Problem detekcji zakończenia (II)

Plan wykładu

Celem obecnego wykładu jest przedstawienie przeglądu algorytmów poświęconych tematyce detekcji zakończenia zapoczątkowanego w poprzednim module. Wykład obejmie omówienie następujących algorytmów detekcji zakończenia dla atomowego modelu przetwarzania: algorytm z wykorzystaniem liczników wiadomości, jednofazowy algorytm detekcji zakończenia, wektorowy algorytm detekcji zakończenia. Przedstawione zostaną również algorytmy detekcji zakończenia statycznego oraz dynamicznego. Zostaną również pokazane dowody poprawności tych algorytmów oraz uwagi na temat ich złożoności obliczeniowej.

Atomowy model przetwarzania

W *atomowym modelu przetwarzania,* czasy przetwarzania lokalnego są zaniedbywalne (zerowe), a opóźnienia są związane tylko z transmisją.

Można przyjąć, że procesy (uaktywniane natychmiastowo) są przez cały czas pasywne i sprowadzić problem detekcji zakończenia do wyznaczania stanów kanałów. Przyjmijmy dla uproszczenia, model żądań typu OR. Oznacza to, że dowolna wiadomość w kanale uaktywni proces.

Wówczas zakończenie przetwarzania opisuje następujący predykat:

$$Aterm(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (\mathcal{IT}_i = \varnothing)$$

Detekcja zakończenia liczniki wiadomości (1)

Rozważmy dla modelu atomowego rozwiązanie problemu detekcji zakończenia przetwarzania rozproszonego z zastosowaniem liczników wiadomości zaproponowane przez Matterna:

- $sc_i(\tau)$ – określa liczbę wiadomości wysłanych do chwili τ przez P_i
- $rc_i(\tau)$ – określa liczbę wiadomości odebranych do chwili τ przez P_i
- $SC(\tau)$ – sumaryczna liczba wiadomości wysłanych przez wszystkie procesy przetwarzania rozproszonego do chwili τ
- $RC(\tau)$ – sumaryczna liczba wiadomości wysłanych przez wszystkie procesy przetwarzania rozproszonego do chwili τ

Detekcja zakończenia liczniki wiadomości (2)

Dla modelu atomowego, równość $SC(\tau) = RC(\tau)$ oznacza, że każda wiadomość wysłana została odebrana. Tym samym wszystkie kanały są w chwili τ puste, a więc osiągnięty został stan zakończenia. Problem polega na tym, że wyznaczenie $SC(\tau)$ i $RC(\tau)$ nie jest w systemie rozproszonym proste.

Detekcja zakończenia liczniki wiadomości (3)

W swej idei, algorytmy detekcji z użyciem liczników zakończenia wykorzystują koncepcję ciągu cykli detekcyjnych. W tym jednak wypadku celem wysłania wiadomości kontrolnej (znacznika) do wszystkich monitorów jest wyznaczenie sumarycznej liczby wiadomości wysłanych i odebranych przez wszystkie procesy aplikacyjne. Dlatego wiadomość kontrolna wysłana przez inicjatora Q_α przesyłana jest między monitorami, a po dotarciu do wszystkich monitorów wraca do Q_α.

Detekcja zakończenia liczniki wiadomości (4)

Każdy z monitorów Q_i dodaje do znacznika stany liczników sc_i i rc_i odpowiadające bieżącej chwili τ_i. Ponieważ kanały wprowadzają opóźnienia, zebrane liczniki odpowiadają więc różnym momentom τ_i, $1 \leq i \leq n$.

Detekcja zakończenia liczniki wiadomości (5)

Oznaczmy przez SC^* i RC^* odpowiednio sumy zebranych przez znacznik liczników, a więc:

$$SC^* = sc_i(\tau_i), \qquad RC^* = rc_i(\tau_i),$$

pamiętając, że:

$$SC(\tau) = sc_i(\tau), \; RC(\tau) = rc_i(\tau),$$

Przykłady wyznaczania liczników *RC i *SC****

Dla ilustracji przedstawionej koncepcji rozważmy obecnie dwa przykłady przedstawione na slajdzie. Na przykładzie po lewej stronie slajdu, proces detekcji rozpoczynany jest w chwili $\tau_b = \tau_4$ przez monitor $Q_\alpha = Q_4$ procesu P_4, W chwili tej $sc_4(\tau_4)=0$ oraz $rc_4(\tau_4) = 0$. Znacznik jest przesyłany następnie do monitora Q_3 procesu P_3, który odczytuje wartości $sc_3(\tau_3) = 1$ oraz $rc_3(\tau_3) = 0$. Następnie, kolejno wyznaczane są wartości: $sc_2(\tau_2) = 2$, $rc_2(\tau_2) = 1$, $sc_1(\tau_1) = 1$, $rc_1(\tau_1) = 1$. W efekcie w chwili τ_e otrzymujemy $SC^* = 4$ a $RC^* = 2$.

Z kolei sytuacja na przykładzie po prawej stronie slajdu jest następująca: $sc_1(\tau_1) = 0$, $rc_1(\tau_1) = 1$, $sc_2(\tau_2) = 0$, $rc_2(\tau_2) = 0$, $sc_3(\tau_3) = 1$, $rc_2(\tau_2) = 0$, $sc_4(\tau_4) = 0$, $rc_4(\tau_4) = 0$. Tym samym $SC^* = 1$ a $RC^* = 1$.

Równość liczników wiadomości wysłanych i odebranych mogłoby sugerować, że wszystkie kanały są puste, i że tym samym wystąpiło zakończenie. Zauważmy jednak, że w żadnym momencie τ z przedziału obserwacji $\langle \tau_b, \tau_e \rangle$ kanały nie są jednocześnie puste, a więc $SC(\tau) \neq RC(\tau)$ dla każdego τ, $\tau \in \langle \tau_b, \tau_e \rangle$. Powstaje zatem problem, czy i kiedy na podstawie faktu, że $SC^* = RC^*$ można wnioskować o wystąpieniu zakończenia.

