Czas wirtualny, złożoność algorytmów

Plan wykładu

Celem wykładu jest zaznajomienie studenta z pojęciem zegara logicznego, scharakteryzowanie różnych rodzajów kanałów komunikacyjnych, a także przedstawienie analizy poprawności i złożoności algorytmów rozproszonych. W pierwszej części wykładu przedstawione zostaną dwa typy zegarów logicznych tzw. zegary skalarne i wektorowe oraz pojęcia z tym związane. Następnie zaprezentowane i omówione zostaną kanały komunikacyjne FIFO i typu FC. Dla każdego z omawianych mechanizmów przedstawiony zostanie przykładowy algorytm zawierający jego implementację. W drugiej części wykładu zdefiniowane zostaną takie pojęcia jak rząd funkcji czy też funkcja kosztu. Pojęcia te zostaną wykorzystane następnie przy omawianiu złożoności komunikacyjnej i czasowej algorytmów rozproszonych. Na zakończenie przedstawione zostaną warunki poprawności algorytmu rozproszonego.

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

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

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

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

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

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

Zegary logiczne - właściwości

W konsekwencji definicji, zegar logiczny posiada następujące właściwości:

1. 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'*.
2. W przypadku przesyłania wiadomości M, czas logiczny przyporządkowany zdarzeniu nadania wiadomości 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 **skalarnym**.

Realizacja zegarów skalarnych

L. Lamport zaproponował realizację skalarnego zegara logicznego, w której funkcja $\mathcal{T}(E)$ implementowana była przez zmienne naturalne $clock_i$, $1 \leq i \leq n$, 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. Szczegółową implementację tego mechanizmu przedstawia algorytm L. Lamporta.

Algorytm Lamporta (1)

Algorytm używa pakietów typu PACKET, przenoszących wiadomości aplikacyjne w polu $data$ oraz zawierających pole $clock$ będące etykietą czasową pakietu. Wiadomości $msgIn$ są typu PACKET, a zmienne $pcktOut$ są typu PACKET. Zmienna $clock_i$ reprezentuje skalarny zegar logiczny procesu P_i. Stała d oznacza jakąś ustaloną wartość, o którą jest zwiększana wartość zegara logicznego. Algorytm ten opisuje działania realizowane przez monitor Q_i w wyniku zajścia określonych zdarzeń.

Algorytm Lamporta (2)

Zdarzenie wysłania wiadomości aplikacyjnej $msgOut$ implikuje zwiększenie zmiennej $clock_i$ oraz utworzenie pakietu $pcktOut$. Pakiet ten zawiera wiadomość aplikacyjną w polu $pcktOut.data$, oraz aktualną wartość zegara lokalnego $clock_i$ w polu skalarnej etykiety czasowej $pcktOut.clock$. Tak zbudowany pakiet jest następnie wysyłany do monitora skojarzonego z adresatem wiadomości.

Algorytm Lamporta (3)

Odebranie pakietu $pcktIn$ przez monitor Q_i, prowadzi do wyznaczenia nowej wartości zegara logicznego odbiorcy, jako:

$$clock_i := max(clock_i, pcktIn.clock) + d$$

gdzie $pcktIn.clock$ jest wartością lokalnego zegara logicznego nadawcy w chwili odpowiadającej zajściu zdarzenia wysłania pakietu $pcktIn$. Po uaktualnieniu zegara przez monitor, wiadomość aplikacyjna jest udostępniona procesowi aplikacyjnemu P_i w wyniku wykonania przez monitor Q_i operacji $deliver(P_j, P_i, msgIn)$ i w efekcie zachodzi zdarzenie odbioru $e\_receive(P_j, P_i, msgIn)$.

W szczególności, każde zdarzenie lokalne odnotowane przez Q_i powoduje zwiększenie zmiennej $clock_i$ monitora Q_i.

Przykład synchronizacji zegarów logicznych

W przykładzie obrazującym synchronizację skalarnych zegarów przedstawionym na slajdzie, wartości zegarów logicznych wyznaczono zgodnie z propozycją Lamporta przyjmując, że zdarzeniom inicjującym odpowiada czas wirtualny 0, a każde kolejne zdarzenie lokalne zmienia wartość zegara o d=1.

Rozważmy teraz pary zdarzeń współbieżnych: $E_2^1 \| E_3^1$, $E_3^1 \| E_1^2$ oraz $E_1^2 \| E_3^2$. Relacje między czasami wirtualnymi tych zdarzeń są następujące:

$$\mathcal{T}(E_2^1) = \mathcal{T}(E_3^1)$$
$$\mathcal{T}(E_3^1) < \mathcal{T}(E_1^2)$$
$$\mathcal{T}(E_1^2) > \mathcal{T}(E_3^1)$$

Relacja między zbiorem zdarzeń i zbiorem wartości zegara skalarnego

Możliwe relacje między częściowo uporządkowanym zbiorem zdarzeń a w pełni uporządkowanym zbiorem wartości funkcji $\mathcal{T}$ przedstawione są na slajdzie.

