ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1998
Seria: INFORMATYKA z. 34 Nr kol. 1381

Henryk KRAWCZYK, Jerzy PROFICZ
Politechnika Gdańska, Wydział Elektroniki, Telekomunikacji i Informatyki

SYMULACYJNA METODA ANALIZY ZALEŻNOŚCI CZASOWYCH W APLIKACJACH ROZPROSZONYCH

Streszczenie. Przedstawiono symulator pracujący w środowisku PVM analizujący zachowanie się wykonywanej aplikacji w zależności od zmian czasu komunikacji. Umożliwia on wykrycie anomalii uwarunkowanych czasowo (wyścigi, zakleszczenia) już na etapie projektowania architektury aplikacji rozproszonej.

A SIMULATION METHOD OF TIME DEPENDENCES ANALYSIS IN DISTRIBUTED APPLICATIONS

Summary. We perform a simulator working in PVM environment, which analyses behaviour of the application according to the change of communication time. It enables time dependent anomalies detection (races, deadlocks), already in the designing of the application architecture.

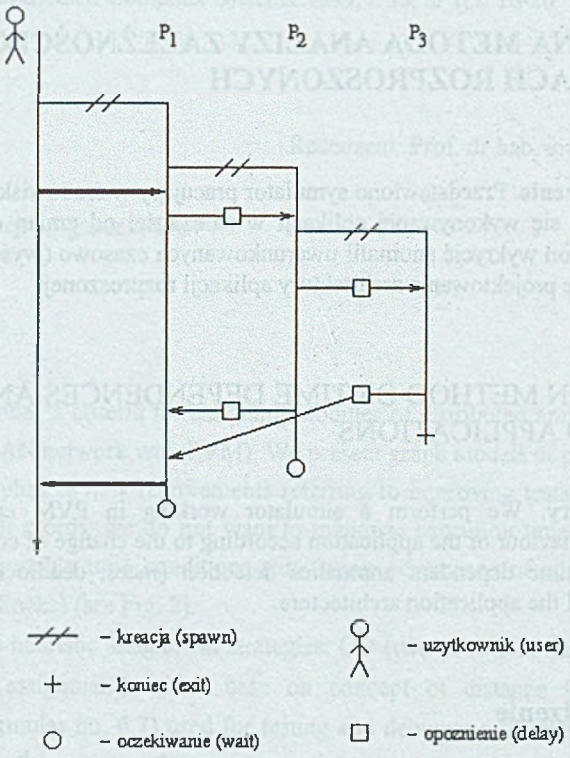
1. Wprowadzenie

Przetwarzanie sekwencyjne i współbieżne realizowane są w środowisku jednoprocesorowym, zaś przetwarzanie równoległe i rozproszone w środowiskach wieloprocesorowym lub wielokomputerowym, na ogół sieciowym. W związku z powyższym ich schematy wykonania różnią się znacznie. Aplikacja sekwencyjna realizowana jest zwykle według następujących kroków:

- wprowadzenie danych wejściowych,
- wykonanie programu z ewentualnym wprowadzeniem poprawnych danych i wyprowadzeniem wyników.

Aplikacje rozproszone (współbieżna) działają już według innego schematu:

- inicjalizacja pracy poprzez kreowanie procesów, rozpoczęcie funkcjonowania z zadanymi danymi wejściowymi,
- realizacja właściwych funkcji na danych wraz z możliwością powtórzenia procedury tworzenia nowych procesów,
- zakończenie wykonania usługi, zwrócenie wyników, zakończenie życia procesów lub przejście do stanu oczekiwania na nowe dane czy określone zdarzenia.