Poprawność algorytmu detekcji zakończenia dla atomowego modelu przetwarzania

W algorytmie detekcji zakończenia rozważane są dwie fazy detekcji. Pierwsza rozpoczyna się o chwili τ_b^1 i kończy w chwili τ_e^1, a druga odpowiednio w chwilach τ_b^2 i τ_e^2. Przyjęto ponadto, że: $\tau_b^1 < \tau_e^1 < \tau_b^2 < \tau_e^2$.

Oznaczmy teraz przez SC^* i RC^* liczniki wyznaczone w pierwszej fazie, a przez SC^{**} i RC^{**} liczniki oznaczone w drugiej fazie.

Detekcja zakończenia dla atomowego modelu przetwarzania - twierdzenie 10.1

Twierdzenie 10.1

Jeżeli

$$RC^* = SC^* = RC^{**} = SC^{**}$$

to monitorowane przetwarzanie aplikacyjne osiągnęło stan zakończenia przed zakończeniem procesu detekcji

Dowód

W celu udowodnienia powyższego twierdzenia wykażemy, że jeżeli $RC^* = SC^{**}$, to przetwarzanie jest w stanie zakończenia. W tym wykażemy w dalszym ciągu prawdziwość pięciu lematów.

Lematy 10.1.1, 10.1.2

Lemat 10.1.1

Lokalne liczniki wiadomości są monotoniczne. Tak więc, jeżeli $\tau \leq \tau'$, to $sc_i(\tau) \leq sc_i(\tau')$ i $rc_i(\tau) \leq rc_i(\tau')$.

Dowód
Dowód tego lematu wynika wprost z definicji liczników rc_i i sc_i.

Lemat 10.1.2

Sumaryczna liczba wiadomości wysłanych lub odebranych jest monotoniczna. Jeżeli zatem $\tau \leq \tau'$, to $SC(\tau) \leq SC(\tau')$ i $RC(\tau) \leq RC(\tau')$.

Dowód

Dowód wynika wprost z definicji SC i RC.

Lemat 10.1.3 oraz 10.1.4

Lemat 10.1.3

$$RC^* \leq RC(\tau_e^1)$$

Lemat 10.1.4

$$SC^{**} \leq SC(\tau_b^2)$$

Dowód lematu 10.1.3

Dowód

Licznik RC^* z definicją jest sumą zebranych liczników $rc_i(\tau_i)$, przy czym dla każdego $i \in \{1, 2, \ldots, n\}$, $\tau_b^1 \leq \tau_i \leq \tau_e^1$.

Z kolei, $RC(\tau_e^1)$ jest sumą liczników $rc_i(\tau_e^1)$

Dowód lematu 10.1.3 (2)

Ponieważ liczniki są monotoniczne i dla każdego $i \in \{1, 2, \ldots, n\}$: $\tau_i \leq \tau_e^1$, więc tym samym $rc_i(\tau_e^1) \leq rc_i(\tau_i)$ dla każdego *i*, a w konsekwencji:

$$RC^* \leq RC(\tau_e^1)$$

Dowód lematu 10.1.4

Dowód

Z definicji, SC^{**} jest sumą liczników $sc_i(\tau_i)$, przy czym dla każdego $i \in \{1,2, \ldots, n\}$, $\tau_b^2 \leq \tau_i \leq \tau_e^2$.

Z kolei, $SC(\tau_b^2)$ jest sumą liczników $sc_i(\tau_b^2)$.

Dowód lematu 10.1.4 (2)

Ponieważ liczniki są monotoniczne i dla każdego i, $i \in \{1,2, \ldots ,n\}$, $\tau_b^2 \leq \tau_i$, więc tym samym $sc_i(\tau_b^2) \leq sc_i(\tau_i)$ dla każdego i, a w konsekwencji:

$$SC(\tau_b^2) \leq SC^{**}.$$

Lemat 10.1.5

Lemat 10.1.5

Dla wszystkich τ: $RC(\tau) \leq \quad SC(\tau)$

Dowód
Relacja jest prawdziwa, gdyż kanały są z założenia niezawodne.

Dowód twierdzenia 10.1

Dowód
Z założenia wynika, że:

(1) $RC^* = SC^{**}$

Z kolei z przedstawionych lematów otrzymujemy, że:

- $SC^{**} \geq SC(\tau_b^2)$ (lemat 10.1.4)
- $SC(\tau_b^2) \geq SC(\tau_e^1)$ (lemat 10.1.2)
- $SC(\tau_e^1) \geq RC(\tau_e^1)$ (lemat 10.1.5)
- $RC(\tau_e^1) \geq RC^*$ (lemat 10.1.3)

Dowód twierdzenia 10.1 (2)

Relacje (1) i (2) zachodzą jednocześnie tylko wówczas, gdy:

$$SC^{**} = SC(\tau_b^2) = SC(\tau_e^1) = RC(\tau_e^1) = RC^*$$

Tym samym w chwili τ_e^1, $SC(\tau_e^1) = RC(\tau_e^1)$ a to oznacza, że wszystkie kanały są puste, a więc wystąpiło zakończenie przetwarzania.

Jednofazowy algorytm detekcji zakończenia: założenia

Wadą poprzedniego rozwiązania jest konieczność dwukrotnego przesyłania wiadomości kontrolnej do wszystkich procesów. Istnieją propozycje realizacji detekcji zakończenia w jednym przebiegu, czyli *algorytmy jednofazowe*. Załóżmy, że monitory Q_i: $i \in \{1,2, \ldots , n\}$ połączone są w logiczny pierścień.

Ideą rozwiązania jest przesyłanie wiadomości kontrolnej (znacznika) typu TOKEN między kolejnymi monitorami, zawierającego informacje ewentualnie wystarczające inicjatorowi do podjęcia decyzji o wykryciu zakończenia.

