Zagadnienie równoważenia obciążenia w systemach rozproszonego przetwarzania w sieciach stacji roboczych

Andrzej Król, Andrzej Uszok
Krzysztof Zieliński

Instytut Informatyki
AGH

Luty 1993

Spis treści

1 Zagadnienie load-balancingu i jego aspekty **3**

1.1 Wstęp 3
1.2 Motywy stosowania systemów LB 4
1.3 Aspekty projektowania systemu LB 5
1.3.1 Typ algorytmu 5
1.3.2 Sposób oceny obciążenia 6
1.3.3 Stopień centralizacji algorytmu 7
1.3.4 Podstawowy elementy funkcjonalne algorytmu LB 7
1.3.5 Problemy transferu procesów 9
1.3.6 Stabilność algorytmu 9
1.4 Przykłady systemów i algorytmów LB 10
1.4.1 V-system 10
1.4.2 Sprite 10
1.4.3 Condor 11
1.4.4 Stealth 11
1.4.5 Stabilny, adaptacyjny, symetrycznie inicjowany algorytm LB 12
1.5 LB w obiektowych systemach rozproszonych 14
1.6 Podsumowanie 16

2 Koncepcja rozwiązania problemu LB dla platformy ANSA **18**

2.1 Wprowadzenie 18
2.2 Faza monitoringu obciążenia systemu 18
2.3 Faza konsolidacji modułu LB z systemem ANSA 19
2.4 Faza podejmowania decyzji o rozdzieleniu zadań na poszczególne węzły . . 20
2.5 Badania statystyczne i optymalizacja algorytmu LB 21

3 Monitor obciążenia systemu dla modułu LB systemu ANSA **22**

3.1 Założenia 22
3.2 Kapsuły systemu 22
3.2.1 MLB – Monitor Load Balancingu 22
3.2.2 Tester 24
3.2.3 PM – presentation manager 24
3.3 Opis interfejsów kapsuł 24
3.3.1 Interfejsy kapsuły *MLB* 24

3.3.2 Interfejsy kapsuły *TESTER* 26
3.4 Komunikacja miedzy kapsułami 27
3.5 Metoda uzyskiwania informacji o obciążeniu węzłów 27
3.6 Uwagi 28
3.7 Pliki systemu 28
3.8 Weryfikacja działania MLB 28
3.9 Prace obecne 29

A imakefile **31**

B mlb.dpl **33**

C tester.dpl **45**

D pm.dpl **49**

E fundata.dpl **51**

F fstat.h, fundata.h **52**

G fstat.c **53**

Rozdział 1

Zagadnienie load-balancingu i jego aspekty

1.1 Wstęp

Systemy rozproszone, które upowszechniły się bardzo w ostatnich latach dzięki dostępności tanich mikroprocesorów i ogromnemu postępowi w technologi komunikacyjnej, dokładniej w wyniku rozwoju sieci komputerowych, charakteryzują się wieloma dodatnimi cechami. Nad systemami scentralizowanymi górują przede wszystkim elastycznością dostępu do swoich zasobów. Dodanie, usunięcie, przemieszczenie zasobów komputerowych w systemie rozproszonym, przynajmniej z punktu widzenia sprzętowego, jest stosunkowo łatwym zadaniem. Daje to znaczne korzyści ekonomiczne, jak i zwiększa możliwości projektanta i użytkownika systemu. Jednakże zarządzanie takim systemem, jest o wiele bardziej skomplikowane niż pojedyńczym, nawet dużym komputerem. Celem zarządzania takim systemem jest jak najlepsze wykorzystanie wszystkich jego zasobów do rozwiązania zadań jakie stawiają mu jego użytkownicy. Pojawiają się jednak tutaj problemy, których nie było w budowanych do tej pory systemach komputerowych:

- System może składać się z dowolnego zbioru komputerów, różnych typów i wielkości, od superkomputerów, poprzez stacje robocze, aż do mikrokomputerów, zbiór ten może dynamicznie zmieniać swój stan podczas pracy.

- Zasoby systemu mimo, że potencjalnie duże są jednak rozproszone, oznacza to że nie możliwe jest przydzielenie więcej niż stosunkowo małej części tych zasobów (np jedno CPU) dla pojedynczego zadania.