Łatwo wykazać, że skalarne zegary logiczne zdefiniowane przez Lamporta spełniają podstawowy **warunek poprawności** zegarów, tzn. :

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

Jak widać jednak z przykładu, relacja odwrotna nie jest prawdziwa, a więc:

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

Tym samym zegary skalarne nie mogą być wykorzystywane do określania **relacji poprzedzania** między zdarzeniami. Są one natomiast wystarczające do jednoznacznego uporządkowania zbioru wszystkich zdarzeń. W tym celu tworzy się na ogół **rozszerzoną etykietę czasową** będącą konkatenacją skalarnego zegara logicznego oraz, przykładowo, statycznego identyfikatora procesu.

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

Mattern i niezależnie Fidge (a także inni badacze) zaproponowali realizację zegarów wektorowych, w której funkcja $\mathcal{T}^V$ implementowana była przez zmienne tablicowe $vClock_i$, $1 \le i \le n$, skojarzone z poszczególnymi procesami. Zmienna $vClock_i$ jest tablicą [1.. n] liczb naturalnych, odpowiadającą pewnej aproksymacji czasu globalnego z perspektywy procesu P_i. Zmienna $vClock_i[i]$ reprezentuje przy tym w każdej chwili skalarny czas lokalny procesu P_i, a zmienna $vClock_i[j]$, $j \neq i$ aktualne wyobrażenie procesu P_i o bieżącym skalarnym czasie lokalnym procesu P_j. 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.

Szczegółową implementację mechanizmu zegara wektorowego przedstawia algorytm zaprezentowany na kolejnych slajdach.

Algorytm Matterna (1)

Algorytm używa typu wiadomości PACKET posiadającej dodatkowe pole $vClock$ zawierające wektor będący etykietą czasową wiadomości.

Zmienna d jest wartością, o jaką powiększane są pozycje tablicy $vClock_i$, reprezentującej wektorowy logiczny zegar czasowy.**Algorytm Matterna (2)**

Zdarzenie wysłania wiadomości aplikacyjnej powoduje zwiększanie $vClock_i[i]$, oraz utworzenie pakietu $pcktOut$ zawierającego wiadomość aplikacyjną $msgOut$ w polu $pcktOut.data$ i nową wartość zegara $vClock_i$ jako etykietę czasową pakietu w polu $pcktOut.vClock$. Tak zbudowany pakiet jest następnie przesyłany do monitora skojarzonego z adresatem wiadomości aplikacyjnej.

Algorytm Matterna (3)

Odebranie pakietu $pcktIn$ przez Q_i powoduje zwiększenie $vClock_i[i]$, a następnie wyznaczenie nowej wartości zegara wektorowego odbiorcy P_i, jako:

$$\forall k :: 1 \leqslant k \leqslant n :: vClock_i[k] := max(vClock_i[k], pcktIn.vClock[k])$$

Po uaktualnieniu zegara przez monitor, wiadomość aplikacyjna jest udostępniona procesowi P_i i zachodzi odpowiednie zdarzenie odbioru.

Podobnie jak w przypadku zegarów skalarnych, monitor Q_i uaktualnia zegar $vClock_i$ w wyniku zajścia każdego zdarzenia wewnętrznego procesu aplikacyjnego. Uaktualnienie to jest związane ze zmianą jedynie pozycji $vClock_i[i]$.

Zegary wektorowe (1)

W każdej chwili czasu rzeczywistego

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

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

Relacje na etykietach wektorowych

Zdefiniujmy teraz relacje na zmiennych wektorowych $vClock_i$ i $vClock_j$, reprezentujących tablice $[1..n]$, w następujący sposób:

$$vClock_i = vClock_j \Leftrightarrow \forall_k\ vClock_i[k] = vClock_j[k]$$

$$vClock_i \neq vClock_j \Leftrightarrow \exists_k\ vClock_i[k] \neq vClock_j[k]$$

$$vClock_i \leqslant vClock_j \Leftrightarrow \forall_k\ vClock_i[k] \leqslant vClock_j[k]$$

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

$$vClock_i < vClock_j \Leftrightarrow vClock_i \leqslant vClock_j \ \wedge\ vClock_i \neq vClock_j$$

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

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

Zegary wektorowe (2)

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

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

Można wykazać, że nie istnieją zegary wektorowe o liczbie elementów mniejszej od n, dla których prawdziwe byłoby powyższe twierdzenie.