Rys. 1. Przykładowe wykonanie aplikacji rozproszonej
Fig. 1. An example of distributed application execution

Porównując funkcjonowanie tych dwóch typów aplikacji łatwo zauważyć, że różnią się one przede wszystkim możliwościami komunikowania się. Na rys. 1 zaprezentowano sposób opisu funkcjonowania aplikacji na poziomie kreowania i komunikowania się procesów. Ważną cechą jest tutaj niedeterminizm, polegający na tym, że konkretne obliczenia zależą od kolejności wystąpienia zdarzeń komunikacyjnych [6]. Kolejność ta może zależeć od architektury systemu komputerowego jak i danych wejściowych, w przypadku aplikacji

rozproszonych istotny staje się również czas przesyłania danych z otoczenia bądź modułów pracujących na różnych komputerach. Często od tych czasów zależy kolejność wykonywania zdarzeń komunikacyjnych, czyli występowanie wyścigów.

Pojawia się więc potrzeba analizy tego typu sytuacji zarówno na etapie projektowania, jak i testowania aplikacji rozproszonej. Rozproszone środowiska programowania dostarczają dużą dowolność wykonania aplikacji. Zatem konieczne jest opracowanie odpowiedniego narzędzia, które umożliwi analizę zachowania się aplikacji rozproszonych. W pracy zaproponowano symulator zależności czasowych (SZAC), który analizuje aplikacje na trzech poziomach:

- systemowym, sprowadzającym się do rejestracji drzewa kreowanych procesów,
- komunikacyjnym sprowadzającym się do rejestracji zdarzeń komunikacyjnych,
- decyzyjnym, uwzględniającym powyższe dwie cechy, jak i elementy decyzyjne istotne z punktu widzenia przetwarzania sekwencyjnego.

Symulator ten umożliwia rejestrację zachowania aplikacji rozproszonej dla zmieniających się czasów komunikacji. W rozdziale 2 znajduje się dokładny opis modelu przetwarzania rozproszonego. W rozdziale 3 opisano ideę i architekturę symulatora, natomiast w rozdziale 4 przedstawiono opis graficznego pakietu wizualizacji obliczeń w systemach rozproszonych, umożliwiającego wykrycie anomalii uwarunkowanych czasowo. Wnioski końcowe zostały umieszczone w rozdziale 5.

2. Model obliczeń rozproszonych

W proponowanym modelu aplikacja jest zbiorem procesów $P = \{p_i | i = 1, 2, \ldots, N\}$ komunikujących się za pomocą przesyłania wiadomości, jest to jedyny sposób przesyłania danych i synchronizacji między procesami. Środowisko, w którym wykonuje się ta aplikacja, dostarcza mechanizmów ich wymiany. Każdy proces posiada kolejkę Q, w której znajduje się ciąg komunikatów. Środowisko umożliwia dowolnemu procesowi dopisanie własnej wiadomości na koniec dowolnej kolejki, jak również wyjęcie dowolnego komunikatu z własnej kolejki komunikatów (przy czym kryterium wyboru może być zależne od informacji w nim zawartych).

Operacje przesyłania wiadomości mogą być blokujące, czyli takie, które oddają sterowanie do procesu dopiero po zajściu zdarzenia komunikacyjnego, albo nie blokujące. Zdarzenie odebrania wiadomości polega na wyjęciu określonej wiadomości z kolejki i w opisywanym modelu jest ono blokujące. Zdarzenie wysłania polega na wstawieniu wiadomości do kolejki odbiorcy i zostaje zakończone w momencie pobrania jej z tej kolejki.

To zdarzenie jest nie blokujące, a sterowanie jest zwracane zaraz po przekazaniu do środowiska odpowiednich parametrów. Zakłada się również, że środowisko rozproszone gwarantuje zachowanie kolejności wysyłanych wiadomości między dwoma dowolnymi procesami. Zdarzenia są powiązane relacją dopasowania: zdarzenie wysłania s jest dopasowane do zdarzenia odbioru r wtedy i tylko wtedy, gdy wiadomość towarzysząca zdarzeniu s została odebrana przez zdarzenie r. Relacja ta jest symetryczna.

Wykonaniem aplikacji jest zbiór zdarzeń komunikacyjnych uporządkowanych relacją *HB* (*ang. happened before*). Relacja ta została zdefiniowana w [6] przez L. Lamporta i określa logiczną kolejność występowania zdarzeń komunikacyjnych w aplikacji. Kolejność logiczna implikuje kolejność fizyczną (według czasu), ale nie na odwrót.