- Informacja jaką mamy o stanie systemu nie jest nigdy świeża i pełna, dlatego nie da się uniknąć sytuacji, że podjęta decyzja będzie błędna.

- Użytkownik systemu nie powinien zauważać rozproszenia zasobów. Dla niego system powinien wyglądać jak konwencjonalny, scentralizowany system komputerowy, nie powinno mieć też znaczenia, którego komputera używa aby uzyskać do niego dostęp.

Brakuje na razie sprawdzonego i efektywnego oprogramowania wspomagającego zarządzanie takim rozproszonym systemem. Jedną z podstawowych części takiego oprogramowania jest moduł load-balancingu (LB), którego zadaniem jest sprawiedliwe i niewidoczne na zewnętrz systemu rozdzielenie obciążenia pomiędzy węzły, tak aby uzyskać ogólnie maksymalną wydajność działania. Mimo napotykanych trudności podjęto liczne próby napisania takiego modułu, ponieważ zyski wynikające z jego stosowania dla efektywności systemu rozproszonego są bardzo duże.

1.2 Motywy stosowania systemów LB

Zadania w systemie rozproszonym pojawiają się przypadkowo, z różnym natężeniem na dowolnym węźle systemu. Można rozróżnić cztery rodzaje zadań obliczeniowych wykonywanych przez system:

- Praca interakcyjna – edycja, rozwijanie programów, przetwarzanie tekstu,
- Przetwarzanie wsadowe – np. nieinterakcyjne uzupełnianie bazy danych, obliczenia księgowe,
- Przetwarzanie transakcji – zapytania do i uaktualnianie bazy danych,
- Obliczenia inżynierskie i naukowe – symulacje komputerowe.

Wprowadzają one bardzo różne obciążenie systemu, od stosunkowo niewielkiego dla prac interakcyjnych, po bardzo duże dla symulacji, które często wymagają całych dostępnych zasobów. Maję one również różne wymagania co do czasu swojego zakończenia. Czas ten powinien być krótki dla prac interakcyjnych i przetwarzania transakcji, a nie jest tak ważny dla przetwarzania wsadowego. Systemowi bez modułu LB bardzo trudno sprostać jest tym różnorodnym wymaganiom, mimo iż w systemie istnieją często środki zapewniające ich spełnienie. Stosowanie strategi LB, czyli przesyłania zadań z przeciążonych węzłów na inne węzły, przynosi różnorodne korzyści:

- *Polepszenie wydajności systemu*
  Równomierne użytkowanie zasobów prowadzi do większej przepustowości systemu i zmniejszenia średniego czasu odpowiedzi systemu na zadanie, ogranicza też czas w jakim procesory pozostają bezczynne.
- *Lepsze sprostanie wymaganiom sprzętowym zadania*
  Pożądanym może być aby wykonywać pewien proces na określonym komputerze, przykładowo z kooprocesorem numerycznym lub procesorem macierzowym.
- *Sprostanie wymaganiom programowym zadania*
  Pewne procesy wymagają oprogramowania dostępnego tylko na określonych węzłach (być może z przyczyn licencyjnych).
- *Unikanie ograniczeń zasobów*
  Zadanie, które nie może dalej się wykonywać z powodów ograniczeń w zasobach, może zostać przeniesione na inny komputer, dysponujący potrzebnymi środkami.

- *Zwiększenie szybkości*
  Przeniesienie dużego procesu, w sensie czasu obliczeń i zajętości pamięci, na szybszy komputer, powinno znacznie skrócić czas obliczeń.

- *Zminimalizowanie obciążeń komunikacyjnych*
  Przeniesienie procesu, do innych procesów, z którymi często się on komunikuje, lub do pliku lub danych, z których często korzysta może znacznie skrócić czas tracony na komunikację.

- *Polepszenie odporności i niezawodności*
  Rozproszenie zasobów i procesów wykonujących się na nich umożliwia kontynuację działania systemu mimo awarii pewnej liczby jego węzłów.

- *Umożliwienie budowy systemów skalowalnych*
  Wydajność takich systemów rośnie równomiernie z wzrostem ich zasobów.

