach typu FLOOD, ECHO i SHORT, powiększona o wagę inicjatora, wynikającą z otrzymanych już przez niego wiadomości typu SHORT i ECHO. Algorytm kończy się zatem, gdy waga inicjatora osiągnie wartość 1, co oznacza, że wszystkie działania związane z tworzeniem, zapamiętywaniem oraz redukcją obrazu grafu *WFG*, zostały zakończone.

Przykład działania dwufazowego algorytmu detekcji zakleszczenia rozproszonego

Przeanalizujmy działanie tego algorytmu na konkretnym przykładzie przedstawionym w prezentacji. Przyjmiemy ponownie dla uproszczenia, że warunkiem uaktywnienia procesu jest otrzymanie potwierdzenia GRANT od wszystkich procesów tworzących zbiór warunkujący. Niech Q_1 będzie inicjatorem procesu detekcji zakleszczenia rozproszonego, a P_4 – jedynym aktywnym procesem. Najpierw, inicjator $Q_\alpha = Q_1$ przekazuje do monitorów procesów tworzących jego zbiór warunkujący wiadomości typu FLOOD, każdą z wagą 1/3. Ponieważ proces P_4 jest aktywny, monitor Q_4 może od razu wysłać w odpowiedzi wiadomość typu ECHO, z wagą 1/3 równą wadze otrzymanej wiadomości typu FLOOD. Z kolei monitor Q_3 przekazuje wiadomość FLOOD do następnika, a Q_2 przekazuje wiadomość typu FLOOD do Q_3 i Q_4. Monitor Q_2 dzieli oczywiście wagę otrzymanej wiadomości FLOOD na pół, a więc wysyła wiadomości z wagami 1/6. W następnym kroku, monitor Q_4 jako skojarzony z procesem aktywnym, odpowiada wiadomościami typu ECHO do Q_3 i Q_2, z wagami równymi odpowiednio 1/3 i 1/6. Monitor Q_3 wysyła natomiast wiadomość typu SHORT bezpośrednio do inicjatora w odpowiedzi na odebranie kolejnej wiadomości typu FLOOD od Q_2. Następnie, po otrzymaniu wiadomości typu ECHO z wagą 1/3, Q_3 przekazuje tę wiadomość do swoich poprzedników Q_1 i Q_2 , dzieląc na pół otrzymaną wagę. Ponieważ pojedyncza wiadomość aplikacyjna typu GRANT symulowana tu przez wiadomość typu ECHO od Q_4 nie byłaby wystarczająca do uaktywnienia P_2, Q_2 wysyła wiadomość typu SHORT z wagą 1/6 do inicjatora Q_1. W ostatnim kroku, po otrzymaniu wiadomości ECHO od Q_3, monitor Q_2 przekaże dalej wiadomość typu ECHO do Q_1. Algorytm kończy się, gdy inicjator Q_1 otrzyma z powrotem wiadomości z sumaryczną wagą równą 1.

Analiza złożoności czasowej dwufazowego algorytmu detekcji zakleszczenia

Przeanalizujmy złożoność czasową algorytmu, przyjmując podobnie jak poprzednio, że graf niezorientowany odpowiadający grafowi *WFG*, jest scharakteryzowany przez średnicę d i najdłuższą ścieżkę l.

Zauważmy, że przesłanie wiadomości typu ECHO przez wszystkie krawędzie (kanały) musi być poprzedzone przesłaniem wiadomości FLOOD w kierunku przeciwnym. Jak łatwo sprawdzić, przejście wiadomości typu FLOOD przez wszystkie krawędzie wymaga $d+1$ kroków. Stosując następnie argumentację podobną do tej odnoszącej się poprzednio do wiadomości typu GRANT w algorytmie detekcji zakleszczenia w środowisku synchronicznym dla modelu k spośród r, stwierdzamy, że w rozważanym wyżej przykładzie dla uaktywnienia procesu P_2 wymagane byłoby odebranie obu wiadomości typu GRANT, symulowanych tu przez wiadomości typu ECHO, wysłanych przez Q_3 i Q_4.