Kanały FIFO

W ogólności, każda wiadomość jest przetwarzana (interpretowana) w pewnym kontekście, określonym często przez wiadomości wcześniej odebrane. Brak właściwego kontekstu, spowodowany na przykład zmianą kolejności przesyłanych wiadomości, może prowadzić do niewłaściwej interpretacji wiadomości i w efekcie do błędów przetwarzania. Jeżeli, dla ilustracji, rozważymy system rezerwacji miejsc lotniczych, to właściwym kontekstem dla wiadomości anulującej rezerwację, jest oczywiście wiadomość zgłoszenia rezerwacji. Wiadomość o anulowaniu nie powinna być odebrana i standardowo przetwarzana przez system rezerwacji, jeśli nie dotarło jeszcze odpowiednie zgłoszenie. Dla takiej aplikacji wskazana byłaby zatem komunikacja gwarantująca porządek odbioru zgodny z kolejnością wysłania - porządek **FIFO**. Dokładniej, należałoby zagwarantować w sposób niewidoczny dla procesów aplikacyjnych, że jeżeli proces P_i wysyła do P_j wiadomość M przed wysłaniem do P_j wiadomości M', to proces P_j nie odbierze (nie otrzyma) M' dopóty, dopóki nie odbierze M.

Mechanizm gwarantujący właściwą kolejność interpretowania wiadomości można oczywiście wbudować w procesy aplikacyjne, lecz wówczas to projektant aplikacji musi rozwiązać wspomniany problem. Wskazane jest jednak często inne postępowanie, polegające na skonstruowaniu, na przykład przez odpowiednią rozbudowę monitorów Q_i, warstwy programowej pośredniczącej między środowiskiem komunikacyjnym nie zachowującym uporządkowania FIFO, a procesami aplikacyjnymi P_i. Warstwa taka – przechowując i ewentualnie wstrzymując w buforach wiadomości, które dotarły zbyt wcześnie – zapewniałaby dostarczenie wiadomości do procesów aplikacyjnych w porządku FIFO. Ideę takiego podejścia ilustruje slajd kolejny, a jego przykładową implementację algorytm przedstawiony na kolejnych slajdach. Formalnie, warstwa monitorów powinna zagwarantować dla wszystkich procesów i każdej wiadomości, że:

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

Algorytm Müllender'a (1)

Algorytm używa wiadomości typu PACKET posiadających dodatkowe pole $seqNo$ będące licznikiem sekwencyjnym wiadomości.

Zbiór $delayBuf_i$ pełni rolę bufora pakietów oczekujących na nadejście i odbiór pakietów go poprzedzających.

Numery sekwencyjne dla wiadomości ostatnio przekazanych aplikacji przechowywane są w tablicy $delivNo_i[j]$.

Tablica $seqNo_i$ zawiera numery sekwencyjne wiadomości wysyłanych przez P_i.

Algorytm Müllender'a (2)

W opisywanym algorytmie, każda wiadomość aplikacyjna wysłana przez proces P_i do procesu P_j jest umieszczona w pakiecie, zawierającym dodatkowo kolejny numer sekwencyjny $seqNo$ odpowiadający liczbie wiadomości wysłanych przez P_i do P_j. Tak więc, każde wysłanie związane jest ze zwiększaniem licznika wiadomości wysłanych $seqNo_i[j]$ i umieszczeniem tego licznika w pakiecie w polu $pcktOut.seqNo$ numeru sekwencyjnego wiadomości.

Algorytm Müllender'a (3)

Po odebraniu przez monitor Q_i wiadomości wysłanej przez proces P_j do procesu P_i, monitor sprawdza czy numer sekwencyjny w odebranym pakiecie jest większy o jeden od numeru sekwencyjnego wiadomości ostatnio przekazanej do aplikacji ($delivNo_i[j]$). Jeśli tak, to odebrana przez Q_i wiadomość jest przekazywana procesowi aplikacyjnemu. W przeciwnym razie,

odebrany przez Q_i pakiet jest umieszczany w zbiorze $delayBuf_i$ pakietów oczekujących na nadejście i odbiór pakietów go poprzedzających.

Algorytm Müllender'a (4)

Po przekazaniu każdej wiadomości procesowi aplikacyjnemu P_i, monitor Q_i sprawdza, czy w zbiorze $delayBuf_i$ nie ma pakietu, którego wiadomość mogłaby teraz zostać przekazana aplikacji. Jeśli taka wiadomość istnieje, to jest ona przekazywana procesowi P_i. Jeśli jednak warunek przekazania wiadomości dla żadnego pakietu nie jest spełniony, to monitor oczekuje na nadejście kolejnego pakietu.

Cechy kanałów FIFO

Kanały FIFO są pewnym mechanizmem synchronizacji wymaganym przez wiele aplikacji. Ułatwiają one znalezienie rozwiązania i konstrukcję algorytmów rozproszonych dla wielu problemów. Z drugiej jednak strony, są mechanizmem ograniczającym, w porównaniu z kanałami nonFIFO, współbieżność komunikacji, a tym samym efektywność przetwarzania.

