ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1997
Seria: INFORMATYKA z. 32 Nr kol. 1356

Marcin KAPUSTA, Krzysztof KOŻBIAŁ, Aleksander LAURENTOWSKI*
Akademia Górniczo - Hutnicza w Krakowie, Katedra Informatyki

KONSYSTENTNOŚĆ INFORMACJI W SYSTEMIE MONITOROWANIA OBIEKTOWYCH APLIKACJI ROZPROSZONYCH

Streszczenie. Artykuł prezentuje algorytm porządkowania zdarzeń w systemach rozproszonych, wywodzący się z dwóch algorytmów zegarów wektorowych: algorytmu bezpośrednich zależności i algorytmu adaptacyjnego. Został on opracowany dla potrzeb środowiska monitorowania obiektowych aplikacji rozproszonych MODIMOS. Wyniki przeprowadzonych pomiarów wskazują, iż zastosowanie algorytmu jest uzasadnione częstością występowania zakłóceń porządku przyczynowo - skutkowego i w związku z tym umiarkowany narzut na czas wykonania aplikacji może zostać zaakceptowany.

DATA CONSISTENCY IN SYSTEM OF MONITORING DISTRIBUTED OBJECT APPLICATIONS

Summary. This paper presents an ordering algorithm derived from the direct dependencies and adaptive vector clocks algorithms. The algorithm was implemented in the MODIMOS distributed object applications monitoring system. Tests have shown usefulness of the algorithm in the presence of high percentage of causal inconsistencies and therefore the introduced overhead tends to be acceptable.

* Praca częściowo finansowana w ramach grantu KBN nr 8T11C01210. A. Laurentowski jest stypendystą Fundacji na Rzecz Nauki Polskiej.

1. Wstęp

Artykuł ten dotyczy problemu zapewnienia konsystentności informacji w systemie monitorowania obiektowych aplikacji rozproszonych. Prezentuje on algorytm zegarów wektorowych powstały z połączenia algorytmu bezpośrednich zależności [6] z własnościami algorytmu adaptacyjnego [7]. Prezentowany algorytm powstał dla potrzeb systemu MODIMOS [8] rozwijanego w Katedrze Informatyki Akademii Górniczo - Hutniczej w Krakowie. System ten umożliwia obserwację i wizualizację zdarzeń związanych z istnieniem obiektów oraz komunikacją w rozproszonych systemach heterogenicznych programowo. Ponieważ wybór algorytmu jest ściśle związany z cechami środowiska, w którym ma on zostać zastosowany, po zaprezentowaniu algorytmu pokrótce omówiony zostanie system MODIMOS wraz z jego cechami charakterystycznymi. W końcowej części artykułu zostaną przedstawione wyniki pomiarów przeprowadzonych w rozproszonym środowisku obiektowym SR [9, 10], w którym zastosowano ten algorytm.

2. Opis algorytmu

Monitorowanie zdarzeń w systemie rozproszonym wiąże się z możliwością zakłócenia porządku przyczynowo-skutkowego obserwowanych zdarzeń. Dla odtworzenia ich właściwej kolejności należy zastosować odpowiedni algorytm porządkujący. Istnieje wiele technik zapewniających odtwarzanie konsystentności informacji o stanie rozproszonych aplikacji, ale ogólnie można je podzielić na dwa rodzaje: aktywne i pasywne. Pierwsza technika, tzw. obrazów (ang. *snapshots*), polega na budowie stanu globalnego obserwowanej aplikacji tylko na życzenie procesu monitorującego. W tym podejściu program rozproszony musi być rozbudowany tak, aby rozumiał prośbę monitora o stworzenie stanu globalnego i umiał na nią odpowiednio zareagować. Odmienną grupę stanowią techniki pasywne, w których proces monitorujący jest bierny, natomiast program rozproszony jest rozszerzony o odpowiedni kod instrumentujący, który informuje monitor o każdym zdarzeniu zachodzącym w obliczeniu. Do tej grupy zalicza się metody wykorzystujące różnego rodzaju zegary: rzeczywiste, logiczne czy wektorowe. Wśród metod wykorzystujących zegary wektorowe można wyróżnić następujące algorytmy:

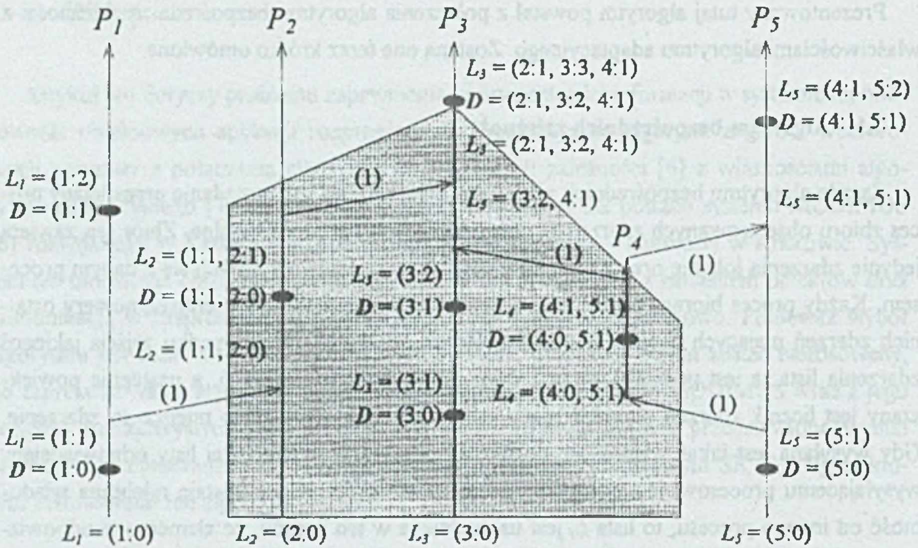
- algorytm Fidge'a (klasyczne zegary wektorowe) [3, 6],
- algorytm Zwaenepoela (bezpośrednich zależności) [6],
- algorytm Jarda, Jourdana (adaptacyjny) [6, 7].

Prezentowany tutaj algorytm powstał z połączenia algorytmu bezpośrednich zależności z właściwościami algorytmu adaptacyjnego. Zostaną one teraz krótko omówione.

2.1. Algorytm bezpośrednich zależności

Zasadą algorytmu bezpośrednich zależności Zwaenepoela jest posiadanie przez każdy proces zbioru obserwowanych zdarzeń D poprzedzających zdarzenia lokalne. Zbiór ten zawiera jedynie zdarzenia lokalne oraz z procesów bezpośrednio komunikujących się z danym procesem. Każdy proces biorący udział w obliczeniu utrzymuje listę L_i zawierającą numery ostatnich zdarzeń mających miejsce w poszczególnych procesach. W przypadku zajścia jakiegoś zdarzenia lista ta jest podstawiana pod zbiór obserwowanych zdarzeń, a następnie powiększany jest licznik zdarzeń odpowiadający procesowi, w którym miało miejsce to zdarzenie. Gdy wysyłana jest jakaś wiadomość do innego procesu, to element tej listy odpowiadający wysyłającemu procesowi jest dołączany do tej wiadomości. Kiedy zostaje odebrana wiadomość od innego procesu, to lista L_i jest uaktualniana w ten sposób, że elementowi odpowiadającemu procesowi wysyłającemu tę wiadomość przypisywane jest maksimum z jego starej wartości oraz wartości niesionej przez odebraną wiadomość. Wiadomość informująca przesyłana do monitora w momencie zajścia zdarzenia zawiera D. Na podstawie otrzymanych wiadomości informujących monitor odtwarza konsystentny obraz stanu obserwowanej aplikacji. Rysunek 1 przedstawia przykładowe obliczenie z zaimplementowanym algorytmem Zwaenepoela.

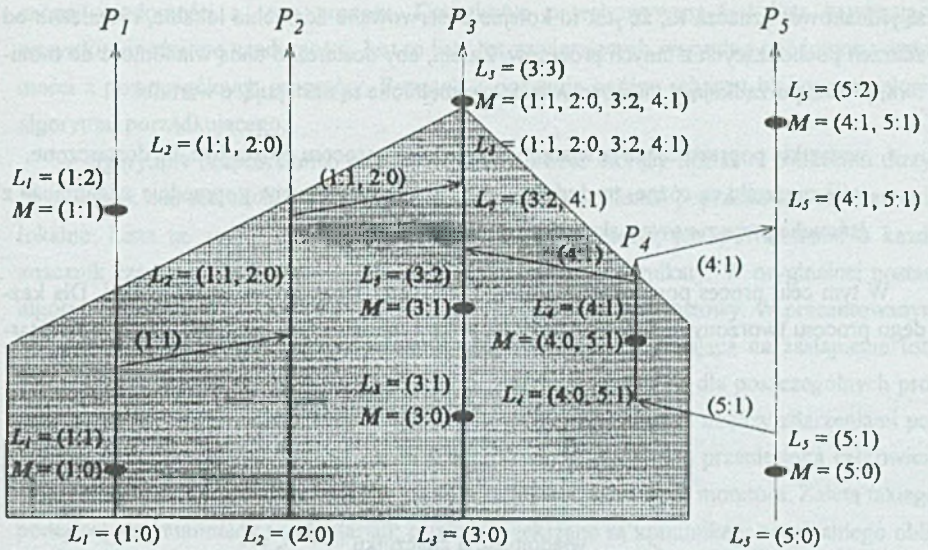
Algorytm bezpośrednich zależności charakteryzuje się minimalnym i stałym narzutem informacji dołączanej do każdej wiadomości (identyfikator procesu-nadawcy i wartość jego licznika zdarzeń lokalnych). Jednak aby algorytm ten mógł być stosowany, musi być przyjęte pewne ograniczenie na zbiór zdarzeń obserwowanych: w każdym procesie pomiędzy zdarzeniem odebrania wiadomości i zdarzeniem wysłania wiadomości musi być obserwowane przynajmniej jedno zdarzenie. Założenie takie jest nazywane NIVI (ang. *Non InVisible Interaction*). Potrzeba spełnienia tego założenia wynika stąd, że informacja o zależnościach przechodnich między zdarzeniami (czyli zależnościach między zdarzeniami z procesów nie komunikujących się ze sobą bezpośrednio) nie jest przechowywana w każdym procesie, ale jest rozproszona. Gdyby nie zaobserwowano zdarzenia między przyjęciem wiadomości a wysłaniem następnej, to zależności relacyjne powstałe dzięki tym dwóm wiadomościom zostałyby utracone.