W ogólności czas potrzebny na dotarcie wiadomości typu ECHO od monitora Q_4 procesu aktywnego do Q_2, zależy od najdłuższej ścieżki między Q_4 a Q_2. Tak więc, najdłuższa ścieżka w grafie niezorientowanym odpowiadającym grafowi determinuje czas wymagany na propagację wiadomości typu ECHO od monitora procesu aktywnego do inicjatora. Obserwacja ta prowadzi wprost do wniosku, że złożoność czasowa algorytmu wynosi $(d + 1) + l$.

Detekcja zakleszczenia dla modelu predykatowego (1)

W przedstawionym dalej algorytmie detekcji zakleszczenia dla modelu predykatowego wykorzystana jest koncepcja **ciągu** (**łańcucha, sekwencji**) **cykli detekcyjnych** (ang. *wave sequence*), polegająca na wielokrotnym w ogólności komunikowaniu się inicjatora procesu

detekcji z pozostałymi monitorami w celu wyznaczenia stanu globalnego, lub wartości predykatu globalnego.

Istotne jest przy tym, że po zainicjowaniu jednego cyklu, następny cykl może być rozpoczęty dopiero po uzyskaniu odpowiedzi od wszystkich monitorów, czyli po zakończeniu poprzedniego cyklu. Warunek ten jest jedynym warunkiem synchronizacyjnym w tym podejściu.

Detekcja zakleszczenia dla modelu predykatowego (2)

W celu uproszczenia prezentacji algorytmu detekcji zakleszczenia dla modelu predykatowego przyjmiemy, że monitory procesów aplikacyjnych tworzą strukturę pierścienia. Niech ponadto $succ(i) = (i) \ mod_n + 1$, a $pred(i) = (i + n - 2) \ mod_n + 1$.

Algorytm detekcji zakleszczenia dla modelu *predykatowego* (1)

Omawiany algorytm wykorzystuje trzy typu wiadomości. Pierwszy z nich, typ `FRAME`, to pakiet służący do przenoszenia wiadomości aplikacyjnej. Drugi typ to typ `TOKEN`, który będzie reprezentował znacznik przesyłany pomiędzy monitorami. Zawiera on dwa pola, pierwsze z nich $\mathcal{PD}$, zawiera informację o zbiorze procesów uważanych za zakleszczone, drugie o nazwie *firstWave* informuje czy jest to pierwszy, czy też kolejny obieg znacznika w pierścieniu. Ostatni z typów, to typ `ACK`, który będzie wykorzystywany do potwierdzeń przesyłanych między monitorami.

Algorytm detekcji zakleszczenia dla modelu *predykatowego* (2)

Najistotniejsze zmienne wykorzystywane przez ten algorytm są następujące:

- $firstWave_i$ – wartość $True$ oznacza że dla Q_i nie zakończył się pierwszy cykl detekcji.
- $contPassive_i$ – wartość $True$ tej zmiennej oznacza, że proces P_i był pasywny przez cały czas od ostatniego wysłania znacznika `TOKEN`.
- $\mathcal{PD}_i$ – zbiór procesów potencjalnie zakleszczonych
- $notAck_i$ – liczba niepotwierdzonych wiadomości wysłanych przez P_i.
- $passive_i$ – zmienna oznaczająca, czy proces P_i jest pasywny.

Algorytm detekcji zakleszczenia dla modelu *predykatowego* (3)

Procedura `WaitUpdateProc` jest odpowiedzialna za przetrzymanie znacznika do czasu spełnienia odpowiedniego warunku. Ponadto w pierwszym przebiegu wiadomości typu `TOKEN` inicjalizowana jest zmienna $contPassive_i$.