Część z tych zalet ujawnia się dopiero w systemach *heterogenicznych*, niejednorodnych, czyli zbudowanych z różnych komputerów, z różnorodnym oprogramowaniem, ale także systemy *homogeniczne*, czyli jednorodne czerpią duże korzyści z stosowania LB. Jak wykazali Livny i Melman [12] nawet w takich systemach, bardzo prawdopodobne jest, że z powodu statystycznych fluktuacji w napływie nowych zadań, co najmniej jeden komputer pozostaje bezczynny podczas gdy inne są bardzo obciążone. Tak więc napewno warto jest zainwestować w rozwijanie metod i algorytmów LB. Świadczy o tym ogromna ilość prac naukowych prowadzonych w tej dziedzinie.

1.3 Aspekty projektowania systemu LB

1.3.1 Typ algorytmu

Można rozróżnić trzy typy podejścia do określania strategii podziału obciążenia:

- Statyczne – Decyzja o rozdziale zadań jest wbudowana w algorytm i nie wykorzystuje ona w żaden sposób informacji o aktualnym stanie i obciążeniu systemu, a używa jedynie wiedzy o jego budowie i działaniu. Przykład stanowi algorytm, który w sposób cykliczny przydziela zadania kolejnym węzłom lub inny trochę lepszy, przydzielający zadanie do danego węzła i z jakimś prawdopodobieństwem $p(i)$, które może zależeć od wielkości zasobów danego komputera. Algorytmy te mogą, i rzeczywiście tak się dzieje, podejmować błędne decyzje, przydzielając zadanie do przeciążonego węzła, gdy inne komputery są niewykorzystane.

- Dynamiczne – Do podejmowania decyzji o transferze zadań wykorzystują, przynajmniej w części, dane o aktualnym stanie systemu, jest to ich główna przewaga nad algorytmami statycznymi. Aby jednak wykorzystać krótkookresowe fluktuacje w rozkładzie obciążenia na poszczególnych komputerach, muszą zbierać, przechowywać i analizować informację o systemie, co oczywiście powoduje jego dodatkowe obciążenie. W dobrze zaprojektowanym dynamicznym algorytmie LB warto jest jednak ponieść tą cenę. Jest to typ algorytmu, na którym koncentruje się większość współczesnych prac badawczych.

- Adaptacyjne – Jest to specjalna klasa algorytmów dynamicznych. Ich cechą charakterystyczną jest adaptowanie swojej aktywności do zmiany stanu systemu, poprzez zmianę parametrów lub nawet strategi podejmowanych działań. Podstawowym przykładem takiego algorytmu jest algorytm, który zawiesi swoje działanie w momencie gdy system jest tak bardzo obciążony, że jakakolwiek akcja nie jest w stanie polepszyć jego wydajności, a wręcz przeciwnie pogorszyć. Algorytm nieadaptacyjny kontynuowałby pracę wprowadzając dodatkowe obciążenie i przesyłając zadania z węzła na węzeł mimo, że nie może to poprawić stanu systemu.

1.3.2 Sposób oceny obciążenia

Określenie indeksu, wskaźnika obciążenia poszczególnych komputerów systemu jest jedną z kluczowych decyzji dla poprawności i skuteczności działania algorytmu LB. Powinien on pozostawać w jak najlepszej korelacji z aktualną wydajnością systemu, z średnim czasem odpowiedzi systemu na zadania. Winien on być równy zero dla bezczynnego komputera i zwiększać się wraz z wrostem jego obciążenia. Można wyróżnić kilka zasobów, których ilość prawie zawsze jest ograniczeniem dla wykonujących się zadań, jest to: procesor, pamięć, dostęp do sieci, dysku, ilość miejsca w tablicach systemowych. Zasoby te można mierzyć między innymi poprzez:

- aktualną lub średnią, z pewnego okresu, długości kolejki procesów czekających na CPU,
- stopień wykorzystania czasu procesora,
- obciążenie sieci komputerowej,
- długość kolejki komunikatów,
- liczbę operacji dyskowych,
- wielkość dostępnej pamięci,
- ilość przełączeń kontekstu,
- ilość wykonywanych wywołań systemowych.