Rys. 1. Znaczniki w algorytmie zależności bezpośrednich
Fig. 1. Stamps in indirect dependencies algorithm

2.2. Algorytm adaptacyjny

W algorytmie adaptacyjnym każdy proces posiada zbiór obserwowanych zdarzeń M spełniających relację pośredniej zależności (dwa zdarzenia spełniają tę relację, jeśli pomiędzy tymi zdarzeniami istnieje ścieżka komunikatów, która nie zawiera żadnego obserwowanego zdarzenia). Każdy proces biorący udział w obliczeniu utrzymuje listę L_i zawierającą numery ostatnich zdarzeń mających miejsce w poszczególnych procesach. W przypadku zajścia jakiegoś zdarzenia lista ta jest podstawiana pod zbiór obserwowanych zdarzeń, a następnie jest ona zmniejszana do pojedynczego elementu, odpowiadającego aktualnemu procesowi, w którym miało miejsce to obserwowane zdarzenie. Po zmniejszeniu rozmiaru listy powiększany jest licznik zdarzeń w elemencie, który pozostał na liście. Gdy wysyłana jest jakaś wiadomość do innego procesu, to lista L_i jest dołączana do tej wiadomości. Kiedy zostaje odebrana wiadomość od innego procesu, to lista jest uaktualniana w ten sposób, że każdemu elementowi listy przypisywane jest maksimum z jego starej wartości oraz odpowiadającej mu wartości znajdującej się w niesionej przez odebraną wiadomość liście. Wiadomość notyfikująca przesyłana do monitora w momencie zajścia zdarzenia zawiera M.



Rys. 2. Znaczniki w algorytmie adaptacyjnym
Fig. 2. Stamps in adaptive algorithm

Dzięki kasowaniu listy L_i narzut informacji dołączany do wiadomości rozrasta się jedynie przy wystąpieniu długich ścieżek zdarzeń komunikacyjnych nie zawierających zdarzeń obserwowanych. Nie musi być już spełniony warunek NIVI, a więc można obserwować dowolny zbiór zdarzeń w obliczeniu. Rysunek 2 przedstawia przykładowe obliczenie z zaimplementowanym algorytmem adaptacyjnym.