Znacznik jest zatrzymywany i przechowywany w monitorze Q_i aż do momentu τ spełnienia jednego z następujących warunków:

1. Proces P_i nie był procesem pasywnym przez cały czas od ostatniego wysłania znacznika.
2. Możliwe jest uaktywnienie procesu P_i po odebraniu wiadomości aktualnie dostępnych lub takich które zostaną wysłane przez procesy nie należące do zbioru procesów zakleszczonych.
3. Liczba wiadomości niepotwierdzonych wysłanych przez proces P_i wynosi 0.

Warto zauważyć, że nie jest możliwe by nie spełniony był jeden z warunków 1 lub 3 przez czas nieograniczony, gdyż procesy pasywne nie wysyłają nowych wiadomości a, z drugiej strony,

wszystkie wiadomości zostają potwierdzone w skończonym czasie. Tak więc gwarantuje to że znacznik zostanie w końcu przesłany do kolejnego procesu w pierścieniu.

Algorytm detekcji zakleszczenia dla modelu *predykatowego* (4)

W algorytmie tym proces detekcji zakleszczenia inicjuje monitor Q_α wysyłając znacznik typu TOKEN do kolejnego monitora w pierścieniu. Znacznik zawiera zbiór procesów $\mathcal{PD}$, które zgodnie z bieżącą wiedzą, są potencjalnie zakleszczone. Początkowo $\mathcal{PD} = \mathcal{P}$, gdzie $\mathcal{P}$ oznacza zbiór wszystkich procesów.

Algortym detekcji zakleszczenia dla modelu *predykatowego* (5)

Po odebraniu znacznika proces, uaktualnia na podstawie danych w nim zawartych lokalny zbiór procesów potencjalnie zakleszczonych. Zauważmy, że znacznik przebywający i przetwarzany w monitorze Q_i jest natychmiast przesyłany dalej, gdy proces $P_i \notin \mathcal{PD}$**.** W przeciwnym razie, wywoływana jest omówiona wcześniej procedura WAITUPDATEPROC. Jeśli spełnione zostaną warunki pozwalające na zakończenie przetrzymywania znacznika, to realizowane jest jego przesłanie do kolejnego monitora w pierścieniu.

Algorytm detekcji zakleszczenia dla modelu *predykatowego* (6)

Jeżeli znacznik zostanie odebrany przez monitor inicjujący detekcję to podobnie jak w przypadku każdego innego monitora, aktualizowany jest lokalny zbiór procesów potencjalnie zakleszczonych i znacznik jest ewentualnie wstrzymywany zgodnie z omówionymi wcześniej zasadami. Następnie znacznik jest przesyłany dalej jeśli dopiero co zakończony obieg był pierwszym obiegiem znacznika w pierścieniu lub gdy zmienił się od ostatniej wizyty znacznika zbiór procesów potencjalnie zakleszczonych, który nie jest zbiorem pustym.

W przeciwnym wypadku, czyli gdy zbiór procesów potencjalnie zakleszczonych nie zmienił się podczas kolejnego obiegu znacznika proces inicjatora stwierdza że procesy znajdujące się w tym zbiorze są procesami zakleszczonymi. Oczywiście jeśli zbiór ten jest zbiorem pustym o zakleszczeniu nie może być mowy.

Algorytm detekcji zakleszczenia dla modelu *predykatowego* (7)

Wysłanie wiadomości aplikacyjnej powoduje każdorazowo zwiększenie zmiennej $notAck_i$ informującej o niepotwierdzonych wiadomościach wysłanych przez proces P_i.

Odebranie wiadomości aplikacyjnej przez monitor Q_i powoduje dostarczenie tej wiadomości do procesu P_i i wysłanie potwierdzenia typu ACK do monitora skojarzonego z procesem nadawcy.

Odebranie takiego potwierdzenia przez monitor umożliwia zmniejszenie o 1 wartości zmiennej $notAck_i$.

W sytuacji w której proces staje się procesem aktywnym, zmiennej $contPassive_i$ jest nadawana wartość $False$.