Można również nie obliczać jednego wskaźnika dla kolejnych węzłów, ale używać wektora współczynników obciążenia danego komputera. Należy bowiem wyróżnić dwie podstawowe kategorie mierzonych współczynników są to wielkości zależne od czasu lub od przestrzeni. Kombinacja tych danych w jeden parametr likwiduje to rozróżnienie, zadania natomiast mogą mieć różne preferencje co do żądanych zasobów. Pewne zadania mogą wymagać szybkiego zakończenia, inne dużej pamięci. Nie ma jednak jak narazie wypracowanej najlepszej metody obliczania takiego indeksu, czy zbioru indeksów. Z dotychczasowych doświadczeń wynika, że największy wpływ na wydajność systemu ma wielkość kolejki do procesora. Wydaje się również, że używanie bardzo szczegółowej informacji o systemie nie zawsze prowadzi do znaczącego wzrostu efektywności, a napewno jest o wiele bardziej kosztowna. Prosty indeks obciążenia jest w większości przypadków dostatecznie skuteczny.

1.3.3 Stopień centralizacji algorytmu

W trakcie projektowania algorytmu LB trzeba podjąć decyzje co do stopnia scentralizowania ośrodka zbierającego informacje i podejmującego decyzje o rozdziale zadań. Struktura algorytmu może być: scentralizowana, hierarchiczna, całkowicie zdecentralizowana, lub może być jakąś kombinacją tych możliwości. Im algorytm jest bardziej rozproszony tym bardziej jest odporny na różnego rodzaju awarie części systemu, dzięki rozproszeniu i redundęncji elementów modułu LB. Awarie te mogą być katastrofą dla systemu scentralizowanego, który mimo że o wiele łatwiejszy do skonstruowania, to jest jednak o wiele bardziej wrażliwy. Błąd na komputerze zarządzającym jądrem systemu, może spowodować całkowite zawieszenie pracy przez moduł LB. Drugą słabością algorytmów scentralizowanych jest ich ograniczona przepustowość, co w pewnych przypadkach, przy dużej liczbie zapytań odnośnie rozmieszczenia zadań, może ograniczać szybkość rozkładu obciążenia. Obu tych słabości pozbawiony jest algorytm rozproszony.

1.3.4 Podstawowy elementy funkcjonalne algorytmu LB

1.3.4.1 Strategia transferu procesów – *Transfer policy*

Ten element algorytmu LB decyduje o tym czy dany węzeł jest w odpowiednim stanie, aby uczestniczyć w wymianie zadań, jako dawca, lub odbiorca obciążenia. Dwa najpopularniejsze rozwiązania to:

- Algorytm z progiem (*threshold*) [12] –
  Przyjmujemy dwa progi obciążenia, wyrażone w jakiś przyjętych jednostkach: dolny *T1* i górny *T2*. Jeśli obciążenia danego węzła spodnie poniżej *T1* to staje się on odbiorcą zadań, odbiorca (*receiver*), a jeśli wzrośnie powyżej *T2* to staje się on dostarczycielem zadań, dawca (*sender*). Progi te mogą mieć tą samą wartość, lub zależnie od algorytmu, różną.

- Algorytm porównawczy [12] –
  Rozważamy obciążenie komputera porównując je z obciążeniem innych węzłów i na tej podstawie decydujemy czy jest on odbiorcą, czy dawcą zadań. Można na przykład przyjąć, że jest on odbiorcą pracy od jakiegoś innego komputera jeśli jego obciążenie jest mniejsze od tego drugiego o jakąś określoną wartość *delta*, lub jest jednym z najmniejszych w systemie.

1.3.4.2 Strategia wyboru zadań – *Selection policy*

Zadaniem tej części algorytmu jest wybór odpowiedniego zadania do transferu. Aby podjąć tą decyzję rozważa on kilka czynników:

1. Koszty transmisji danego zadania na inny węzeł – powinnien być on minimalny. Im mniejsze rozmiary zadania tym oczywiście jest on mniejszy.

2. Czas obliczeń, który pozostał zadaniu do wykonania – powinnien być on dostatecznie długi, aby miało sens ponoszenie kosztów transferu.

3. Ilość zależności od lokalnych zadania – powinno być ich jak najmniej ponieważ po przeniesieniu rozważanego zadania zależności staną się zdalnymi, obciążając sieć i spowalniając obliczenia.