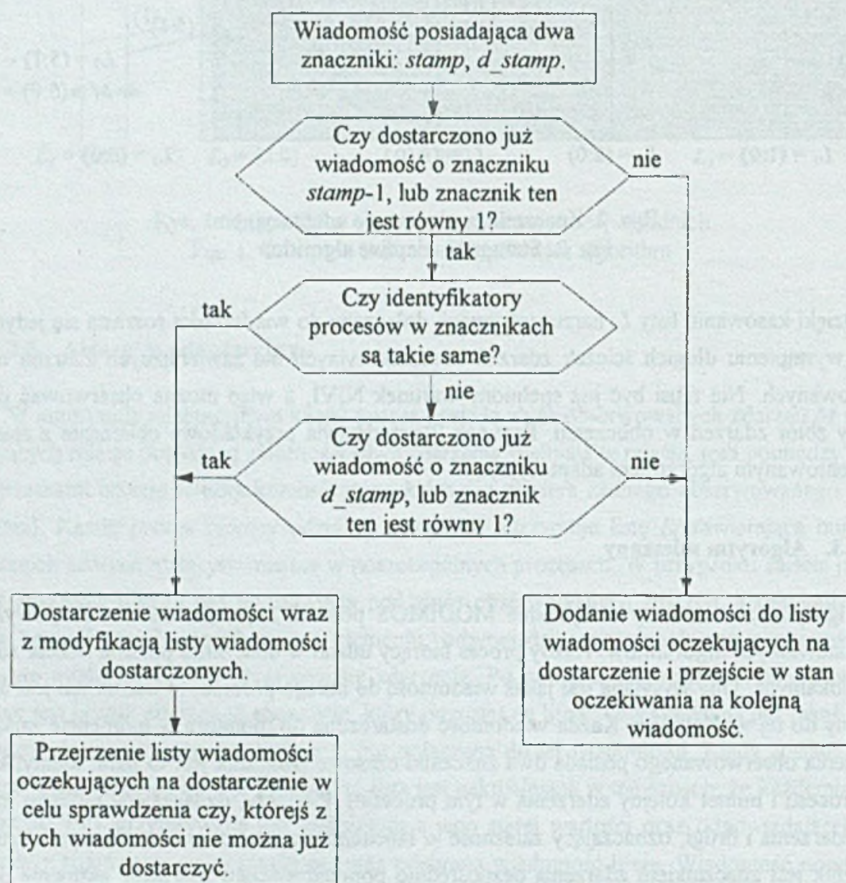
2.3. Algorytm mieszany

Algorytm zastosowany w systemie MODIMOS powstał z połączenia dwóch powyżej przedstawionych algorytmów. Każdy proces biorący udział w obliczeniu posiada licznik zdarzeń lokalnych. Gdy wysyłana jest jakaś wiadomość do innego procesu, to licznik ten jest dołączany do tej wiadomości. Każda wiadomość dostarczona do monitora w momencie zajścia zdarzenia obserwowanego posiada dwa znaczniki czasowe (znacznik jest to para: identyfikator procesu i numer kolejny zdarzenia w tym procesie). Pierwszy, oznaczający moment zajścia zdarzenia i drugi, oznaczający zależność w łańcuchu przyczynowo - skutkowym. Drugi znacznik jest znacznikiem zdarzenia bezpośrednio poprzedzającego zdarzenie aktualnie obserwowane (raportowane do monitora), np. znacznikiem zdarzenia wysłania wiadomości z innego procesu, gdy aktualnie raportowane jest odebranie tej wiadomości. Jeśli oba znaczniki

są jednakowe, oznacza to, że jest to kolejne obserwowane zdarzenie lokalne, niezależne od zdarzeń pochodzących z innych procesów. Zatem, aby dostarczyć daną wiadomość do monitora, proces porządkujący musi sprawdzić, czy spełnione są następujące warunki:

- wszystkie poprzedzające ją wiadomości z danego procesu muszą być już dostarczone,
- jeśli znaczniki są różne, to dodatkowo musi być dostarczona poprzednia wiadomość z łańcucha przyczynowo - skutkowego.

W tym celu proces porządkujący przechowuje listę dostarczonych wiadomości. Dla każdego procesu tworzony jest element tej listy, w którym pamiętany jest numer ostatnio dostar-



Rys. 3. Schemat blokowy działania proponowanego algorytmu porządkującego
Fig. 3. Block scheme of the proposed ordering algorithm

czonej wiadomości z tego procesu. Dodatkowo przechowywana jest lista zawierająca wszystkie opóźnione wiadomości. Jest to lista list zawierających wszystkie opóźnione wiadomości z poszczególnych procesów. Rysunek 3 pokazuje ogólny schemat blokowy działania algorytmu porządkującego.

W algorytmie bezpośrednich zależności każdy proces biorący udział w obliczeniu utrzymuje licznik zdarzeń zachodzących w tym procesie i listę zdarzeń poprzedzających zdarzenia lokalne. Lista ta, zwana też historią bezpośrednich zależności, jest powiększana o każdy znacznik czasowy niesiony przez odebrane przez proces komunikaty. W oryginalnej postaci algorytmu nigdy nie jest ona kasowana, a więc ma charakter przyrostowy. W prezentowanym algorytmie wykorzystano cechę algorytmu adaptacyjnego, pozwalającą na zastąpienie listy tylko jednym licznikiem. Odpowiedzialność za przechowywanie list dla poszczególnych procesów, jak również wnioskowanie na ich podstawie o zachodzącym między zdarzeniami porządku przyczynowo-skutkowym jest w proponowanym algorytmie przeniesiona całkowicie na proces porządkujący, dostarczający uporządkowane zdarzenia do monitora. Zaletą takiego podejścia jest minimalny i stały narzut, o jaki powiększane są komunikaty oryginalnego obliczenia. Ponadto bardzo upraszcza ono implementację algorytmu. W przypadku systemów heterogenicznych programowo [8], w których dołączanie znaczników czasowych odbywa się w różnych środowiskach programowania, posiadających różne charakterystyki, wymagania i możliwości, stały i określony narzut może mieć istotne znaczenie. Niestety, dla poprawnego działania algorytmu obserwowana aplikacja musi spełniać warunek NIVI, tak jak w przypadku oryginalnego algorytmu Zwaenepoela. Wiele systemów monitorowania, m. in. MODIMOS, spełnia jednak ten warunek.

3. Opis systemu MODIMOS

Przedstawiony powyżej algorytm został zastosowany w systemie monitorowania rozproszonych aplikacji obiektowych MODIMOS. Aby uzasadnić dobór algorytmu, pokrótce zostaną przedstawione główne cechy tego systemu; bardziej szczegółowo jest on omówiony w [8].