Jak wiadomo w celu wykazania poprawności algorytmu należy udowodnić twierdzenia mówiące o jego żywotności i bezpieczeństwie. Rozważając ciąg cykli wykonania algorytmu , oznaczymy przez τ_i^k moment czasu globalnego zakończenia cyklu k - tego w monitorze Q_i, a więc moment wysłania znacznika z Q_i do Q_{i+1} w k - tym cyklu. Analogicznie, τ_α^k oznaczać będzie moment zakończenia cyklu k - tego w monitorze inicjatora Q_α, to znaczy moment zakończenia całego cyklu k. Ponadto, dla każdego elementu X (zmiennej, predykatu), $X[\tau]$ oznaczać będzie wartości elementu X w chwili τ. W szczególności, $\mathcal{PD}[\tau_i^k]$ oznacza wartość zbioru $\mathcal{PD}$ zawartego w znaczniku, w momencie wysłania znacznika przez monitor Q_i w ramach k - tego cyklu detekcji. Przyjmiemy jeszcze następującą definicję:

$$contPassive_i^*(\tau', \tau'') \equiv (\tau' \leq \tau'') \wedge \forall \tau :: \tau' \leq \tau \leq \tau'' :: passive_i[\tau]$$

Jeżeli algorytm detekcji zakleszczenia zostanie zainicjowany w chwili τ_b, to zakończy się w skończonym czasie, w pewnej chwili τ_e, $\tau_e > \tau_b$.

Dowód:

Zauważmy, że znacznik przebywający i przetwarzany w monitorze Q_i jest natychmiast przesyłany dalej, gdy $P_i \notin \mathcal{PD}$. W przeciwnym razie, znacznik jest zatrzymywany i przechowywany w monitorze Q_i aż do momentu τ spełnienia następującego warunku:

$$(\neg contPassive_i[\tau]) \vee (activate_i(\mathcal{AV}_i \cup (\mathcal{P} \setminus \mathcal{PD})[\tau]) \vee (notAck_i = 0)[\tau]$$

Oczywiście warunek $(contPassive_i[\tau]) \wedge (notAck_i \neq 0)[\tau])$ nie może zachodzić przez czas nieograniczony, gdyż procesy pasywne nie wysyłają nowych wiadomości i, z drugiej strony, wszystkie wiadomości zostają potwierdzone w skończonym czasie. Tak więc w końcu warunek wysłania znacznika zostanie spełniony, a w konsekwencji cały cykl również zakończy się w skończonym czasie. Następny cykl, $k + 2$ jest inicjowany, gdy $|\mathcal{PD}\ [\tau_\alpha^k]| > |\mathcal{PD}\ [\tau_\alpha^k + 1]|$. Wówczas funkcja: $k \rightarrow |\mathcal{PD}\ [\tau_\alpha^k]|$ jest monotonicznie malejąca, z początkową wartością $|\mathcal{P}| = n$. Stąd liczba cykli jest skończona, a więc w skończonym czasie algorytm detekcji zakończy się.

Niech τb oraz τe oznaczają odpowiednio moment rozpoczęcia i zakończenia algorytmu detekcji zakleszczenia:

a) Jeżeli algorytm kończy się i zbiór $\mathcal{PD}$ jest zbiorem pustym ($\mathcal{PD} = \varnothing$), to dla każdego zbioru $\mathcal{B}$, $\mathcal{B} \neq \varnothing$ i $\mathcal{B} \subseteq \mathcal{P}$, predykat $deadlock(\mathcal{B})$ miał wartość $False$ w chwili τ_b.

b) Jeżeli algorytm kończy się i zbiór $\mathcal{PD}$ nie jest zbiorem pustym ($\mathcal{PD} \neq \varnothing$), to predykat $deadlock(\mathcal{PD})$ ma wartość $True$ w chwili τ_e. Ponadto dla każdego zbioru $\mathcal{B}$, takiego, że $deadlock(\mathcal{B})$ miał wartość $True$ w chwili τ_b , $\mathcal{B}$ jest podzbiorem zbioru $\mathcal{PD}$ ($\mathcal{B} \subseteq \mathcal{PD}$).