Podjęcie właściwej decyzji nie jest sprawą łatwą, ponieważ zarówno określenie przewidywanego czasu zakończenia zadania jak i ilości różnorodnych powiązań lokalnych opierać się może jedynie na bardzo niepewnych i niepełnych założeniach i danych. W tym celu próbuje się stosować różnorodne techniki probabilistyczne i heurystyczne żadna z nich nie jest jednak jak na razie doskonała. Najprostszym rozwiązaniem jest wybór ostatnio uruchomionego zadania na danym komputerze. Powinno ono posiadać jeszcze dość długi czas obliczeń, oraz ilość utworzonych powiązań lokalnych nie powinna być jeszcze tak duża.

1.3.4.3 Strategia lokalizacji dawców i biorców – *Location policy*

Celem tej części algorytmu jest znalezienie, dla danego węzła, odpowiedniego partnera dla transmisji zadania,to jest dawcę lub odbiorcę obciążenia w zależności od potrzeby. W przypadku algorytmu rozproszonego stosuje się metodę zapytań (*polling*), polegającą na tym, że węzeł zapytuje inne węzły o to czy są one odpowiednie dla niego. Zapytania do węzłów mogą następować kolejno lub równolegle poprzez multicast. Węzły mogłą być wybierane do zapytania przypadkowo lub na podstawie poprzednio zgromadzonych informacji. Alternatywą jest zapytanie wszystkich węzłów poprzez broadcast. W przypadku strategii scentralizowanej węzeł kontaktuje się z wyróżnionym komputerem, koordynatorem, który zbiera dane o systemie i na tej podstawie wybiera partnera dla niego.

1.3.4.4 Strategia zbierania informacji – *Information policy*

Ten element algorytmu decyduje o tym kiedy, skąd i jaka informacja będzie zbierana o stanie systemu. Można wyróżnić trzy rodzaje podejścia do tego problemu:

1. Na żądanie – *Demand-driven*
   To podejście jest stosowane w algorytmach rozproszonych, charakteryzuje się dynamicznym podejmowaniem działann zależnych od stanu systemu. Informacja jest zbierana dopiero wtedy gdy występuje taka potrzeba. W algorytmach, w których inicjatywa należy do dawcy (*sender-initiated*), dawca (*sender*) szuka danych o odbiorcy swoich nadmiarowych zadań. W algorytmach, w których inicjatywa należy do odbiorcy (*receiver-initiated*), to odbiorca (*receiver*) inicjuje działania. Można również stosować oba te podejścia jednocześnie (*symmetrically-initiated*).

2. Okresowo – *Periodic*
   Można stosować to podejście zarówno w strategiach scentralizowanych, jak i rozproszonych. W tym podejściu dane na temat stanu systemu zbierane są okresowo, co

pewien określony kwant czasu. Strategia ta jest statyczna, nie adaptuje się ona do potrzeb systemu. W przypadku gdy system jest bardzo obciążony, tylko dociąża system.

3. W momencie zmiany stanu – *State-change-driven*
W tym podejściu węzeł rozpowszechnia dane o swoim stanie, jeśli tylko zmienił się on w określonym stopniu. W algorytmie scentralizowanym dane przesyłane są do koordynatora, a w algorytmie rozproszonym do partnerów.

1.3.5 Problemy transferu procesów

Jeśli wiadomo już skąd, dokąd i jakie zadanie przesłać, to pozostaje jeszcze problem, który sprawia dużo trudności – jak to zrobić. Można wyróżnić dwa rodzaje przesyłań:

- Przesyłanie z wywłaszczaniem – *Preemptive task transfer*
Jest to przesyłanie zadania w środku jego wykonywania, wiąże się to z zebraniem całego stanu procesu: obrazu pamięci wirtualnej, bloku kontrolnego procesu, nieprzeczytanych buforów, komunikatów, deskryptorów plików, liczników. Jest to niekiedy bardzo duża ilość danych, które muszą być przesłane poprzez sieć. Co więcej muszą być one kompletne aby zapewnić możliwość odtworzenia stanu procesu na innym węźle. Jest to generalnie bardzo kosztowna operacja, zabierająca dużo czasu

- Przesyłanie bez wywłaszczania – *Nonpreemptive task transfer*
Jest to przesłanie nowego zadania, którego nie rozpoczęto jeszcze wykonywać, dlatego nie trzeba przesyłać jego stanu. Jest to na pewno mniej kosztowne niż przesyłanie z wywłaszczaniem.