Celem systemu MODIMOS (*Managed Object-based Distributed Monitoring System*) jest umożliwienie obserwacji i wizualizacji zdarzeń związanych z istnieniem obiektów w rozproszonych systemach heterogenicznych programowo. Śledzone środowiska powinny spełniać wymaganie zgodności swego modelu obliczeniowego z Uniwersalnym Modelem Obliczeniowym opracowanym przez autorów systemu MODIMOS. Nie jest to zgodność w sensie identyczności, UMO został opracowany jako nadzbiór najpopularniejszych modeli obliczenio-

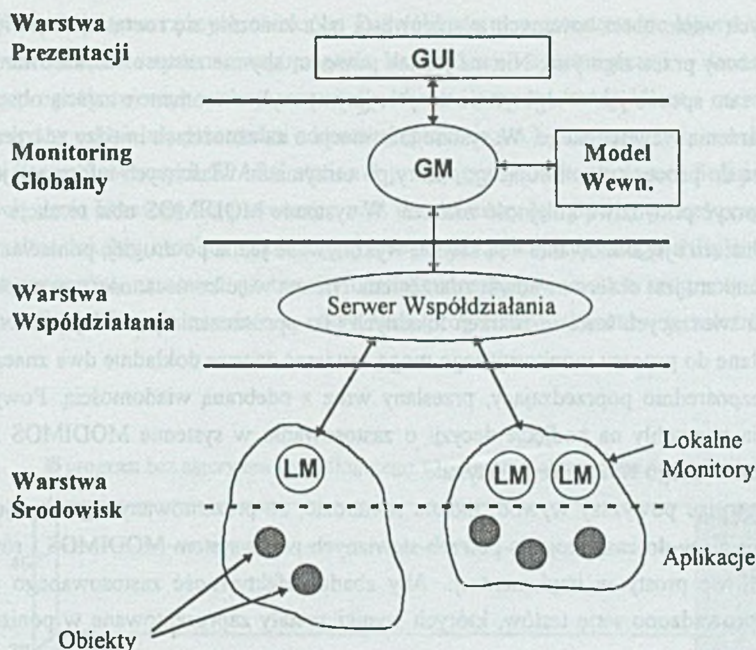
wych. To podejście pozwala założyć, że większość modeli różnych środowisk można będzie odwzorować na ten uniwersalny, abstrakcyjny model obliczeniowy. W UMO znalazły się następujące jednostki, tworzące logiczne warstwy aplikacji rozproszonej:

- środowisko programowania,
- kontener (przestrzeń adresowa dla obiektów, najczęściej proces s.o.),
- obiekt,
- interfejs,
- metoda (a ściślej aktywacja metody w obiekcie - serwerze),
- wywołanie (stan obiektu - klienta: wywołanie metody z interfejsu odległego obiektu).

Każdemu z powyższych elementów przyporządkowana jest para komplementarnych zdarzeń: utworzenie i zniszczenie danego elementu. Informacja o zajściu tych zdarzeń powinna być dostarczana do monitora globalnego. MODIMOS może pracować w dwóch trybach: *off-line*, umożliwiającym analizę pliku ze śladem działania aplikacji, oraz *on-line*, kiedy system nasłuchuje zdarzeń z sieci i dzięki temu praca monitorowanej aplikacji analizowana jest na bieżąco.

Rysunek 4 przedstawia wielowarstwową architekturę systemu MODIMOS umożliwiającą dostarczanie komunikatów notyfikujących o zdarzeniach zachodzących w monitorowanych środowiskach do globalnego monitora. Najniższa warstwa, Warstwa Środowisk jest implementowana poprzez zinstrumentowanie kodu aplikacji napisanych w różnych środowiskach programowania przez specjalne preprocesory. Warstwa Współdziałania ma na celu zapewnienie platformy współdziałania heterogenicznych programowo komponentów systemu MODIMOS, zwłaszcza w przypadku jego pracy w trybie *on-line*. GM (Globalny Monitor) przechowuje zdarzenia nadchodzące z całego systemu monitorowanego w specjalnej bazie danych zwanej Modelem Wewnętrznym. Monitor Globalny współpracuje z Graficznym Interfejsem Użytkownika (GUI) umożliwiającym prezentację danych.

W każdym z możliwych trybów pracy systemu MODIMOS (*on-line* i *off-line*) istnieje niebezpieczeństwo otrzymania wiadomości o zdarzeniach w kolejności innej, niż one zaszły. Niebezpieczeństwo to wynika z faktu, że śledzone obliczenia odbywają się w środowiskach rozproszonych i komunikują się poprzez sieć potencjalnie stosując zróżnicowane protokoły komunikacyjne. Niezbędne jest więc zastosowanie jakiegoś mechanizmu porządkowania tych wiadomości. Mechanizm ten może znajdować się w każdym ze środowisk i działać pomiędzy jego obiektami i Monitorem Lokalnym, lub pomiędzy obiektami monitorowanymi i Monitorem Globalnym.