Dowód implikacji a)

W celu udowodnienia implikacji a), wykażemy następującą implikację równoważną:

$$(\exists \mathcal{B} :: \mathcal{B} \subseteq \mathcal{P} :: deadlock(\mathcal{B})\ [\tau_b]) \Rightarrow (\mathcal{PD}\ [\tau_e] \neq \varnothing)$$

Zgodnie z założeniem, w chwili τ_b zachodzi predykat $deadlock(\mathcal{B})$ i $\mathcal{B} \neq \varnothing$. $\mathcal{PD}$ początkowo jest równe $\mathcal{P}$ i stąd oczywiście w chwili τ_b, $\mathcal{B} \subseteq \mathcal{PD}$. Ponieważ $deadlock(\mathcal{B})$ jest predykatem stabilnym, prawdą jest, że:

$$\forall \tau :: \tau \geqslant \tau_b :: deadlock(\mathcal{B})\ [\tau]$$

Stąd, korzystając z definicji zakleszczenia dla modelu predykatowego, otrzymujemy:

$$\forall \tau :: \tau \geqslant \tau_b :: (\forall P_i :: P_i \in \mathcal{B} :: passive_i[\tau] \wedge \neg activate_i(\mathcal{IT}_i[\tau] \cup \mathcal{AV}_i[\tau] \cup \mathcal{P} \setminus \mathcal{B}))$$

Wszystkie procesy P_i należące do $\mathcal{B}$ są stale pasywne począwszy od chwili τ_b, a więc od pierwszej wizyty znacznika. Pokażemy teraz *nie wprost*, że żaden proces należący do $\mathcal{B}$ nie może być usunięty z $\mathcal{PD}$. Postawmy zatem hipotezę, że w chwili $\tau > \tau_b$, tuż przed możliwym usunięciem z $\mathcal{PD}$ pewnego procesu P_i, zachodzi relacja $\mathcal{B} \subseteq \mathcal{PD}$. Stale pasywny proces P_i może być usunięty z $\mathcal{PD}$, pod warunkiem, że spełniona jest w chwili τ relacja:

$$activate_i(\mathcal{AV}_i[\tau] \cup \mathcal{P} \setminus \mathcal{PD}\ [\tau]) = True$$

Zauważmy jednak, że $\mathcal{AV}_i[\tau] = \mathcal{AV}_i[\tau_b] \cup \{\mathcal{P}_k$: wiadomości wysłane z P_k do P_i stały się dostępne dla P_i w przedziale $\langle \tau_b, \tau \rangle$ }.

Rozważmy teraz wiadomość, która stała się dostępna dla P_i w przedziale czasu $\langle \tau_b, \tau \rangle$. Jej nadawca P_j albo należał do $\mathcal{P} \setminus \mathcal{B}$, albo jest procesem należącym do $\mathcal{B}$, którego kanał wyjściowy $C_{j,i}$ nie był pusty w chwili τ_b. Tak więc:

$$\mathcal{AV}_i[\tau] \subseteq (\mathcal{AV}_i[\tau_b] \cup \mathcal{IT}_i[\tau_b] \cup \mathcal{P} \setminus \mathcal{B})$$

Z postawionej hipotezy wiemy, że $activate_i(\mathcal{AV}_i[\tau] \cup \mathcal{P} \setminus \mathcal{PD}[\tau]) = True$, a stąd otrzymujemy:

$$activate_i(\mathcal{AV}_i[\tau_b] \cup \mathcal{IT}_i[\tau_b] \cup \mathcal{P} \setminus \mathcal{B} \cup \mathcal{P} \setminus \mathcal{PD}[\tau]) = True.$$

Ponieważ $\mathcal{B} \subseteq \mathcal{PD}[\tau]$, powyższa relacja redukuje się do:

$$activate_i(\mathcal{AV}_i[\tau_b] \cup \mathcal{IT}_i[\tau_b] \cup \mathcal{P} \setminus \mathcal{B}) = True.$$

Zauważmy, że ostatnia równość jest w sprzeczności z przyjętym założeniem: $deadlock(\mathcal{B})[\tau_b] = True$. Tak więc, żaden proces należący do $\mathcal{B}$ nie może być usunięty z $\mathcal{PD}$, co kończy dowód implikacji a).