Kanały typu *FC*

Uwzględniając wspomniane wcześniej cechy kanałów FIFO, zaproponowano rozwiązanie, nazwane kanałami typu FC (ang. *Flush Channels*), które w swym zamyśle miało łączyć zalety kanałów FIFO i nonFIFO, a więc pewien stopień synchronizacji i współbieżnej komunikacji.

Kanał typu FC realizuje zestaw czterech mechanizmów (operacji) komunikacji:

- $send^t$ (ang. *two-way-flush send*),
- $send^f$ (ang. *forward-flush send*),
- $send^b$ (ang. *backward-flush-send*),
- $send^o$ (ang. *ordinary send*).

Zdarzenia odpowiadające wykonaniu tych operacji oznaczono przez $send^t$, $send^f$, $send^b$ oraz $send^o$, a wiadomości przesyłane z ich użyciem – przez M^t, M^f , M^b , M^o . Wiadomości spełniają przy tym pewne warunki odnoszące się do zmiany ich kolejności w kanale (wyprzedzania).

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, wysłana w wyniku wykonania operacji $send^t$, z definicji nie wyprzedza w kanale żadnej wiadomości, ani też nie dopuszcza by inna wiadomość ją wyprzedziła. Wiadomość taką nazywać też będziemy wiadomością typu TF. Wiadomość M^f (wiadomość typu FF), wysłana w efekcie wykonania operacji $send^f$, nie wyprzedza w kanale żadnej innej wiadomości. Z kolei wiadomość M^b (wiadomość typu BF), wysłana w wyniku wykonania operacji $send^b$, nie dopuszcza by ją wyprzedziła inna wiadomość. W końcu wiadomość M^o (wiadomość typu OF),

wysłana w efekcie wykonania operacji $send^o$, może wyprzedzać i być wyprzedzana przez inne wiadomości.

Łatwo zauważyć, że najmniejszą swobodę w komunikacji i w efekcie najmniejszy poziom współbieżności, oferuje operacja $send^t$, a największą swobodę - operacja $send^o$. Pierwsza z nich modeluje w istocie kanały FIFO, a ostatnia - kanały nonFIFO.

Implementacja kanałów FC

Kanały typu FC, utożsamiane z operacjami $send^t$, $send^f$, $send^b$ i $send^o$ mogą być implementowane z użyciem różnych mechanizmów: selektywnego rozgłaszania, liczników, potwierdzeń itp. Dla przykładu, przedstawimy implementację kanałów typu FC.

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

Oznaczmy przez $FL_{i,j}$ stan kanału $FC_{i,j}$. Stosownie do definicji, kanał $FC_{i,j}$ narzuca pewne ograniczenia na kolejność odbioru przez P_j wiadomości wysłanych przez P_i. Formalnie ograniczenia te wyraża binarna **relacja poprzedzania typu** F, oznaczona przez $\prec^+_{i,j}$, zdefiniowana na zbiorze $FL_{i,j}$ w sposób następujący:

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

Bezpośrednie poprzedzanie

Jeżeli natomiast zachodzi predykat:

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

to powiemy, że M **bezpośrednio poprzedza** M' i fakt ten oznaczamy $M \prec_{i,j} M'$, tym samym:

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

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

Zauważmy, że znajomość relacji $\prec_{i,j}$, pozwala na prostą implementację kanałów FC, w której wiadomości M' nie są przekazywane procesom aplikacyjnym jeśli nie zostały wcześniej przekazane wszystkie wiadomości poprzedzające w relacji $\prec_{i,j}$. W celu implementacji kanałów FC należy zatem rozwiązać już tylko problem efektywnego wyznaczenia relacji $\prec_{i,j}$ i przekazywania istotnych jej elementów do monitora odbiorcy.

W prezentowanym algorytmie zaproponowano mechanizm sukcesywnej konstrukcji relacji $\prec_{i,j}$ poprzez stosowne jej uaktualnienie przy wysyłaniu kolejnych wiadomości.

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

W mechanizmie tym, istotne znaczenie ma wiadomość typu TF lub BF ostatnio wysłana kanałem $FC_{i,j}$, oznaczona przez $M_{i,j}{}^b$. Zauważmy, że wszystkie wiadomości M wysłane po $M_{i,j}{}^b$, powinny być również po niej odebrane. Początkowo $M_{i,j}{}^b = \emptyset$. W tym kontekście, konstrukcja relacji $\prec_{i,j}$ realizowana jest stosownie do typu wysłanej wiadomości, w sposób następujący:

1. Jeżeli jest typu OF i $M_{i,j}{}^b \neq \emptyset$, to $\prec_{i,j} := \prec_{i,j} \cup \{\langle M_{i,j}{}^b, M \rangle\}$

2. Jeżeli jest typu BF i $M_{i,j}^{\ b} \neq \emptyset$, to $\prec_{i,j} := \prec_{i,j} \cup \{\langle M_{i,j}^{\ b}, M\rangle\}$, następnie, $M_{i,j}^{\ b} := M$.