W algorytmie detekcji zakończenia przetwarzania rozproszonego przesyłana jest jest wiadomość kontrolna (znacznik) typu TOKEN między kolejnymi monitorami. Wiadomość ta zawiera:

- identyfikator inicjatora Q_α – *initId*
- numer sekwencyjny *detectNo* cyklu detekcji zainicjowanego przez Q_α,
- Pole *SRAccu,* sumy liczników $SRBalance_i$ monitorów Q_i odwiedzonych już przez znacznik

- pole flagi niepoprawności procesu detekcji *invalid*. Flaga ta przyjmuje ostatecznie wartość *True*, jeżeli którykolwiek proces aplikacyjny otrzymał między kolejnymi cyklami detekcji wiadomość, która narusza warunek konieczny poprawności detekcji: $maxDetectNo_i \geq tokenIn.detectNo$, gdzie liczba $maxDetectNo_i$ oznacza numer sekwencyjny cyklu detekcji skojarzony z wiadomością wysłaną najpóźniej, spośród wszystkich wiadomości odebranych przez Q_i

Z kolei wiadomości aplikacyjne typu PACKET zawierają dodatkowo pole *detectNo*, zawierające numer sekwencyjny cyklu detekcji.

Jednofazowy algorytm detekcji zakończenia (2)

Algorytm wykorzystuje następujące zmienne:

- $SRBalance_i$ – lokalne liczniki (ang. *send-receive balance*) o początkowej wartości 0. Wartość licznika $SRBalance_i$ w każdej chwili jest równa $sc_i - rc_i$.
- $detectNo_\alpha$ – numer sekwencyjny cyklu detekcji zakończenia, zainicjowanego przez monitor Q_α
- $maxDetectNo_i$ – zmienna określającą numer sekwencyjny cyklu detekcji skojarzony z wiadomością wysłaną najpóźniej, spośród wszystkich wiadomości odebranych przez Q_i.

Wiadomość *msgIn* oznacza wiadomość aplikacyjną wysyłaną przez proces P_i, która jest umieszczana w polu *data* pakietu *pcktOut*. Zmienna $terminationDetected_i$ zostaje ustawiona na *True* jeżeli wykryte zostało zakończenie.

Jednofazowy algorytm detekcji zakończenia (3)

Inicjator detekcji Q_α inkrementuje licznik cyklu detekcji $detectNo_\alpha$ oraz wysyła znacznik do swojego następnika w pierścieniu. Początkowe wartości pól *initId*, *detectNo*, *SRAccu* oraz *invalid* w znaczniku są ustawiane na, kolejno, identyfikator monitora Q_α, numer cyklu detekcji, sumę liczników oraz *False*.

Jednofazowy algorytm detekcji zakończenia (4)

Przesyłając wiadomość aplikacyjną monitor Q_i inkrementuje zmienną $SRBalance_i$ oraz przypisuje jej numer cyklu detekcji zakończenia.

Jednofazowy algorytm detekcji zakończenia (5)

Otrzymując wiadomość aplikacyjną monitor Q_i dekrementuje licznik $SRBalance_i$. Jeżeli wartość numer cyklu detekcji przesłana w otrzymanej wiadomości aplikacyjnej jest większa niż bieżąca wartość $maxDetectNo_i$, to zmiennej $maxDetectNo_i$ przypisana zostaje właśnie ta wartość.

Jednofazowy algorytm detekcji zakończenia (6)

W każdym odwiedzanym monitorze Q_i uaktualniany jest numer sekwencyjny bieżącego cyklu detekcji. Jeżeli $Q_i = Q_\alpha$, to inicjator decyduje, że wykryte zostało zakończenie, jeżeli wartość pola *SRAccu* jest równa zero i równocześnie flaga *invalid* nie ma przypisanej wartości *True*. Jeżeli znacznik powrócił do inicjatora, i nie może on podjąć decyzji o zakończeniu detekcji, to rozpoczyna on nowy cykl detekcyjny.

Jednofazowy algorytm detekcji zakończenia (7)

Dla monitorów $Q_i \neq Q_\alpha$, pola znacznika są aktualizowane w następujący sposób: do pola znacznika *SRAccu* dodawana jest aktualna wartość zmiennej $SRBalance_i$; fladze *invalid* przypisywana jest suma logiczna bieżącej wartości tej flagi oraz wartości relacji $maxDetectNo_i \geq tokenIn.detectNo$. Ta ostatnia relacja jest prawdziwa, jeżeli do monitora dotarł już pewien pakiet *pckt* z etykietą *pckt.detectNo* o wartości większej lub równej niż numer sekwencyjny bieżącego cyklu detekcji $tokenOut.detectNo$. Oznacza to, że pakiet ten został wysłany po wizycie bieżącego znacznika. Zauważmy, że zdarzenie odbioru pakietu *pckt* będzie uwzględnione w końcowej wartości licznika *SRAccu*, a nie będzie uwzględnione zdarzenie nadania tego pakietu. Tym samym, konkluzja dotycząca zakończenia

przetwarzania byłaby formułowana na podstawie niespójnego obrazy przetwarzania, a więc w ogólności nie byłaby poprawna. W takim wypadku, wynik bieżącego cyklu detekcji należy zatem uznać za niepoprawny. W efekcie, wykrycie zakończenia wymaga wykonania kolejnego cyklu detekcji.

Wektorowy algorytm detekcji zakończenia: założenia

Przedstawiony zostanie obecnie algorytm wykorzystujący wektor liczników $vSRNo_i$ będący tablicą [1..*n*]. Przyjęto przy tym, że procesy nie komunikują się same ze sobą, a monitory tworzą strukturę o topologii pierścienia.

Wektorowy algorytm detekcji zakończenia (1)

Cyrkulujący w pierścieniu znacznik typu TOKEN przenosi *n*-elementowy wektor $vSRAccu$ zawierający sumę wektorów $vSRNo_i$ monitorów odwiedzonych już w danym cyklu detekcji przez TOKEN.

Wektorowy algorytm detekcji zakończenia (2)

Jak już wspomniano, w algorytmie wektorowym wykorzystywany jest wektor liczników $vSRNo_i$ będący tablicą [1..*n*]. Elementy $vSRNo_i[j]$, dla $i \neq j$, każdego procesu P_i określają liczbę wiadomości wysłanych przez proces P_i do P_j od czasu ostatniej wizyty znacznika.

Wartość bezwzględna elementu $vSRNo_i[i]$ określa liczbę wiadomości odebranych przez P_i od czasu ostatniej wizyty znacznika.