W tak zdefiniowanym środowisku niedeterminizm czasowy może się manifestować jedynie poprzez wyścig, czyli możliwość dopasowania się co najmniej dwóch różnych akcji nadawczych do jednej akcji odbiorczej [1], lub równoważnie, prowadzi do otrzymania co najmniej dwóch różnych wykonań aplikacji dla jednego zestawu danych. Należy zwrócić uwagę na to, że sam wyścig nie jest jeszcze błędem, można nawet wyróżnić kategorię wyścigów zamierzonych, które są wprowadzane przez projektanta w celu zwiększenia wydajności, czy wręcz są częścią implementowanego algorytmu.

Błąd czasowy definiuje się jako błąd, który pojawia się dla pewnych danych przypadkowo, co oznacza możliwość jego nieobecności w kolejnym wykonaniu aplikacji dla tych samych danych. Taki błąd może ujawnić się jedynie w przypadku wystąpienia dwóch różnych wykonań (przy tym samym komplecie danych) i jest spowodowany istnieniem wyścigu. Z kolei, najczęstszą przyczyną pojawienia się wyścigu jest zmiana zależności czasowych w środowisku, tzn. zmiana czasu, jaki jest potrzebny do wstawienia wiadomości do kolejki lub czasu wykonywania obliczeń.

Przedstawiony model umożliwia opracowanie prostego mechanizmu zadawania odpowiednich czasów komunikacji pomiędzy procesami. Manipulacja wartościami tych czasów powoduje zmianę zależności czasowych występujących w badanej aplikacji, a w konsekwencji może wpłynąć na wystąpienie wyścigów i w rezultacie umożliwi zaobserwowanie anomalii w wykonaniu aplikacji.

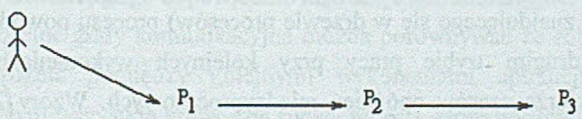
Przyjmując, że kwadratowa macierz opóźnień (1) zawiera zadawane opóźnienia pomiędzy procesami. Wartość t_{ij} oznacza czas, w jakim pojedyncza wiadomość ma być dostarczona z procesu i do procesu j. Symulator powoduje, że czasy zdefiniowane przez użytkownika zostają przestrzegane w danej aplikacji podczas jej wykonania.

$$T = \begin{bmatrix} t_{11} & t_{12} & \cdots & t_{1N} \\ t_{21} & t_{22} & & t_{2N} \\ \vdots & & \ddots & \vdots \\ t_{N1} & t_{N1} & \cdots & t_{NN} \end{bmatrix} \quad (1)$$

Proponowany model uwzględnia dynamiczne tworzenie się procesów, co powoduje pojawienie się dodatkowego problemu: sposobu identyfikacji procesów w kolejnych wykonaniach aplikacji. Wyróżniamy cztery klasy aplikacji, których podział został przeprowadzony według sposobu, w jaki tworzą one procesy:

- wszystkie procesy są tworzone podczas inicjalizacji,
- liczba procesów jest stała dla każdego wykonania,
- liczba procesów zależy jedynie od danych wejściowych,
- liczba procesów zależy od zależności czasowych lub danych wejściowych.