Konstrukcja relacji $\prec_{i,j}$ (3)

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

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

Jak widać, konstrukcja relacji $\prec_{i,j}$ po stronie nadawcy nie stanowi problemu. W celu implementacji kanałów FC trzeba jednak jeszcze rozwiązać kwestię efektywnego przekazywania monitorowi odbiorcy informacji o istotnych elementach relacji $\prec_{i,j}$. Dlatego też w opisywanym algorytmie zaproponowano przesyłanie wiadomości aplikacyjnych M w pakietach, zawierających dodatkowo:

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

Algorytm Kearnsa, Campa i Ahuja (1)

W algorytmie tym używany jest jeden typ wiadomości. Jest to typ `PACKET`, który posiada cztery wspomniane wcześniej pola. Pierwsze z nich, pole $type$, określa rodzaj przesyłanego znacznika, drugie $seqNo$ zawiera numer sekwencyjny wiadomości, kolejne $waitForNo$ jest polem, w którym przesyłany jest numer sekwencyjny wiadomości bezpośrednio poprzedzającej i wreszcie ostatnie pole $data$ zawierające właściwą wiadomość aplikacyjną.

Podstawowe zmienne wykorzystywane przez algorytm mają następujące znaczenie:

- $msgIn$ jest wiadomością aplikacyjną
- $pcktOut$ jest zmienną reprezentującą pakiet przenoszący wiadomość
- $delayBuf_i$ to wektor zbiorów pakietów oczekujących
- $seqNo_i$ jest z kolei wektorem numerów sekwencyjnych wiadomości ostatnio wysłanych
- $backFP_i$ to wektor, który będzie zawierał numery sekwencyjne ostatnio wysłanych wiadomości, których nie można wyprzedzić
- $delivNo_i$ to z kolei wektor zbiorów numerów sekwencyjnych wiadomości ostatnio przekazanych aplikacji
- $delivered_i$ reprezentuje zmienną typu logicznego, która oznacza status próby doręczenia wiadomości do procesu aplikacyjnego

Algorytm Kearnsa, Campa i Ahuja (2)

Procedura `TryToDeliver` to procedura odpowiedzialna za dostarczenie odebranej wiadomości aplikacyjnej do procesu. Przed wykonaniem tej czynności muszą zostać sprawdzone odpowiednie warunki, które mówią czy dostarczenie takie jest możliwe. Jeżeli odebrany pakiet

był pakietem typu OF lub BF to należy sprawdzić czy została już dostarczona wiadomość bezpośrednio poprzedzająca wiadomość odebraną. W tym celu porównywane są odpowiednie wartości z pola *waitForNo* pakietu i tablicy $delivNo_i$. Jeśli natomiast odebrany pakiet jest typu FF lub TF to przed jego dostarczeniem powinny zostać dostarczone wszystkie wiadomości z numerem sekwencyjnym mniejszym lub równym od wartości pola *waitForNo*. W przypadku dostarczenia wiadomości do procesu fakt ten jest odnotowywany w tablicy $delivNo_i$ zawierającej zbiory numerów sekwencyjnych wiadomości przekazanych do procesów aplikacyjnych.

Algorytm Kearnsa, Campa i Ahuja (3)

Wysłanie wiadomości typu OF, jest poprzedzone przygotowaniem odpowiedniego pakietu. Zwiększany jest numer sekwencyjny wiadomości, po czym przepisywany jest on do właściwego pola pakietu. Do pola *waitForNo* pakietu zapisywany jest numer wiadomości ostatnio wysłanej do procesu nadawcy, której nie można wyprzedzić. Numer ten jest przechowywany w odpowiednim polu tablicy $backFP_i$.

Algorytm Kearnsa, Campa i Ahuja (4)

Akcje związane z wysłaniem wiadomości typu BF, są w zasadzie identyczne jak akcje podejmowane przy wysłaniu wiadomości typu OF. Jedyna różnica polega na tym, że po wysłaniu wiadomości jej numer sekwencyjny jest zapamiętywany w tablicy $backFP_i$, czyli w tablicy wiadomości, które nie mogą zostać wyprzedzone.

Algorytm Kearnsa, Campa i Ahuja (5)

Operacja wysłania wiadomości typu FF, wygląda podobnie. Różnice w stosunku do wysyłania wiadomości typu OF polegają na innej wartości pola *waitForNo* pakietu. Otóż w pole to zapisywany jest numer sekwencyjny ostatnio wysłanej wiadomości, czyli numer o 1 mniejszy od bieżącej wartości zmiennej $seqNo_i$.