W dowolnej chwili τ, suma *k*-tych elementów wszystkich liczników $vSRNo_i$, dla każdego $i \in \{1, 2, ..., n\}$, oraz wartości pozycji *k* wektora cyrkulującego znacznika $tokenIn.vSRAccu[k]$, jest równa liczbie wiadomości będących w drodze do P_k. Zmienna $firstWave_i$ przyjmuje wartość $True$, jeżeli monitor Q_i pierwszy raz otrzymał znacznik. Wreszcie zmienna $terminationDetected_i$ posiada zwykłe znaczenie.

Wektorowy algorytm detekcji zakończenia (3)

Inicjator Q_α rozpoczyna detekcję wysyłając znacznik typu TOKEN z wektorem $vSRAccu = 0$ oraz ustawiając wartość zmiennej $firstWave_i$ na $False$.

Wektorowy algorytm detekcji zakończenia (4)

Po otrzymaniu znacznika monitor Q_i sumuje własny wektor wiadomości $vSRNo$, z wektorem $vSRAccu$ zawartym w znaczniku. Warunek $vSRAccu_i[i] > 0$ oznacza, że istnieje wiadomość aplikacyjna uwzględniona już w znaczniku $vSRAccu$ jako wysłana do P_i, a która nie została jeszcze odebrana przez P_i. W tym wypadku znacznik jest *zatrzymywany* w monitorze Q_i (nie jest dalej wysyłany), a jego dalsze przesłanie jest uwarunkowane otrzymaniem odpowiedniej liczby wiadomości aplikacyjnych przez P_i.

Wektorowy algorytm detekcji zakończenia (5)

Każdy monitor Q_i nalicza indywidualnie wiadomości aplikacyjne wysłane przez P_i, zwiększając licznik $vSRNo_i[j]$ na pozycji odpowiadającej procesowi przeznaczenia P_j.

Wektorowy algorytm detekcji zakończenia (6)

Z drugiej strony monitor Q_i zmniejsza licznik własny $vSRNo_i[i]$, gdy odebrana zostanie wiadomość aplikacyjna.

Jeżeli $vSRNo_i[i]$ jest równe 0, to jeżeli dla każdej pozycji wektora liczników jest ona równa zero i flaga $firstWave_i$ nie jest ustawiona, to zostaje podjęta decyzja o wykryciu zakończenia. W przeciwnym wypadku rozpoczynany jest kolejny cykl detekcyjny.

Wektorowy algorytm detekcji zakończenia: cechy

Ważnym charakterystykami tego algorytmu są:

- wyeliminowanie konieczności dołączania etykiet do wiadomości aplikacyjnych

- zatrzymywanie znacznika do momentu spełnienia koniecznych warunków zakończenia
- wykonywanie algorytmu w sposób ciągły aż do momentu stwierdzenia zakończenia przetwarzania aplikacyjnego

Algorytm ten kończy się w ciągu jednej rundy wymiany znacznika od momentu faktycznego zakończenia przetwarzania aplikacyjnego. Dużą jego wadą jest spora liczba informacji przesyłanych w znaczniku.

Algorytm detekcji zakończenia statycznego: koncepcja

Obecnie przedstawiony algorytm zajmuje się problemem wykrycia faktu zakończenia statycznego. W algorytmie tym wykorzystano koncepcję ciągu cykli detekcyjnych, gdzie monitory są logicznie połączone w strukturę topologiczną gwiazdy.

Algorytm detekcji zakończenia statycznego (1)

Algorytm wykorzystuje cztery typy komunikatów. Typ PACKET służy do przesyłania wiadomości aplikacyjnych. Typ QUERY jest zapytaniem wysyłanym przez inicjatora detekcji. Typ ACK służy do potwierdzania wiadomości aplikacyjnych, i wreszcie typ REPLY jest odpowiedzią na wiadomości QUERY. Ten ostatni typ zawiera pole *contPassive* które jest flagą zawierającą wartość zmiennej $contPassive_i$ nadawcy Q_i wiadomości REPLY, w chwili jej wysłania.

Algorytm detekcji zakończenia statycznego (2)

W przedstawionym algorytmie *msgIn* oraz *pcktOut* posiadają swoje zwykłe znaczenie. Komunikat *queryOut* jest komunikatem typu QUERY, komunikaty *replyIn* oraz *replyOut* są typu REPLY, a *ackOut* typu ACK. Aby uniknąć pominięcia wiadomości aplikacyjnej, która została wysłana przed otrzymaniem zapytania QUERY przez pewien monitor Q_i, a odebrana po otrzymaniu zapytania przez inny monitor Q_k – wprowadzona jest zmienna logiczna $contPassive_i$, która jest inicjowana wartością *True*. Za każdym razem, gdy P_i jest uaktywniany, $contPassive_i$ przyjmuje wartość *False*. Z kolei wysyłając wiadomość kontrolną REPLY monitory nadają zmiennej $contPassive_i$ ponownie wartość *True*. Wartość *True* zmiennej $contPassive_i$ w danej chwili oznacza, że proces P_i był pasywny przez cały czas od chwili wysłania ostatniej wiadomości REPLY.

Jeżeli zmienna $sTermDetected_i$ przyjmuje wartość *True*, to monitor Q_α stwierdza decyduje o wykryciu zakończenia statycznego, przypisując wartość *True* zmiennej $terminationDetected_i$. Zmienna $notAck_i$ jest licznikiem niepotwierdzonych wiadomości. Zbiór $\mathcal{AV}_i$ zawiera identyfikatory procesów, od których wiadomości aplikacyjne są już dostępne dla P_i.

Algorytm detekcji zakończenia statycznego (3)

W celu stwierdzenia zakończenia statycznego przetwarzania aplikacyjnego określonego przez predykat

$$Sterm(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (passive_i \wedge (\mathcal{IT}_i \neq \emptyset) \wedge \neg\, activate_i(\mathcal{AV}_i))$$

inicjator Q_α rozpoczyna detekcję przez wysłanie do wszystkich monitorów, w tym do siebie, wiadomości kontrolnej typu QUERY.