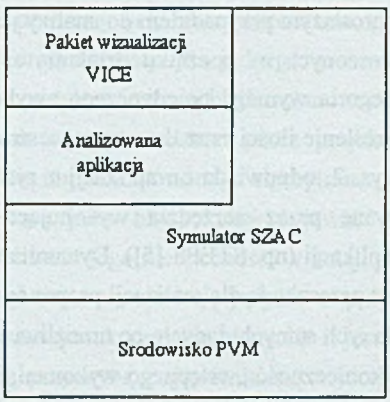
Łatwo zauważyć, że najprostszym przypadkiem do analizy jest pierwsza grupa. Określona a priori ilość procesów tworzonych na początku działania aplikacji umożliwia łatwą ich identyfikację. Następna kategoria wymaga pojedynczego „wykonania wstępnego", podczas którego byłoby możliwe określenie ilości oraz drzewa procesów. Przykład drzewa procesów został zaprezentowany na rys. 2, odpowiada on aplikacji z rys.1. Aplikacje należące do tej klasy mogą być analizowane przez narzędzia wykonujące analizę statyczną zdarzeń komunikacyjnych badanej aplikacji (np. STEPS [5]). Dynamiczne przydzielanie procesów w zależności od danych nie jest przeszkodą dla realizacji proponowanego symulatora, wykonuje on kolejne eksperymenty dla tych samych danych, co umożliwia detekcję błędów czasowych, jedyną niedogodnością jest konieczność „wstępnego wykonania" aplikacji dla każdej zmiany danych. Ostatnią klasą w rozważanej klasyfikacji są aplikacje, w których ilość procesów zależy nie tylko od danych, ale także od zależności czasowych. Proponowany symulator nie jest w stanie analizować tego typu aplikacji, gdyż oprócz kontrolowanego zadawania opóźnień pomiędzy procesami, jest konieczna również ich dokładna identyfikacja, czyli znajomość drzewa procesów, które w tym przypadku może się zmieniać wraz z każdym wykonaniem.



Rys. 2. Przykład drzewa procesów aplikacji z rys. 1
Fig. 2. An example of processes' tree of the application taken from Fig. 1

3. Architektura symulatora aplikacji rozproszonych w środowisku PVM

Symulator SZAC (Symulator ZAleżności Czasowych) został zaimplementowany w języku C i PVM [2]. Środowisko to zostało wybrane ze względu na właściwości zbliżone do prezentowanego modelu obliczeń rozproszonych (rozdział 2). Architekturę symulatora SZAC przedstawiono rys. 3. Bezpośrednio współpracuje on ze środowiskiem PVM, analizując aplikację oraz z pakietem wizualizacji VICE [4] (*ang. a package VIsualising Communications Events*), jak również wymaga bezpośredniej interakcji z użytkownikiem. Od strony aplikacji symulator zastępuje funkcje PVM, przez co możliwa jest rejestracja wszystkich zdarzeń komunikacyjnych oraz drzewa procesów.



Rys. 3. Środowisko symulatora aplikacji w środowisku PVM
Fig. 3. The environment of the PVM application simulator

Ze względu na przyjętą klasę analizowanych aplikacji symulator pracuje w dwóch trybach: rejestracji drzewa procesów (tryb wykorzystywany przy wprowadzeniu nowych danych) oraz rejestracji zdarzeń komunikacyjnych (właściwa praca symulatora). Wystąpienie dodatkowego (nie znajdującego się w drzewie procesów) procesu powoduje wygenerowania ostrzeżenia. W drugim trybie pracy przy kolejnych wykonaniach należy zadawać symulatorowi macierze czasów opóźnień międzyprocesowych. Wzory (2) i (3) przedstawiają przykładowe macierze opóźnień dla aplikacji z rys. 1.

$$T_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{2}$$

$$T_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad (3)$$

W powyższych macierzach „1" oznacza długi czas przesyłania wiadomości pomiędzy odpowiednimi procesami modelując przesyłanie wiadomości przez sieć, natomiast „0" przesyłanie szybkie bez opóźnień, reprezentuje komunikację międzyprocesową na jednym węźle. Właściwe wartości tych czasów zależą od konkretnego systemu (jego szybkości przesyłania komunikatów). Użytkownik znając te czasy, lub zmieniając je w dowolny sposób, może obserwować różne zachowania się aplikacji dla tych samych danych wejściowych.