W obu przypadkach należy przesłać informacje o środowisku, w którym działa proces, jak np: aktualny katalog roboczy lub przywileje i prawa dostępu. W systemach heterogenicznych problemem jest również to, że węzły różnią się konstrukcją (procesorem, pamięcią, systemem operacyjnym), dlatego nie zawsze możliwy jest bezpośredni transfer obrazu zadania z jednego komputera na inny. Stwarza to potrzebę przekształcenia zadania lub nawet jego rekompilację. Mechanizm migracji powinien być niezawodny (*reliable*) i przezroczysty (*transparent*), to znaczy użytkownik nie powinien zauważać żadnych zmian w działaniu systemu.

1.3.6 Stabilność algorytmu

Jest to bardzo ważny aspekt projektowania algorytmu LB. System bez modułu LB może pracować źle, często mogą pojawiać się duże opóźnienia w jego odpowiedzi, a zasoby systemu mogą być nie wpełni wykorzystane – system jednak pracuje. Po zastosowaniu błędnie skonstruowanej strategi LB może okazać się, że system w ogóle nie pracuje, lub jest niestabilny to znaczy pewne rozkłady obciążenia powodują dziwne zachowanie systemu. Może wystąpić tzw *migotanie* zadań lub pojawianie się dodatkowego, bardzo dużego obciążenia spowodowanego działaniem modułu LB. Może okazać się również, że algorytm jest stabilny ale nieefektywny – system pracował lepiej bez niego. Wszystkie te aspekty należy wziąć pod uwagę już na etapie projektowym, a powstały moduł dokładnie przetestować pod względem stabilności i efektywności.

1.4 Przykłady systemów i algorytmów LB

1.4.1 V-system

Artykuł: [12]
System ten wykorzystuje algorytm rozproszony, inicjowany przez zmianę stanu systemu. Każdy węzeł rozgłasza swój stan jeśli tylko ulegnie on znaczącym zmianom. Informacja ta zawiera stopień wykorzystania procesora i pamięci, oraz szczegóły o konstrukcji komputera – rodzaj jego procesora, i to czy posiada kooprocesor numeryczny. Informacja ta jest przechowywana przez każdy węzeł. Jeśli jednak system jest zbyt duży to nie przechowuje się całej informacji, ale jedynie dane o N najlepszych komputerach. Do migracji wyznaczane są jedynie nowo powstałe zadania. Węzeł jest określony jako odbiorca zadań jeśli jest jednym z M najmniej obciążonych węzłów, a w przeciwnym wypadku jest dawcą zadań. Jeśli nowe zadanie pojawia się na jakimś węźle to na podstawie zebranych informacji konstruowany jest zbiór M najgorzej wykorzystanych komputerów. Jeśli rozważany węzeł należy do tego zbioru to zadanie nie jest nigdzie transferowane. W przeciwnym razie wybierany jest przypadkowo jeden komputer z utworzonego zbioru i zapytywany jest o aktualny stan. Taki przypadkowy wybór redukuje szansę, że wiele różnych węzłów wybierze ten sam komputer jako swojego partnera. Jeśli wybrany *host* potwierdzi swój stan, to zadanie migruje do niego, jeśli stan *hosta* zmienił się i nie jest już on odpowiedni do transferu to próbuje się innego komputera z utworzonego zbioru. Praktyka wykazała, że bardzo rzadko potrzeba więcej niż trzech prób.