Po otrzymaniu wszystkich odpowiedzi, inicjator wyznacza iloczyn logiczny otrzymanych zmiennych *contPassive* jako wartość zmiennej $sTermDetected_\alpha$.

- Jeżeli $sTermDetected_\alpha$ przyjmuje wartość *True*, to Q_α stwierdza zakończenie
- W przeciwnym przypadku inicjator wysyła ponownie zapytanie QUERY

Algorytm detekcji zakończenia statycznego (4)

Po otrzymaniu zapytania QUERY w odpowiedzi monitory przesyłają wiadomości typu REPLY natychmiast gdy skojarzony z nimi proces aplikacyjny staje się pasywny, licznik niepotwierdzonych wiadomości wynosi zero i żadna z dostępnych w danej chwili wiadomości nie wystarcza do aktywacji P_i. Wysyłana wiadomość zawiera bieżącą wartość zmiennej logicznej określającej, czy od czasu wysłania ostatniej odpowiedzi QUERY proces wciąż pozostawał pasywny, w polu $contPassive$.

Algorytm detekcji zakończenia statycznego (5)

Wysyłając wiadomości inkrementowany jest licznik $notAck_i$. Licznik ten jest zmniejszany w wyniku otrzymania potwierdzenia wiadomości.

Algorytm detekcji zakończenia statycznego (6)

Otrzymując wiadomość aplikacyjną monitor Q_i wysyła potwierdzenie do jej nadawcy.

W momencie aktywacji procesu P_i zmiennej $contPassive_i$ przypisywana jest wartość $False$.

Algorytm detekcji zakończenia statycznego: dowód poprawności – oznaczenia

Twierdzenie o poprawności przedstawionego algorytmu używać będzie następujących oznaczeń:

τ_i^k – czas globalny wysłania odpowiedzi przez monitor Q_i w *k*-tym cyklu detekcji.

τ_b^k – czas globalny rozpoczęcia *k*-tego cyklu detekcji

τ_e^k – czas globalny zakończenia *k*-tego cyklu detekcji

Zachodzi:

$$\tau_b^k < \tau_i^k < \tau_e^k$$

Algorytm detekcji zakończenia statycznego: dowód poprawności – oznaczenia (2)

$\mathcal{X}[\tau_i^k]$ – jest wartością zmiennej (predykatu) w chwili τ_i^k

Q_α – dla ułatwienia zostanie przyjęte, że inicjator jest dodatkowym procesem, nie związanym z procesami aplikacyjnymi

Twierdzenie 10.2

Twierdzenie 10.2

Jeżeli algorytm detekcji zakończenia statycznego jest wykonywany w chwili, gdy przetwarzanie aplikacyjne osiągnęło stan zakończenia statycznego, to algorytm ten zakończy się w skończonym czasie, stwierdzając wystąpienie zakończenia statycznego przetwarzania aplikacyjnego ($sTerminationDetected_\alpha = True$).

Dowód

Jeżeli przetwarzanie aplikacyjne osiągnęło stan zakończenia w chwili τ^t, to zgodnie z definicją, od tego momentu wszystkie procesy będą zawsze pasywne ($passive_i = True$), ich warunki uaktywnienia nie będą spełnione ($activate_i(\mathcal{AV}_i) = False$), a ponadto w kanałach nie będzie znajdowała się żadna wiadomość aplikacyjna ($\mathcal{IT}_i = \varnothing$).

Stąd, w skończonym czasie zostaną w systemie odebrane potwierdzenia od monitorów, i liczniki $notAck_i$ przyjmą wartość 0. Oznaczmy tę chwilę przez τ^x. Oczywiście $\tau^x > \tau^t$. Od momentu τ^x bieżący cykl detekcyjny nie będzie z pewnością wstrzymywany przez monitory.

Zauważmy jednak, że pewne monitory w tym cyklu mogły wysłać odpowiedzi REPLY przed momentem τ^t. Stąd, dopiero następny cykl detekcyjny z pewnością przypisze już na stałe wszystkim zmiennym $contPassive_i$ wartość $True$.

W konsekwencji, przez cały kolejny cykl detekcyjny k+1, wszystkie zmienne logiczne $contPassive_i$ będą miały wartość $True$ dla każdego $i \in \{1,2,...n\}$. Po odebraniu zatem wiadomości REPLY i przyjęciu przez $sTerminationDetected_\alpha$ wartości $True$, algorytm zakończy się.

Twierdzenie 10.3

Twierdzenie 10.3

Jeżeli algorytm stwierdza wystąpienie zakończenia statycznego przetwarzania aplikacyjnego, to przetwarzanie aplikacyjne jest w istocie w stanie zakończenia statycznego.

Rozważmy dwa kolejne cykle detekcyjne k oraz k+1. Załóżmy, że algorytm stwierdza zakończenie przetwarzania aplikacyjnego kończąc cykl k+1, a więc w chwili τ_e^{k+1}. Wówczas $sTerminationDetected_\alpha$ = $True$.

Szkic dowodu twierdzenia 10.3

Ponieważ cykle są inicjowane sekwencyjnie, istnieje taki moment τ^x, że:

$$\tau_b^k < \tau_e^k \leq \tau^x \leq \tau_b^{k+1} < \tau_e^{k+1}$$

Udowodnimy, że jeżeli $sTerminationDetected_\alpha[\tau_e^{k+1}] = True$, to w chwili $\tau = \tau^x$ przetwarzanie aplikacyjne jest statycznie zakończone, a więc:

$$Sterm(\mathcal{P})[\tau^x] = True.$$

Innymi słowy wykażemy, że dla wszystkich procesów spełnione będą następujące warunki:

C1. $passive_i[\tau^x] = True$ (proces jest pasywny w chwili τ^x)

C2. $\mathcal{IT}_i[\tau^x] = \varnothing$. (nie ma żadnych procesów, od których wiadomości jeszcze nie dotarły do P_i)

C3. $activate(\mathcal{AV}_i)[\tau^x] = False.$ (dostępne wiadomości nie wystarczają do uaktywnienia procesu)

Dowód warunku C1

Z konstrukcji algorytmu wynika, że zakończy się on w chwili τ_e^{k+1} z $sTerminationDetected[\tau_e^{k+1}] = True$, tylko wówczas, gdy każdy proces P_i był pasywny w czasie między τ_i^k a τ_i^{k+1}. Skoro jednak $\tau_i^k \leq \tau^x \leq \tau_i^{k+1}$, więc możemy zatem wnosić, że dla każdego $P_i \in \mathcal{P}$, $passive_i[\tau^x] = True.$

Dowód warunku C2

Zgodnie z konstrukcją algorytmu, w każdym cyklu detekcyjnym odpowiedź REPLY jest wstrzymywana aż do momentu nadejścia potwierdzeń od monitorów procesów przeznaczenia, dla wszystkich wysłanych wcześniej wiadomości aplikacyjnych.