Algorytm Kearnsa, Campa i Ahuja (6)

Ostatnia z operacji wysłania to operacja wysłania wiadomości typu TF. W przypadku wysyłania wiadomości tego typu, czynności wykonywane przed i po jej wysłaniu są zgodne z odpowiednimi czynnościami wykonywanymi w przypadku wysyłania wiadomości typu BF. Jedyna różnica, która ma jednak duże znaczenie, to nadanie innej wartości polu *waitForNo* pakietu. Otóż pole to, podobnie jak w przypadku wiadomości typu FF, przyjmuje wartość numeru sekwencyjnego ostatnio wysłanej wiadomości, czyli wartość zmiennej $seqNo_i$ pomniejszoną o 1.

Algorytm Kearnsa, Campa i Ahuja (7)

Odebranie wiadomości powoduje próbę dostarczenia tej wiadomości do procesu aplikacyjnego. Jeśli próba ta zakończy się niepowodzeniem, co można stwierdzić brakiem odpowiedniego wpisu w tablicy $delivNo_i$, to pakiet taki jest dodawany do zbioru pakietów oczekujących na dostarczenie.

Algorytm Kearnsa, Campa i Ahuja (8)

W przypadku udanego dostarczenia wiadomości odebranej, należy sprawdzić czy wśród wiadomości oczekujących nie ma wiadomości, które mogłyby być także dostarczone do procesu aplikacyjnego. Jeśli operacje takie zostaną znalezione podejmowana jest próba ich dostarczenia. A w przypadku powodzenia wykonania tej operacji, są one usuwane ze zbioru wiadomości oczekujących.

Środowisko nie zachowujące uporządkowania przyczynowego

Porządek FIFO czy porządek gwarantowany przez kanały *FC* jest wystarczający, jeżeli niezbędny kontekst wiadomości M wysłanej z P_i do P_j związany jest tylko z wcześniejszymi wiadomościami przesłanymi bezpośrednio między P_i a P_j. Jednakże, pewne zależności mogą być pośrednie. Na przykład, przyjmijmy zgodnie z rysunkiem na slajdzie, że proces (użytkownik) P_i przesyła wiadomości M_1 i M_2, odpowiednio do P_3 i P_2, zawierające tę samą wersję pewnej opinii. Proces P_2 po otrzymaniu i przeanalizowaniu wiadomości, wysyła stosowny komentarz w wiadomościach M_3 i M_4, odpowiednio do P_3 i P_1.

Jeżeli zgodnie z rysunkiem, wiadomość M_3 dotrze do P_3 przed wiadomością M_1, to sam komentarz bez wyjściowej opinii będzie niezrozumiały. Powstały problem wynika w istocie z faktu, że środowisko komunikacyjne nie gwarantuje tak zwanego **uporządkowania przyczynowego wiadomości**, co rodzi trudności nawet mimo gwarantowania uporządkowania FIFO.

Środowisko zachowujące uporządkowanie przyczynowe

Formalnie, **środowisko komunikacyjne zachowujące uporządkowanie przyczynowe wiadomości** zapewnia dla wszystkich procesów i wiadomości, że:

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

Algorytm Birmana, Schipera i Stephensona (1)

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

W algorytmie tym, z każdym procesem P_i skojarzony jest specyficzny zegar wektorowy zdarzeń, który jest reprezentowany przez tablicę [1.. n]. Wartość tego zegara jest zwiększana przez monitor Q_i tylko przy wysyłaniu wiadomości, a uaktualniana po spełnieniu warunków odebrania wiadomości przez adresata. Dlatego też, dla odróżnienia od typowego zegara wektorowego, zegar ten oznaczany jest przez $vSentClock_i$.

Algorytm Birmana, Schipera i Stephensona (2)

Dokładniej, wysłanie wiadomości, poprzedzane jest zawsze zwiększeniem o 1 wartości zmiennej $vSentClock_i[i]$, która pełni tym samym rolę licznika wysłanych dotychczas wiadomości. Wiadomość aplikacyjna jest umieszczana w pakiecie $pcktOut$ wraz z wektorem $vSentClock_i$, a pakiet ten zostaje wysłany do wszystkich pozostałych monitorów przetwarzania rozproszonego.

Algorytm Birmana, Schipera i Stephensona (3)

Po odebraniu pakietu $pcktIn$ od Q_j, monitor Q_i wstrzymuje dalsze jego przetwarzanie, do momentu gdy spełnione zostaną dwa następujące warunki. Pierwszy z nich polega na sprawdzeniu czy otrzymany pakiet zawiera wiadomość wysłaną przez proces P_j do P_i bezpośrednio po wiadomości o numerze sekwencyjnym zapisanym w odpowiednim polu tablicy $vSentClock_i$. Warunek ten gwarantuje zatem uporządkowanie FIFO w kanale. Drugi z warunków oznacza, że nadawca P_j wiadomości M nie otrzymał przed wysłaniem wiadomości M innej wiadomości M' od P_k o numerze sekwencyjnym większym niż $vSentClock_i[k]$. Gdyby taki przypadek miał miejsce, to wysłanie wiadomości M przez P_j byłoby przyczynowo zależne od wysłania wiadomości M', a wówczas wiadomość M' powinna być odebrana przez wszystkie procesy, w tym również przez P_i, przed wiadomością M.