Rys. 4. Wielowarstwowa architektura systemu MODIMOS
Fig. 4. Multilayer architecture of MODIMOS system

3.1. Cechy mające wpływ na dobór algorytmu

W systemie MODIMOS obserwowane są zdarzenia zachodzące w obiektowych systemach rozproszonych, związane z istnieniem kontenerów, obiektów, wywołaniem i obsługą metod. Należy zauważyć, że właściwie każde z tych zdarzeń może być zinterpretowane jako wysłanie lub odebranie wiadomości. Gdy obiekt jest tworzony, to jakiś inny obiekt musi wykonać odpowiednią instrukcję tworzenia. Podobnie rzecz się ma z tworzeniem kontenera czy też niszczeniem tych elementów programu. Nawet jeśli obiekt jest tworzony tylko w celu wykonania pewnego zadania i potem sam powinien się usunąć, to można tę sytuację zinterpretować jako wysłanie polecenia zniszczenia obiektu przez ten właśnie obiekt. Tak więc można uznać, że wszystkie śledzone przez monitor zdarzenia są zdarzeniami nadania komunikatu i odbioru komunikatu. W takim systemie spełniona jest zasada NIVI (*Non InVisible Interaction*). W związku z tym można dla niego zastosować algorytm bezpośrednich zależności. Algorytm ten wymaga, aby każdy proces biorący udział w obliczeniu utrzymywał listę zawierającą wszystkie zdarzenia poprzedzające zdarzenia lokalne. W przypadku długich obliczeń

(zawierających wiele obserwowanych zdarzeń) lista taka znacznie się rozrasta, co zwiększa narzut wnoszony przez algorytm. Nie ma jednak powodu, aby nie zastosować kasowania tej listy, w taki sam sposób jak w algorytmie adaptacyjnym, czyli w momencie zajścia obserwowalnego zdarzenia wewnętrznego. Wszystkie informacje o zależnościach między zdarzeniami przesyłane są do procesu monitorującego, który po otrzymaniu właściwych informacji jest w stanie odtworzyć prawdziwą kolejność zdarzeń. W systemie MODIMOS obie te akcje - powiększenie historii i jej skasowanie - są zawsze wykonywane jedna po drugiej, ponieważ odebranie komunikatu jest obserwowalnym zdarzeniem. Nie ma więc konieczności utrzymywania listy zdarzeń tworzących historię zdarzeń lokalnych. To uproszczenie powoduje, że wiadomości wysyłane do procesu monitorującego mogą zawierać zawsze dokładnie dwa znaczniki: lokalny i bezpośrednio poprzedzający, przesłany wraz z odebraną wiadomością. Powyższe spostrzeżenia pozwoliły na podjęcie decyzji o zastosowaniu w systemie MODIMOS algorytmu prezentowanego w niniejszym artykule.

Podsumowując powyższy wywód, można stwierdzić, że prezentowany algorytm jest w pełni wystarczający do zaspokojenia potrzeb stawianych przez system MODIMOS i równocześnie możliwie prosty w implementacji. Aby zbadać efektywność zastosowanego algorytmu, przeprowadzono serię testów, których wyniki zostały zaprezentowane w poniższym rozdziale.

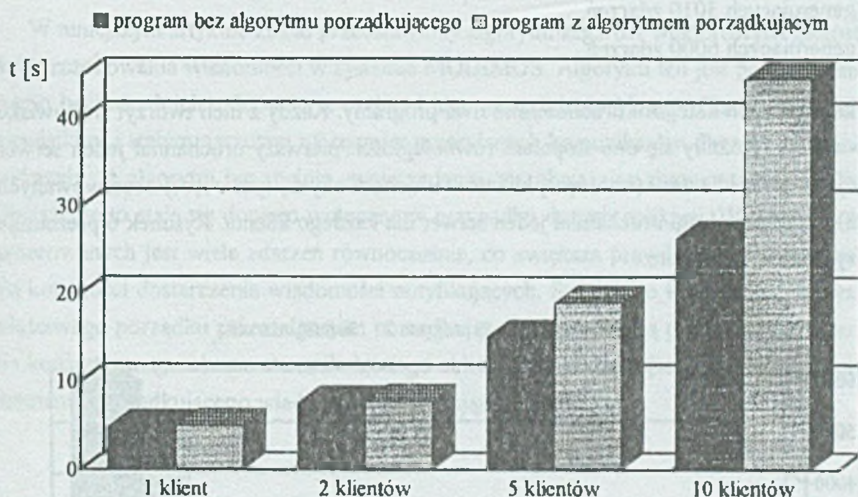
4. Wyniki testów

Przeprowadzone testy miały za zadanie sprawdzenie efektywności oraz przydatności wybranego algorytmu porządkującego. Do przeprowadzenia testów wybrane zostało obiektowe, rozproszone środowisko programowania SR. Dostarcza ono wiele mechanizmów komunikacji i synchronizacji procesów oraz daje możliwość tworzenia aplikacji rozproszonych, działających na wielu stacjach roboczych. Nie bez znaczenia była prostota programowania w tym języku, dostępność literatury, jak i jego dobra znajomość przez autorów. Opracowano preprocesor instrumentujący aplikacje w SR oraz lokalny monitor, zgodnie z zasadami i modelem monitorowania w systemie MODIMOS.