Wówczas $notAck_i$, przyjmie wartość 0. Z drugiej strony można jednak zauważyć, że od chwili τ_i^k do τ_i^{k+1} proces P_i jest ciągle pasywny, a więc nie wysyła wiadomości. Stąd więc wnosimy, że dla każdego procesu $P_i \in \mathcal{P}$, $\mathcal{IT}_i[\tau^x] = \varnothing$.

Dowód warunku C3 (1)

Z konstrukcji algorytmu wynika, że w każdym cyklu detekcyjnym odpowiedź jest opóźniana do momentu uzyskania przez predykat $\neg activate(\mathcal{AV}_i)$ wartości $True$. Skoro algorytm kończy się w chwili τ_e^{k+1}, to predykat $activate(\mathcal{AV}_i)$ $[\tau_i^{k+1}]$ miał wartość $False$. Ponieważ w przedziale $\langle \tau_i^k, \tau_i^{k+1} \rangle$ proces P_i był przez cały czas pasywny, żadna wiadomość nie została w tym czasie odebrana.

Dowód warunku C3 (2)

Stąd też, w przedziale tym do zbioru $\mathcal{AV}_i$ mogły być co najwyżej dołączone dodatkowe wiadomości. W efekcie:

$$\mathcal{AV}_i[\tau^x] \subseteq \mathcal{AV}_i[\tau_i^{k+1}]$$

Z własności monotoniczności predykatu $activate_i$ możemy przy tym wnioskować, że skoro zachodzi $\neg activate_i(\mathcal{AV}_i)[\tau_i^{k+1}]$ to również zachodzi $\neg activate(\mathcal{AV}_i)[\tau^x]$.

Dowód twierdzenia 10.3

Wykazaliśmy zatem, że w chwili τ^x, dla każdego procesu $P_i \in \mathcal{P}$ spełnione są warunki C1, C2 i C3, a więc predykat zakończenia statycznego w chwili τ^x jest prawdziwy ($Sterm(\mathcal{P})[\tau^x] = True$). Ponieważ predykat ten jest stabilny, więc również dla każdego $\tau > \tau^x$, w tym również dla τ_e^{k+1}, $Sterm(\mathcal{P})[\tau^x]=True$.

Algorytm detekcji zakończenia statycznego: Złożoność obliczeniowa

Efektywność, a także złożoność czasowa i komunikacyjna, algorytmu zależy od implementacji cykli detekcyjnych i topologii przetwarzania monitorującego.

W celu wyznaczenia złożoności czasowej zauważmy, że po wystąpieniu zakończenia statycznego, dla jego stwierdzenia potrzebne są w najgorszym przypadku dwa pełne cykle detekcyjne oraz musi być dokończony cykl bieżący.

W każdym cyklu przesyłanych jest n wiadomości typu QUERY i n wiadomości typu REPLY przenoszących jednobitową zmienną. Tak więc zaniedbując wiadomości potwierdzeń (których liczba jest równa liczbie wiadomości aplikacyjnych), otrzymujemy liczbę wiadomości kontrolnych wystarczających do stwierdzenia zakończenia, równą: $3 \times 2n = 6n$.

Przyjmijmy teraz, że struktura topologiczna przetwarzania detekcyjnego jest grafem w pełni połączonym. Pomijając ponownie wiadomości potwierdzeń, łatwo już zauważyć, że złożoność czasowa algorytmu wynosi 3. Oczywiście przy innych topologiach złożoność ta zmienia się odpowiednio. Przykładowo, w przypadku pierścienia złożoność czasowa wyniesie $3n$.

Detekcja zakończenia dynamicznego

Detekcja zakończenia dynamicznego sprowadza się do wyznaczenia wartości predykatów definiujących zakończenie:

$$Dterm(\mathcal{P}) \equiv \forall P_i :: P_i \in \mathcal{P} :: (passive_i \wedge \neg\, activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i))$$

W tym celu zaproponowano algorytm, który stosuje mechanizm cykli detekcyjnych oraz wiadomości kontrolne typu QUERY i REPLY, a także liczniki wiadomości odebranych i wysłanych przez procesy.

Detekcja zakończenia dynamicznego: Algorytm (1)

Algorytm wykorzystuje trzy typy komunikatów, z których PACKET służy do przenoszenia wiadomości aplikacyjnych.

Typ QUERY zawiera pole *vSentNo*, będące wektorem liczników wiadomości wysłanych do adresata wiadomości typu QUERY, zgodnie z wiedzą inicjatora Q_α. W polu *contPassive* zapisywany jest iloczyn logiczny wartości zmiennej $ontPassive_i$ oraz $\neg activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)$, określającego, czy proces może zostać uaktywniony przez sumę logiczną wiadomości już dostępnych oraz tych, które prawdopodobnie znajdują się jeszcze w kanałach komunikacyjnych. Pole *vSentNo* zawiera wektor liczników wiadomości aplikacyjnych wysłanych przez proces obserwowany przez monitor- nadawcę wiadomości REPLY, w chwili jest wysłania.

Detekcja zakończenia dynamicznego: Algorytm (2)

Swoje znaczenie identyczne do poprzednio omawianego algorytmu zachowują $\mathcal{AV}_i$, $terminationDetected_i$, *msgIn*, *pcktOut*, *queryOut* oraz *replyOut* i *replyIn*.