Jeżeli oba warunki zostaną spełnione, monitor odbiorcy Q_i sprawdza, czy adresatem wiadomości jest proces P_i. W przypadku, gdy rzeczywiście wiadomość aplikacyjna skierowana jest do tego procesu, monitor przekazuje ją adresatowi. Następnie monitor uaktualnia swój zegar wektorowy. Jeżeli natomiast nie jest spełniony jeden z powyższych warunków to pakiet zawierający wiadomość M jest zachowywany w monitorze Q_i. Stan taki jest utrzymywany aż do

nadejścia pewnego kolejnego pakietu, którego wiadomość będzie mogła zostać odebrana, być może spełniając w efekcie spełnione zostaną oba powyższe warunki dla wiadomości M.

Algorytm Schipera, Egli, Sandoza (1)

Kolejny algorytm, realizuje natomiast warstwę pośrednią środowiska zachowującego uporządkowanie przyczynowe bez konieczności rozgłaszania wszystkich pakietów.

Jedynym typem komunikatów przesyłanym pomiędzy monitorami jest typ PACKET, zawierający odpowiednie zegary wektorowe, informację o etykietach czasowych zdarzenia wysłania ostatniej wiadomości do każdego z procesów, oraz bieżącej wiadomości.

Algorytm Schipera, Egli, Sandoza (2)

Podstawowe zmienne wykorzystywane w tym algorytmie mają następujące znaczenie:

- $msgIn$ jest wiadomością aplikacyjną
- $pcktOut$ jest zmienną reprezentującą pakiet przenoszący wiadomość
- $vSentClock_i$ to wektorowy zegar logiczny zdarzeń wysłania
- vP_i jest zbiorem odpowiednich par zawierających informację o etykietach czasowych ostatnio wysłanych wiadomości do poszczególnych procesów, którego liczność nie przekracza *n*-1. Poszczególne elementy tego zbioru, czyli odpowiednie pary składać się będą z identyfikatora procesu, do którego wysłana została już wiadomość aplikacyjna i etykiety czasowej zdarzenia wysłania ostatniej takiej wiadomości.
- $delayBuf_i$, to zmienna zawierająca zbiór wiadomości oczekujących

Algorytm Schipera, Egli, Sandoza (3)

Procedura TRYTODELIVER to procedura odpowiedzialna za dostarczenie odebranej wiadomości aplikacyjnej do procesu. Przed wykonaniem tej czynności monitor Q_i sprawdza, czy w zbiorze vM odebranego pakietu znajduje się element dotyczący procesu P_i. Jeśli takiego elementu nie ma, to wiadomość jest natychmiast dostarczana. W przeciwnym razie, w zależności od wyniku porównania etykiety czasowej zapisanej w tym elemencie i wartości lokalnego zegara logicznego, wiadomość ta jest opóźniana lub dostarczana do procesu aplikacyjnego.

Algorytm Schipera, Egli, Sandoza (4)

W przypadku gdy wiadomość zostanie dostarczona do procesu aplikacyjnego uaktualniany jest odpowiednio zbiór vP i logiczny zegar wektorowy.

Algorytm Schipera, Egli, Sandoza (5)

W prezentowanym algorytmie, wysyłana z procesu P_i do procesu P_j wiadomość M, umieszczona jest w pakiecie zawierającym dodatkowo wektorową etykietę czasową zdarzenia wysłania w polu $vSentClock$, oraz aktualny zbiór vP_i jako pole vM. Po wysłaniu pakietu, zbiór vP_i jest zawsze uaktualniany – zmieniany jest na bieżący czas ostatnio wysłanej wiadomości albo dodawany odpowiedni nowy element.

Algorytm Schipera, Egli, Sandoza (6)

Po odebraniu pakietu przez Q_i, monitor próbuje dostarczyć zawartą w pakiecie wiadomość aplikacyjną do procesu aplikacyjnego P_i, wywołując procedurę TRYTODELIVER. Po przekazaniu każdej wiadomości procesowi aplikacyjnemu P_i, monitor Q_i sprawdza, czy w zbiorze $delayBuf_i$ nie ma pakietu, którego wiadomość mogłaby teraz zostać przekazana aplikacji. Jeśli taka

wiadomość istnieje, to zostaje ona przekazana procesowi P_i. W efekcie zegar wektorowy i zbiór vP_i są odpowiednio uaktualniane.

Funkcje kosztu – oznaczenia

Oznaczmy przez Δ^{δ}_{A} zbiór wszystkich poprawnych danych wejściowych δ algorytmu A, a przez $\mathcal{Z}^{*}_{A}(\delta)$ – **koszt wykonywania** tego algorytmu dla danych δ, gdzie $\delta \in \Delta^{\delta}_{A}$ i $\mathcal{Z}^{*}_{A} : \Delta^{\delta}_{A} \rightarrow \mathbb{R}$.