1.4.2 Sprite

Artykuł: [12]
System ten został stworzony dla środowiska stacji roboczych. Jest to system scentralizowany, inicjowany przez zmianę stanu *hostów*. Każda stacja robocza informuje o zmianie swojego stanu, wyróżniony proces koordynatora. Aby znaleźć dla siebie partnera, który odbierze od niej część zadań, musi ona również skontaktować się z koordynatorem. Stacja staje się odbiorcą zadań jeśli przez 30 sekund nie pojawia się na niej zdarzenie pochodzące z klawiatury czy myszy, a liczba procesów wykonujących się na niej jest mniejsza niż liczba jej procesorów. Jednakże wybór zadań, które powinny zostać przeniesione na inne maszyny, dokonuje się ręcznie poprzez decyzję użytkownika. Ponieważ każda stacja robocza jest przypisana do tego użytkownika, który się na niej zalogował, więc system musi zagwarantować jemu przede wszystkim zasoby stacji. Obce zadanie usuwane jest z niej, jeśli tylko użytkownik chce jej używać. Usuwane zadania wybierane są automatycznie i wracają do stacji, na której się po raz pierwszy pojawiły. System *Sprite* stara się w miarę sprawiedliwie przydzielać zasoby zadaniom, dlatego może zdarzyć się, że proces zostanie usunięty ze stacji, a na jego miejsce przetransferowany zostaje inne

zadanie. Stanie się tak jeśli koordynator nie potrafi znaleść wolnego węzła, a stwierdzi, że zadania pewnego użytkownika zajmują znacznie więcej zasobów niż innych. Projektanci tego systemu użyli połautomatycznej strategi wyboru zadań do migracji ponieważ uważali, że skonstruowanie w pełni automatycznego algorytmu nie warte byłoby kosztów jego implementacji.

1.4.3 Condor

Artykuł: [12]
To także system, którego przeznaczeniem jest sieć stacji roboczych, a koncepcja jego budowy zapewnia pełnię możliwości aktualnie pracującemu użytkownikowi . Jego konstrukcja jest bardzo podobna do budowy systemu *Sprite*. Jest to też system scentralizowany, z wyborem zadań do transferu przez użytkownika, który musi zkonsolidować je z specjalną biblioteką funkcji i umieścić w lokalnej kolejce zadań wykonywanych w tle. Zadaniem koordynatora jest znalezienie węzła , na który można przesłać te zadania. Aby temu sprostać pyta on okresowo, co dwie minuty, każdą stację czy jest wolna lub czy ma może zadania do transferu. Zakłada się, że stacja jest wolna jeśli użytkownik nie używał jej przez 12.5 minuty. Natomiast dane o czekających zadaniach są kolejkowane przez koordynatora. Jeśli tylko znajdzie on wolne komputery kolejne zadania migrują na nie. Stacja, która obsługuje takie przetransferowane zadanie jest kontrolowana co 30 sekund, przez lokalny kontroler, czy jej właściciel nie wykazał jakiejś aktywności, jeśli tak to zadanie jest wywłaszczane z zasobów, a jego stan zapamiętywany. Zadanie może oczekiwać w takim stanie na zaprzestanie pracy przez użytkownika przez 5 minut, po tym czasie jest przemieszczane wraz ze swoim zapamiętanym stanem do wyjściowej stacji roboczej. Ten system, podobnie jak *Sprite*, stara się wprowadzić mechanizm sprawiedliwego dostępu do zasobów, dla wszystkich użytkowników. Każdej stacji przypisany jest licznik, na początku ustawiany na zero, a następnie uaktualniany. Jego wartość wzrasta o jeden w momencie gdy zadanie z danego węzła zostanie przetransferowane na inny, a zmniejszany o jeden dla każdego zadania oczekującego na przesłanie dłużej niż określony czas. Koordynator okresowo sprawdza czy jakieś zadanie nie czeka na wolny komputer i jeśli tak jest, a licznik lokalnego komputera jest mniejszy niż liczniki komputerów, których zadania wykonują się aktualnie, to jedno z nich zostaje wywłaszczone a na jego miejsce migruje oczekujące zadanie.

1.4.4 Stealth

Artykuł: [12]
Jak poprzednio zaprezentowane systemy, również i *Stealth* przeznaczony jest dla środowiska stacji roboczych, w którym dostępność zasobów stacji musi być zagwarantowana dla jej aktualnego właściciela. Jego konstruktorzy zauważyli jednak, że nawet gdy stacja jest wykorzystywana przez użytkownika, to i tak często duża część jej możliwości jest nie wpełni spożytkowana. W całym systemie sięga to nawet 50 procent jego zasobów. Dlatego pozwolili oni wykonywać się eksportowanemu zadaniu na stacji, której właściciel jest aktywny. Właściciel stacji zachowuje jednak uprzywilejowany dostęp do zasobów