Zmienna $vSentNo_i$ jest tablicą [1..*n*] liczników, w której element $vSentNo_i[j]$ oznacza liczbę wiadomości wysłanych przez P_i do P_j;

Analogicznie, zmienna $vRecvNo_i$ jest tablicą [1..*n*] liczników, w której element $vRecvNo_i[j]$ oznacza liczbę wiadomości odebranych przez P_i od procesu P_j.

Ponadto, inicjator Q_α przechowuje zmienną $tSentNo_\alpha$ będącą tablicą [1..*n*, 1..*n*]. Wiersz *i* tej tablicy zawiera wektor $vSentNo_i$ przesłany ostatnio do inicjatora Q_α przez monitor Q_i, w wiadomości kontrolnej typu REPLY. W efekcie, każdy element tablicy $tSentNo_\alpha$ reprezentuje wiedzę inicjatora, o liczbie wiadomości aplikacyjnych wysłanych z P_i do P_j. Zauważmy, że wiedza ta nie jest precyzyjna, gdyż nadesłane wektory $vSentNo_i$, $i \in \{1,2,...,n\}$, odnoszą się w ogólności do różnych momentów czasu. $\mathcal{AIT}_i$ jest aproksymacją zbioru identyfikatorów procesów, które wysłały wiadomości niedostępne jeszcze dla P_i. Zmienna logiczna $contPassive_i$ posiada wartość *True*, gdy proces P_i pozostawał pasywny od momentu wysłania ostatniej wiadomości typu REPLY. Jeżeli wartość zmiennej logicznej $dTermDetected_i$ będzie równy *True*, inicjator podejmie decyzję o wykryciu zakończenia dynamicznego.

Detekcja zakończenia dynamicznego: Algorytm (3)

Inicjator detekcji Q_α rozpoczyna cykl detekcyjny wysyłając do wszystkich monitorów Q_i wiadomość kontrolną typu QUERY, zawierającą w polu *vSentNo* odpowiednią kolumnę tablicy $tSentNo_\alpha$, dzięki czemu każdy monitor Q_i będzie mógl dokonać aproksymacji liczby wiadomości wysłanych, lecz jeszcze nie odebranych przez Q_i (znajdujących się w kanałach komunikacyjnych). Wektor ten informuje odbiorcę Q_i o znanej inicjatorowi Q_α liczbie wiadomości wysłanych do P_i przez inne procesy.

Detekcja zakończenia dynamicznego: Algorytm (4)

Następnie inicjator oczekuje na zebranie odpowiedzi od wszystkich pozostałych procesów. Następnie sumuje wartości nadesłanych liczników wiadomości aplikacyjnych wysłanych przez te procesy – suma ta zostanie wykorzystana w następnym cyklu detekcyjnym. Algorytm stwierdza wykrycie zakończenia dynamicznego przetwarzania aplikacyjnego, gdy wszystkie odebrane w danym cyklu przez Q_α zmienne *replyIn.contPassive* mają wartość *True*. W przeciwnym wypadku, Q_α rozpoczyna następny cykl detekcyjny.

Detekcja zakończenia dynamicznego: Algorytm (5)

Po otrzymaniu zapytania typu QUERY, monitor Q_i może wyznaczyć zbiór $\mathcal{AIT}_i$ będący aproksymacją zbioru tych procesów, których wiadomości wysłane do procesu P_i są potencjalnie jeszcze w kanałach komunikacyjnych. (porównywany jest wektor *queryIn.vSentNo* odebrany przez Q_i z $vRecvNo_i$ – jeżeli $queryIn.vSentNo[j] > vRecvNo_i[j]$, to w kanale $C_{j,i}$ znajdą się wiadomości wysłane przez P_j do P_i, a nie odebrane jeszcze przez P_i)

Każdy z monitorów wyznacza wartość zmiennej *replyOut.contPassive*. Wartość *replyOut.contPassive* = *True* wtedy i tylko wtedy, gdy $contPassive_i$ = *True*, a predykat $activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)$ = *False*.

Po wyznaczeniu wartości *replyOut.contPassive* monitor wysyła odpowiedź typu REPLY zawierającą pole *contPassive* oraz aktualny wektor $vSentNo_i$, który będzie użyty przez Q_α do uaktualnienia tablicy $tSentNo_\alpha$.

Detekcja zakończenia dynamicznego: Algorytm (6)

Przy wysyłaniu (odbieraniu) wiadomości aplikacyjnych do Q_j. monitor Q_i inkrementuje *j*-ty wpis w tablicy $vRecvNo_i$ ($vSentNo_i$). W momencie aktywacji procesu P_i zmiennej $contPassive_i$ przypisywana jest wartość *False*.

Twierdzenie 10.4

Jeżeli algorytm jest wykonywany w chwili, gdy przetwarzanie aplikacyjne osiągnęło stan zakończenia dynamicznego, to algorytm ten zakończy się w skończonym czasie, stwierdzając wystąpienie dynamicznego zakończenia przetwarzania aplikacyjnego ($dTerminationDetected_\alpha$ = *True*).

Dowód twierdzenia 10.4 (1)

Cykl detekcyjny nie jest wstrzymywany przez monitory, a więc inicjator otrzyma odpowiedź od każdego Q_i w skończonym czasie.

Zakładamy, że przetwarzanie aplikacyjne osiągnęło stan zakończenia dynamicznego w chwili τ^t ($Dterm(\mathcal{P})[\tau^t] = True$). Od tego momentu zatem, wszystkie procesy pozostają ciągle pasywne:

$$passive_i[\tau^x] \text{ i } activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)[\tau^x] = False, \text{ dla każdego } \tau^x \geq \tau^x.$$

Dowód twierdzenia 10.4 (2)

Tak więc każdy cykl zainicjowany po τ^t, powiedzmy cykl *k*, nada wszystkim zmiennym $contPassive_i$ wartość *True*, i odbierze od monitorów Q_i wektor końcowych wartości liczników $vSentNo_i$. Dlatego, w chwili τ_b^{k+1} rozpoczęcia kolejnego cyklu detekcyjnego, wysłane zostaną ostateczne wartości liczników wiadomości wysłanych, a stąd $\mathcal{AIT}_i[\tau_i^{k+1}] = \mathcal{IT}_i[\tau_i^{k+1}]$. W efekcie otrzymujemy, że:

$$activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)[\tau_i^{k+1}] = activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)[\tau_i^{k+1}]$$

Dowód twierdzenia 10.4 (3)

Ponieważ ten ostatni predykat ma wartość *False* od chwili τ^t, a $\tau^t \leq \tau_i^k < \tau_i^{k+1}$, więc w cyklu *k*+1 wszystkie zmienne *replyIn.contPassive* będą miały wartość *True*, i w efekcie algorytm detekcji dynamicznego zakończenia przetwarzania aplikacyjnego, zakończy się.