Zauważmy jednak jeszcze, że gdy algorytm kończy się i $\mathcal{PD} \neq \varnothing$ to otrzymujemy:

$$\mathcal{B} \neq \varnothing \wedge \mathcal{B} \subseteq \mathcal{PD}$$

Dowód implikacji b)

Zgodnie z konstrukcją, algorytm kończy się w chwili $\tau_e = \tau_\alpha^{k+1}$ z wynikiem $\mathcal{PD} \neq \varnothing$ wtedy i tylko wtedy, gdy

$$|\mathcal{PD}[\tau_\alpha^{k+1}]| = |\mathcal{PD}[\tau_\alpha^{k}]| \text{ i } \mathcal{PD}[\tau_\alpha^{k+1}] \neq \varnothing$$

Niech τ_x, spełnia relację $max(\tau_i^k) \leqslant \tau_x \leqslant min(\tau_i^{k+1})$, Chwila taka istnieje, gdyż przykładowo τ_x może być równe τ_i^k. W szczególności zachodzi:

$$\forall P_i :: P_i \in \mathcal{P} :: \tau_i^k \leqslant \tau_\alpha^k \leqslant \tau_x \leqslant \tau_i^{k+1} \leqslant \tau_\alpha^k + 1$$

Ponieważ zbiór $\mathcal{DP}$ może jedynie ulegać redukcji, otrzymujemy:

$$\forall P_i :: P_i \in \mathcal{P} :: \mathcal{PD}[\tau_i^k] \supseteq \mathcal{PD}[\tau_\alpha^k] \supseteq \mathcal{PD}[\tau_x] \supseteq \mathcal{PD}[\tau_i^{k+1}] \supseteq \mathcal{PD}[\tau_\alpha^{k+1}]$$

Tak więc, jeżeli algorytm zakończy się w chwili $\tau = \tau_\alpha^{k+1}$ i $\mathcal{PD}[\tau] \neq \varnothing$, to

$$\forall P_i :: P_i \in \mathcal{P} :: \mathcal{PD}[\tau_\alpha^k] = \mathcal{PD}[\tau_x] = \mathcal{PD}[\tau_i^{k+1}] = \mathcal{PD}[\tau_\alpha^{k+1}]$$

Niech $\mathcal{PD}^*$ będzie oznaczać zbiór spełniający powyższą zależność. Udowodnimy, że $deadlock(\mathcal{PD}^*)$ zachodzi w chwili $\tau = \tau_x$, to znaczy, spełnione są w chwili τ_x następujące warunki:

C1. $\mathcal{PD}^* \neq \varnothing$ i

C2. $(\forall P_i :: P_i \in \mathcal{PD}^* :: passive_i[\tau_x])$ i

C3. $(\forall P_i :: P_i \in \mathcal{PD}^* :: \neg activate_i(\mathcal{IT}_i[\tau_x] \cup \mathcal{AV}_i[\tau_x] \cup \mathcal{P} \setminus \mathcal{PD}^*))$.

Pokażemy teraz, że powyższe warunki są spełnione, jeżeli algorytm zakończy się z $\mathcal{PD}^* \neq \varnothing$. Z definicji warunku zakończenia wynika wprost, że jeżeli algorytm kończy się z $\mathcal{PD}^* \neq \varnothing$, to warunek C1 jest spełniony w chwili τ_x.