4. Wizualizacja i detekcja wyścigów

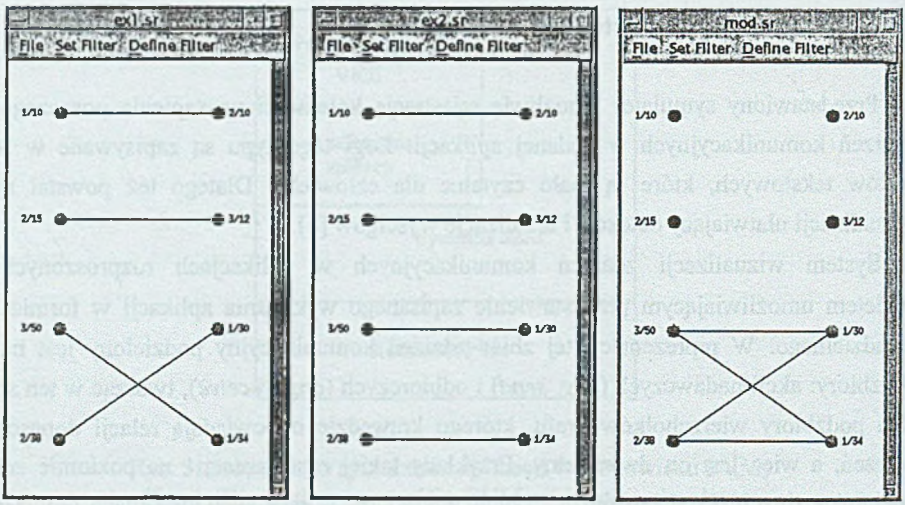
Przedstawiony symulator umożliwia rejestrację kolejności wystąpienia poszczególnych zdarzeń komunikacyjnych w badanej aplikacji. Logi tego typu są zapisywane w formie plików tekstowych, które są mało czytelne dla człowieka. Dlatego też powstał system wizualizacji ułatwiający detekcję i lokalizację wyścigów [4].

System wizualizacji zdarzeń komunikacyjnych w aplikacjach rozproszonych jest pakietem umożliwiającym przedstawienie zapisanego wykonania aplikacji w formie grafu dwudzielnego. W reprezentacji tej zbiór zdarzeń komunikacyjny podzielony jest na dwa podzbiory: akcji nadawczych (*ang. send*) i odbiorczych (*ang. receive*), tworząc w ten sposób dwa podzbiory wierzchołków grafu, którego krawędzie odpowiadają relacji dopasowania zdarzeń, a więc jest on dwudzielny. Przykłady takiej reprezentacji na poziomie zdarzeń komunikacyjnych przedstawiono na rys. 4. Podobnie jak w innych sposobach reprezentacji zdarzenia są etykietowane. Etykieta składa się z identyfikatora procesu oraz numeru linii w kodzie źródłowym, w którym znajdują się odpowiednie instrukcje komunikacyjne. Wskazując (myszką) na odpowiedni wierzchołek, otrzymamy okno z fragmentem programu źródłowego procesu zawierającego odpowiednie instrukcje komunikacyjne.

Tworzone dwudzielne grafy komunikacyjne można porównywać ze sobą. Pozwala to na łatwą lokalizację różnic pomiędzy kolejnymi wykonaniami aplikacji. Podstawowym operatorem znajdującym wszystkie krawędzie (wiadomości), które wystąpiły w jednym logu, a nie były obecne w innym, jest operator *modulo*. Innymi również przydatnymi operatorami, używanymi podczas wyszukiwania wyścigów są: *and*, (znajdujący część wspólną zbiorów krawędzi) oraz *or* (wykonujący sumę zbiorów krawędzi). Dodatkowym udogodnieniem

pomagającym ocenić wielkość różnic pomiędzy grafami jest operator *Coveradge* określający stosunek ilości wierzchołków (krawędzi) dwóch analizowanych grafów.

Aplikacja przedstawiona na rys. 1 została przeanalizowana przez symulator dla przykładowych macierzy (wzory 2, 3). Grafy dwudzielne odczytane z logów zostały przedstawione na rys. 4. Łatwo zauważyć, że zmieniła się kolejność dopasowania zdarzeń komunikacyjnych w procesie 1, gdyż odbiór wiadomości, której wysłanie związano ze zdarzeniem o etykiecie 3/50, odbył się przy różnych zdarzeniach odbiorczych. Jednak w większych aplikacjach nie jest to takie proste, dlatego też można wykorzystać operator *modulo* do wyróżnienia różnic miedzy grafami. Otrzymany wówczas dwa typy wierzchołków ciemniejsze (czerwone), czyli te, które nie mają żadnych krawędzi, i jaśniejsze (zielone), mające dopasowane odpowiadające im zdarzenia.