Twierdzenie 10.5

Jeżeli algorytm stwierdza wystąpienie dynamicznego zakończenia przetwarzania rozproszonego, to przetwarzanie aplikacyjne jest w istocie w stanie zakończenia dynamicznego.

Szkic dowodu twierdzenia 10.5

Oznaczymy dwa ostatnie cykle detekcyjne odpowiednio przez *k* i *k*+1, a przez τ^x taki moment, że:

$$\tau_b^k < \tau_e^k \leq \tau^x \leq \tau_b^{k+1} < \tau_e^{k+1}$$

W dalszej części dowodu pokażemy, że jeżeli algorytm zakończy się w chwili τ_e^{k+1} z $dTerminationDetected_a[\tau_e^{k+1}] = True$, to $Dterm(\mathcal{P})[\tau^t] = True$. Oznacza to dalej, że są spełnione następujące dwa warunki dla każdego Q_i, $i \in \{1,2,...n\}$,

C4. $passive_i[\tau^x] = True$ (Wszystkie procesy są pasywne w chwili τ^x)

C5. $activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)[\tau^x] = False$ (Ani wiadomości już dostępne, ani znajdujące się w kanałach komunikacyjnych nie wystarczą do uaktywnienia procesu P_i)

Dowód warunku C4 pomijamy, jako identyczny jak w przypadku warunku C1 dla twierdzenia o poprawności algorytmu zakończenia detekcji zakończenia statycznego

Dowód warunku C5 (1)

Z konstrukcji algorytmu wynika, że algorytm ten zakończy się w chwili τ_e^{k+1} pod warunkiem, że predykat $activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)$ miał wartość $False$ w chwili τ_e^{k+1} dla każdego $i \in \{1,2,...n\}$.

Ponieważ każdy proces był pasywny w przedziale $\langle \tau_i^k, \tau_i^{k+1} \rangle$ dla każdego zachodzi:

C6. $\mathcal{AV}_i[\tau^x] \subseteq \mathcal{AV}_i[\tau_i^{k+1}]$

oraz $vSentNo_i[j]\,[\tau_i^k] = vSentNo_i[j]\,[\tau^x]$

Dowód warunku C5 (2)

Wyznaczając zbiór $\mathcal{AIT}_i[\tau_i^{k+1}]$ procesów, od których wiadomości są aktualnie transmitowane do P_i, algorytm uwzględnia w istocie następujące zależności:

$$vSentNo_j[i]\,[\tau_j^k] > vRecvNo_i[j]\,[\tau_i^{k+1}], \text{ gdyż}$$

$$vSentNo_j[i]\,[\tau_j^k] = gueryIn.vSentNo[j][\tau_i^{k+1}].$$

Zauważmy, że liczniki są monotoniczne:

$$vRecvNo_i[j]\,[\tau_i^k] \le vRecvNo_i\,[j]\,[\tau^x] \le vRecvNo_i[j][\tau_i^{k+1}]$$

Dowód warunku C5 (3)

Ponieważ dla każdego Q_i, $vSentNo_i[j]\,[\tau_i^k] = vSentNo_i[j][\tau^x]$ różnica między rzeczywistym zbiorem $\mathcal{IT}_i[\tau^x]$ a jego aproksymacją $\mathcal{AIT}_i[\tau_i^{k+1}]$, może wynikać tylko z dotarcia pewnych wiadomości do procesów docelowych. Wówczas jednak odpowiedni nadawcy zostaną uwzględnieni w zbiorze $\mathcal{AV}_i$. Tak więc, biorąc pod uwagę C6 możemy wnosić, że:

$$(\mathcal{AV}_i \cup \mathcal{IT}_i)\,[\tau^x\,] \subseteq (\mathcal{AV}_i \cup \mathcal{AIT}_i)\,[\tau_i^{k+1}].$$

Dalej, z monotoniczności predykatu $activate_i$ wynika, że dla każdego Q_i:

$$\neg activate_i(\mathcal{AV}_i \cup \mathcal{AIT}_i)[\tau_i^{k+1}] \Rightarrow \neg activate_i(\mathcal{AV}_i \cup \mathcal{IT}_i)\,[\tau^x]$$

Oznacza to, że warunek C5 jest spełniony.

Algorytm detekcji zakończenia dynamicznego: Złożoność

Aby wykryć wystąpienie zakończenia dynamicznego, niezbędne są w najgorszym wypadku dwa cykle detekcyjne po zakończeniu cyklu bieżącego

Ponieważ nie są przesyłane potwierdzenia, więc złożoność komunikacyjna algorytmu wynosi $4n$ (a więc jest mniejsza niż algorytm detekcji zakończenia statycznego przedstawionego wcześniej).

Algorytm detekcji zakończenia dynamicznego: Cechy

Porównując dalej ten algorytm z algorytmem detekcji zakończenia statycznego, widzimy, że wiadomości kontrolne w tym algorytmie są bardziej złożone, gdyż zawierają wektory liczników.

Opóźnienie detekcji jest mniejsze, ze względu na brak konieczności oczekiwania na potwierdzenia. Ponadto, wystąpienie zakończenia dynamicznego wyprzedza w czasie wystąpienie zakończenia statycznego. Uzyskane w sumie przyspieszenie momentu wykrycia zakończenia obliczeń rozproszonych może zatem mieć istotne znaczenie praktyczne, zwłaszcza w zastosowaniach czasu rzeczywistego.