Niech teraz $P_i \in \mathcal{PD}^*$. Z konstrukcji algorytmu wynika, że tylko proces stale pasywny począwszy od zakończenia pierwszego cyklu w chwili τ_i^1, może należeć do $\mathcal{PD}^*$. Stąd mamy:

$$contPassive^*_i(\tau_i^1, \tau_i^{k+1}), \text{ a więc } passive_i[\tau_x]$$

Tym samym warunek C2 zajścia zakleszczenia został zweryfikowany. Ponieważ $P_i \in \mathcal{PD}^*$, wnioskujemy z konstrukcji algorytmu , że:

$$\neg activate_i(\mathcal{AV}_i[\tau_i^k] \cup \mathcal{P} \setminus \mathcal{PD}^*) = True.$$

W przeciwnym bowiem razie, P_i zostałby usunięty ze zbioru $\mathcal{PD}^*$. Ale, skoro $\mathcal{AV}_i$ może tylko zmienić się przez dołączanie nowych elementów, a $\mathcal{P} \setminus \mathcal{PD}^*$ nie zmienia się w przedziale $\langle \tau_\alpha^k, \tau_\alpha^{k+1} \rangle$, otrzymujemy:

$$(\mathcal{AV}_i[\tau_x] \cup \mathcal{P} \setminus \mathcal{PD}^*) \subseteq \mathcal{AV}_i[\tau_i^{k+1}] \cup \mathcal{P} \setminus \mathcal{PD}^*)$$

Z konstrukcji algorytmu wnioskujemy dalej, że wszystkie kanały wyjściowe procesów należących do $\mathcal{PD}^*$ są puste w chwili τ_α^k, gdyż znacznik jest zatrzymywany przez monitor skojarzony z pasywnym procesem należącym do $\mathcal{PD}^*$ do momentu uzyskania potwierdzeń dla wszystkich wysłanych wcześniej wiadomości ($notAck_i = 0$).

Ponieważ procesy te są przy tym stale pasywne od pierwszej wizyty znacznika, ich kanały wyjściowe są puste w chwili τ_x. Stąd:

$$(\mathcal{IT}_i[\tau_x] \cap \mathcal{PD}^*) = 0 \text{ i dlatego } \mathcal{IT}_i[\tau_x] \in \mathcal{P} \setminus \mathcal{PD}^*.$$

W konsekwencji:

$$(\mathcal{IT}_i[\tau_x] \cup \mathcal{AV}_i[\tau_x] \cup \mathcal{P} \setminus \mathcal{PD}^*) = (\mathcal{AV}_i[\tau_x] \cup \mathcal{P} \setminus \mathcal{PD}^*)$$

Z zależności i otrzymujemy:

$$(\mathcal{IT}_i[\tau_x] \cup \mathcal{AV}_i[\tau_x] \cup \mathcal{P} \setminus \mathcal{PD}^*) \subseteq (\mathcal{AV}_i[\tau_i^{k+1}] \cup \mathcal{P} \setminus \mathcal{PD}^*)$$

Z kolei z monotoniczności predykatu $activate_i$ oraz z powyższego równania wnioskujemy, że jeżeli zachodzi:

$$\neg activate_i(\mathcal{AV}_i[\tau_i^{k+1}] \cup \mathcal{P} \setminus \mathcal{PD}^*)$$

to zachodzi również

$$\neg activate_i(\mathcal{IT}_i[\tau_x] \cup \mathcal{AV}_i[\tau_x] \cup \mathcal{P} \setminus \mathcal{PD}^*)$$

Z konstrukcji wnosimy dalej, że predykat jest spełniony dla wszystkich $P_i \in \mathcal{PD}^*$. Tym samym warunek C3 został wykazany.

Powyższe punkty dowodzą, że predykat $deadlock(\mathcal{PD}^*)$ zachodzi w chwili τ_x. Ponieważ jak wiadomo, predykat ten jest stabilny i $\tau_x \leqslant \tau_e$, $deadlock(\mathcal{PD}^*)$ zachodzi również w chwili τ.

Dowodzi to pierwszego stwierdzenia implikacji b). Zauważmy na koniec, że jeżeli istnieje zbiór $\mathcal{B}$, dla którego $deadlock(\mathcal{B})[\tau_b]=True$, to z własności i wnioskujemy, że $\mathcal{B} \in \mathcal{P} \setminus \mathcal{PD}^*$. Tym samym drugie stwierdzenie implikacji b) zostało udowodnione. □