Rys. 4. Wizualizacja logów dla macierzy ze wzorów (2) i (3) oraz ich porównanie za pomocą operatora *modulo*

Fig. 4. Visualisation of the application logs for the matrices from formulas (2) and (3), and their comparison using *modulo* operator

Możliwość wystąpienia wyścigu jest związana z jaśniejszymi wierzchołkami, których stopień jest większy od 0.

5. Uwagi końcowe

Przedstawiono prosty symulator do oceny zależności czasowych wybranych aplikacji rozproszonych. Symulator wykonany w C+PVM może pracować w dwóch trybach:

- rejestracji drzewa procesów,
- rejestracji akcji komunikacyjnych.

Użytkownik definiuje zależności czasowe poprzez zmiany macierzy T i następnie wykonuje wielokrotnie aplikacje z jednoczesnym zapisem jej zachowania do logu. Każdorazowo po wykonaniu eksperymentu następuje porównanie umożliwiające wykrycie wyścigu. Skuteczne wykrywanie anomalii czasowych zależy od odpowiedniego doboru wartości macierzy T. Problem ten wymaga jednak oddzielnego potraktowania.

Przyszłe prace będą ukierunkowane zarówno na rozszerzenie możliwości funkcjonalnych: rozproszone aplikacje obiektowe (JAVA, CORBA), jak i przenośność na inne środowiska (MPI, RPC).

LITERATURA

[1] Damodaran-Kamal S. K., Francioni J. M.: Testing Races in Parallel Programs with OtOt Strategy, Proceedings of International Symposium of Software Testing and Analysis, pp. 216-227, 1994

[2] Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V.: PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing, Oak Ridge National Laboratory, 1994

[3] Krawczyk H., Krysztop B., Proficz J.: DARE Tool for Testing Time Dependent Errors in Distributed Systems, Technical Report, Faculty of Electronics, Telecommunications and Informatics of Technical University of Gdańsk, Gdańsk, 1997

[4] Krawczyk H., Proficz J.: Pakiet wizualizacji akcji rozproszonych, Materiały Krajowego Sympozjum Telekomunikacyjnego, Bydgoszcz, str. 77-86, 1997

[5] Krawczyk H., Wiszniewski B.: Interactive Testing Tool for Parallel Programs, Software Engineering for Parallel and Distributed Systems, I. Jelly, I. Gordon and P. Croll eds., Chapman-Hall, London, pp. 98-109, 1996

[6] Lamport L.: Time, Clocks, and the Ordering of Events in a Distributed System, Commun. ACM 21, 7, pp. 558-565, 1978

Recenzent: Dr inż. Katarzyna Stąpor

Wpłynęło do Redakcji 15 grudnia 1997 r.

Abstract

The paper presents model of distributed processing (see Fig. 1) able to visualise time-dependencies occurred in a running application. It is based on the *happened before* defined by Lamport [6]. Using this model we designed and implemented a simulator in C+PVM [2]. The basic function of it is recording of tested application behaviour for different time constraints, first of all, different values of communication times. This allows to analyse distributed applications behaviour as function of this time parameters, and check, if this behaviour will be acceptable in real system described by the same parameters.

The simplified architecture of the simulator is shown in Fig. 3. It allows to store recorded application behaviours in a log and present them as bipartite graphs (see Fig. 4). The package VIsualising Communication Events (VICE) enables comparison and manipulation on the graphs which is helpful in the race detection. The most useful operator is *modulo* one which indicates the differences between different executions' of the application, beside it there are other operators (*and, or*) and some processes' filter functions.

This system can be passed to support various testing and debugging strategies of distributed software, and to evaluate some quality parameters such as test efficiency or fault coverage.