Niech ponadto, μ będzie rozmiarem danych wejściowych δ (rozmiarem zadania), takim 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 zatem będziemy odwzorowanie:

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

Gdzie Δ^{μ}_{A} oznacza zbiór wszystkich danych wejściowych o rozmiarze μ algorytmu *A*.

Funkcje kosztu wykonania algorytmów

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

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

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

W praktyce interesuje nas na ogół nie tyle dokładna wartość $\mathcal{Z}_{A}(\mu)$, co raczej jej rząd. Dlatego stosuje się powszechnie następujące oznaczenia.

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:

$$\boldsymbol{f = O(g)}$$

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

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

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:

$$f = \Theta(g)$$

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:

$$f = \Omega(g)$$

jeżeli $g = O(f)$.

Złożoność czasowa

W przypadku algorytmów rozproszonych, *złożoność czasowa* jest funkcją kosztu wykonania, wyrażoną przez liczbę kroków algorytmu do jego zakończenia przy spełnieniu następujących założeń:

- czas wykonywania każdego kroku (operacji, zdarzenia) 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) a czas transmisji jest jednostkowy. Taką postać definicji stosować będziemy w dalszej części wykładu.

Złożoność komunikacyjna

Złożoność komunikacyjna jest funkcją kosztu wykonania algorytmu wyrażaną przez:

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

W konsekwencji wyróżniamy zatem tak zwaną złożoność pakietową i bitową.

Przykład (1) – bariera

Aby zilustrować pojęcie złożoności czasowej oraz komunikacyjnej, rozważmy bardzo prosty, przykładowy algorytm implementujący operację synchronizacji procesów tzw. bariery, w której każdy proces może kontynuować działanie tylko wtedy, gdy wszystkie inne procesy osiągną pewien określony stan. Załóżmy, że topologia połączeń ma strukturę grafu pełnego, czyli każdy proces jest połączony kanałem komunikacyjnym z wszystkimi pozostałymi procesami.

Rozważmy najpierw rozwiązanie, w którym pewien wydzielony, znany z góry proces-koordynator rozgłaszałby wiadomość typu `BARRIER` oznaczającą początek synchronizacji (bariery) i wstrzymywałby przetwarzanie. Każdy proces po otrzymaniu takiego komunikatu również wstrzymywałby przetwarzanie, odsyłając komunikat potwierdzający ten fakt do koordynatora. Po zebraniu wszystkich potwierdzeń koordynator rozsyłałby do wszystkich procesów komunikat typ `END` oznaczający koniec operacji bariery.

Przykład (1) – rysunek

W tym przykładzie koordynator wysyła *n*-1 wiadomości początku bariery, otrzymuje n-1 potwierdzeń, i następnie wysyła *n*-1 wiadomości końca bariery. Widać więc, że złożoność komunikacyjna wynosi 3*(*n*-1), podczas gdy złożoność czasowa, przy założeniu wykorzystania istniejącego mechanizmu rozgłaszania, wynosi 3.

Przykład (2) – bariera

Aby lepiej zrozumieć pojęcia złożoności czasowej oraz komunikacyjnej, rozważmy jeszcze jeden algorytm implementujący operację bariery w środowisku rozproszonym o topologii pierścienia. Idea tego algorytmu byłaby następująca. Koordynator, czyli pewien wyznaczony proces, wysyłałby do swojego następnika w pierścieniu komunikat typu `BARRIER` oznaczający początek operacji bariery. Każdy proces po otrzymaniu tego komunikatu natychmiast wstrzymywałby pracę i przesyłałby komunikat dalej. Gdy komunikat ten dotarłby ponownie do koordynatora, ten ostatni wysyłałby kolejny komunikat, tym razem typu `END`, powiadamiający o końcu bariery – po otrzymaniu tego komunikatu wszystkie procesy podejmowałby dalsze przetwarzanie.

Przykład (2) – rysunek

W przykładzie ilustrującym ten algorytm każdy proces przesyła jedną wiadomość typu BARRIER, razem jest więc ich *n*. Analogicznie, każdy proces wysyła jedną wiadomość typu END. Ich liczba również jest równa *n*. Złożoność komunikacyjna wynosi więc 2*n*, i taka sama jest złożoność czasowa omawianego algorytmu.

Warunki poprawności

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

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

Na przykład, w odniesieniu do problemu wzajemnego wykluczania, właściwość bezpieczeństwa oznacza, że nigdy dwa procesy nie znajdą się jednocześnie w swoich sekcjach krytycznych, a właściwość żywotności - że każdy z procesów znajdzie się w końcu w swojej sekcji krytycznej (w skończonym, choć nieprzewidywalnym czasie).