4.1. Pomiar narzutu wnoszonego przez algorytm

Pierwszy test polegał na pomiarze czasu istnienia obiektu klient w rozproszonym środowisku typu klient - serwer. Został on przeprowadzony dla dwóch kategorii programów: bez algorytmu porządkującego oraz z omawianym wcześniej algorytmem. W każdej kategorii znajdowały się cztery podobne programy, o niewielkiej złożoności obliczeniowej, które

różniły się ilością tworzonych obiektów. Pozwoliło to na przetestowanie algorytmu w zależności od obciążenia systemu wspomagania komunikacji i synchronizacji w środowisku SR. Każdy program tworzył osobny serwer dla klienta, ponadto każdy klient i serwer uruchomiony był w oddzielnej przestrzeni adresowej (procesie s.o.). Programy były uruchomione na trzech stacjach roboczych SPARC station 5 pracujących pod kontrolą s.o. Solaris 2.4, połączonych siecią Ethernet. Na pierwszej z nich uruchamiany był główny proces programu oraz opcjonalnie lokalny monitor (zawierający proces porządkujący), druga służyła do uruchomienia serwerów, natomiast na trzeciej uruchomione były wszystkie programy typu klient. Rysunek 5 przedstawia uśrednione czasy życia programów - klientów w zależności od ich ilości.



Rys. 5. Średnie czasy życia programów typu klient
Fig. 5. Mean time of clients life

Porównując programy bez algorytmu porządkującego z programami wyposażonymi w ten mechanizm można zauważyć, że w przypadku niewielkiej liczby uruchomionych procesów narzut wnoszony przez mechanizm porządkujący jest niewielki (około 5%). Wzrasta on dopiero w przypadku uruchomienia dużej liczby klientów. Fakt ten można wytłumaczyć zwiększeniem się równoległości programu, a co za tym idzie - zwiększeniem prawdopodobieństwa zakłócenia kolejności dostarczanych informacji oraz możliwym wydłużeniem działania podsystemu komunikacji i synchronizacji środowiska SR. Dokonano również pomiarów narzutu

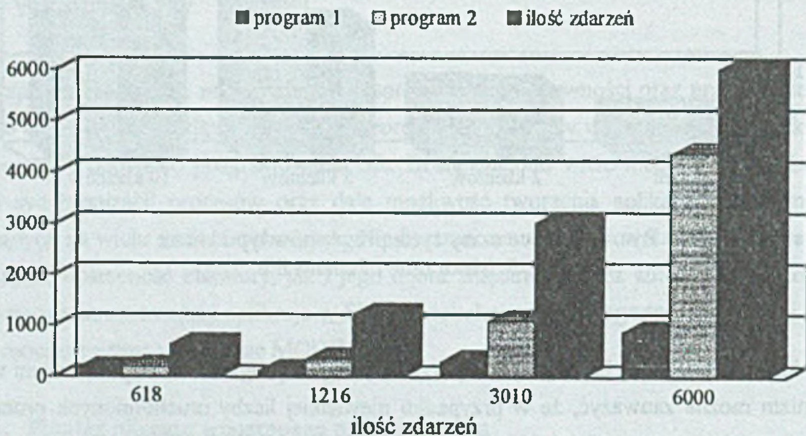
na zdalne wywołania metod. Narzut ten jest stały i mieści się w granicach od 50 do 100 milisekund na wywołanie, w zależności od ilości powtórzeń operacji. W przypadku obliczeń o znaczącej złożoności obliczeniowej procentowy udział tego narzutu będzie zatem niewielki.

4.2. Pomiar ilości wiadomości opóźnianych

Kolejny test miał wykazać zasadność stosowania mechanizmu porządkującego wiadomości notyfikujące. Polegał on na przeprowadzeniu pomiaru ilości opóźnionych przez algorytm wiadomości dla czterech kategorii programów:

- generujących 618 zdarzeń,
- generujących 1216 zdarzeń,
- generujących 3010 zdarzeń,
- generujących 6000 zdarzeń.

W każdej z tych kategorii uruchomiono dwa programy. Każdy z nich tworzył środowisko klient - serwer. Różniły się one stopniem równoległości: pierwszy uruchamiał jeden serwer obsługujący wielu klientów (im więcej klientów uruchomionych, tym więcej obserwowanych zdarzeń), natomiast drugi uruchamiał jeden serwer dla każdego klienta. Rysunek 6 prezentuje dane uzyskane w tym teście.



Rys. 6. Ilość wiadomości opóźnianych w stosunku do wszystkich wiadomości
Fig. 6. Number of delayed messages in comparison with total number of messages

Jak widać, nawet w przypadku występowania niewielkiej ilości obserwowanych zdarzeń zachodzi potrzeba stosowania algorytmu porządkującego. Ogromne znaczenie dla działania tego mechanizmu ma stopień równoległości monitorowanego programu. Przy bardziej rozbudowanych obliczeniach różnica ilości wiadomości opóźnianych przez algorytm porządkujący dla dwóch rodzajów programów jest kilkakrotna. Przy wysokim obciążeniu systemu stanowi już bardzo wysoki procent wszystkich zaobserwowanych zdarzeń. Należy jednak podkreślić, że bez zastosowania takiego algorytmu informacje uzyskane z obserwacji takiego systemu byłyby niekonsystentne, czyli praktycznie nieprzydatne.

5. Podsumowanie

W niniejszym artykule został przedstawiony algorytm zegarów wektorowych zastosowany do porządkowania wiadomości w systemie MODIMOS. Algorytm ten jest połączeniem algorytmu bezpośrednich zależności z cechami algorytmu adaptacyjnego. Charakteryzuje się on niewielkim i stałym narzutem na rozmiar przesyłanych komunikatów. Przeprowadzone testy wykazały, iż algorytm ten spełnia swoje zadanie, nie obciążając zbytnio samego obliczenia. Obciążenie to staje się dopiero widoczne w przypadku dużych aplikacji. W takim przypadku generowanych jest wiele zdarzeń równocześnie, co zwiększa prawdopodobieństwo zakłócenia kolejności dostarczenia wiadomości notyfikujących. Stwarza to konieczność odtwarzania właściwego porządku przez algorytm porządkujący będący częścią programu. Dla otrzymania konsystentnego obrazu stanu śledzonego obliczenia konieczne jest jednak stosowanie mechanizmu porządkującego wiadomości notyfikujące.

LITERATURA

[1] Babaoglu Ö., Marzullo K.: Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. Distributed Systems, ACM Press, Frontier Series (S. Mullender Ed.), 1994, s. 55-93.

[2] Chandy K., Lamport L.: Distributed Snapshots: determining global states of distributed systems. ACM TOCS, 1985, t. 3, nr 1, s. 63-75.

[3] Fidge C.: Logical time in Distributed Computing Systems. Computer, Aug. 1991, t. 24, nr 8, s. 28-33.

[4] Fronentin E., Raynal M.: Characterising and Detecting The Set of Global States Seen by all Observers of a Distributed Computation. Proceedings of the 15th International Conference on Distributed Computing Systems, IEEE Computer Society Press, 1995.

[5] Lamport L.: Time, Clocks, and the Ordering of Events in a Distributed System. Comm. ACM, Jul. 1978, t. 21, nr 7, s. 558-564.

[6] Raynal M., Singhal M.: Logical time: Capturing Causality in Distributed Systems. Computer, Feb. 1996, t. 29, nr 2, s. 49-56.

[7] Jard C., Jourdan G.-C.: Dependency Tracking and Filtering in Distributed Computations. ACM Symposium on Principles of Distributed Computing, ACM Press, New York 1994.

[8] Szymaszek J., Laurentowski A., Zieliński K.: System monitorowania heterogenicznych programowo obiektowych środowisk rozproszonych. ZN Pol. Śl. s. Informatyka z. 30, Gliwice 1996, s. 193-206.

[9] Andrews G., Olsson R.: The SR Programming Language: Concurrency in practice. Benjamin/Cummings Publishing Company, 1992.

[10] Laurentowski A., Zieliński K.: SR - obiektowo zorientowany język programowania rozproszonego. Informatyka, 1993, nr 6, s. 15-19.

Recenzent: Doc. dr hab. inż. Tadeusz Czachórski

Wpłynęło do Redakcji 22 listopada 1996 r.

Abstract

This paper presents a novel implementation of vector clocks algorithm. The algorithm is based on the direct dependencies algorithm with features of adaptive algorithm. In these two algorithms each process has a list to store events preceding the currently observed event. In the direct dependencies algorithm this list contains all the events directly preceding the current event - this situation is shown in Figure 1. To use this algorithm the NIVI (*Non InVisible Interaction*) condition must be assured in the monitored application. It means that in every process between *receive* and *send* event there must be at least one observable event. This algorithm provides a small overhead added to the original computation. In the second algorithm all events directly preceding the currently observed event are stored in the previously mentioned list. This situation is shown on a figure 2. In this algorithm the NIVI condition

doesn't have to be assured, but intrusion is greater than in a previous algorithm. In the algorithm presented in this paper each process doesn't have to store a list containing information about the preceding events, but has got only one counter of local events. This counter can be considered as a one-element list, which is being re-set after each observable event (similarly to resetting the list in adaptive algorithm). The rest of the proposed algorithm is similar to the Zwaenepoel's algorithm, each process adds this counter to the messages sent to other processes. Figure 3 shows the algorithm used by the ordering process in the monitor. Naturally, in this algorithm like in the indirect dependencies algorithm, the NIVI condition must be granted. This algorithm was used in the MODIMOS system. The MODIMOS is a multilayer system of monitoring distributed object applications, its architecture is shown on a figure 4. In this system all observable events satisfy the NIVI condition, enabling usage the proposed algorithm. To check overhead and necessity of using this algorithm two test were made. The first test measured average clients lifetime in a distributed client - server environment. Results of this test are shown on figure 5. The second test measured number of delayed messages in a similar distributed environment, with results shown on figure 6. The tests proved that the proposed ordering algorithm has a acceptable level of overhead, but even in a small environment number of inverted messages is high and usage of an ordering algorithm is therefore indispensable.