poprzez specjalny mechanizm ich alokacji, przyznający mu największy priorytet. Zadania właściciela otrzymują wszystkie potrzebne im zasoby, a obce zadanie musi zadowolić się tym co pozostanie. Wybór zadań do transferu jest tu wpełni automatyczny, w odróżnieniu od poprzednio opisanych systemów. Główną jego przewagą jest to, że zastąpił kosztowne globalne operacje przydziału zadań do węzłów, tanimi operacjami lokalnego, priorytowego, przydziału zasobów. Pozwala wykorzystać nie tylko wolne stacje, ale również nie wpełni obciążone.

1.4.5 Stabilny, adaptacyjny, symetrycznie inicjowany algorytm LB

Artykuł: [12]
Ponieważ jest to algorytm symetrycznie inicjowany, więc zarówno dawca (*sender*), jak i odbiorca (*receiver*) inicjują aktywność LB. Opiszemy najpierw, osobno te dwa algorytmy, a następnie właściwy algorytm, który jest ich kombinacją. Są to oczywiście algorytmy rozproszone.

1.4.5.1 Load-balancing inicjowany przez dawcę (*sender*)

Inicjatywa w tej strategi należy do przeciążonego węzła, dawcy *sender*, który próbuje pozbyć się nadmiarowych zadań, poprzez znalezienie w systemie odpowiedniego ich odbiorcy i wysłanie tych zadań do niego. Algorytm rozpoczyna pracę, gdy pojawia się nowy proces, dla którego zabrakło zasobów komputera. Komputer wysyła wtedy zapytania do innych komputerów i na podstawie otrzymanych odpowiedzi, podejmuje decyzję czy i gdzie wysłać ten proces. Strategi ta wykazuje niestety niestabilność przy dużym obciążeniu systemu. Wtedy znalezienie nie przeciążonego węzła jest mało prawdopodobne, więc i dawca (*sender*) najczęściej poniesie porażkę w swych poszukiwaniach. Jednakże intensywność zapytań o wolne węzły rośnie wraz z wzrostem szybkości przybywania nowych zadań, osiągając w końcu punkt gdy koszty tego są większe niż uzyskane zyski w przyśpieszenu pracy systemu. Zaletą tego algorytmu jest to, że używa on do migracji tylko nowo powstałych zadań (przesyłanie bez wywłaszczania), co jest o wiele mniej kosztowne.

1.4.5.2 LB inicjowany przez odbiorcę (*receiver*)

Aktywność LB wyzwalana jest kiedy kończy się wykonywać jakieś zadanie, na którymś z węzłów systemu, a długość kolejki procesów na tym komputerze spadnie poniżej progu T. Rozpoczyna on wtedy proces poszukiwania przeciążonych węzłów, od których nadmiarowe zadania mogłyby zostać przetransferowane do niego. Problem tej strategi jest to, że jeśli ten proces się nie powiedzie i nie zostanie znalezione żadne zadanie, to moc tego komputera jest tracona do chwili gdy jakieś nowe zadanie zakończy się na nim i opisany proces rozpocznie się od nowa. Należy więc wtedy wprowadzić okresowe poszukiwanie systemu przez odbiorcę (*receiver*). Stabilność tego algorytmu jest znacznie lepsza niż poprzedniego. W przypadku dużego obciążenia systemu odbiorca (*receiver*) szybko znajdzie dawcę (*sender*) i jego efektywność wzrasta wraz z wzrostem obciążenia. Wadą tego algorytmu jest to, że wybrane do migracji zadania najczęściej były już wykonywane i trzeba stosować przesyłanie z wywłaszczaniem, co jest kosztowne.

1.4.5.3 Właściwy algorytm

Algorytm symetryczny łączy zalety obydwu opisanych strategii, przy małym obciążeniu inicjowane przez dawców (*sender*) akcje łatwo odnoszą sukces, przy dużym obciążenu to inicjowane przez odbiorców (*receiver*) działania są bardziej skuteczne. Ale stworzony algorytm odziedziczył też wady obu swych poprzedników, aby je wyeliminować zmodyfikujmy nieco powstałą strategię. Głównym powodem niestabilności, są niepotrzebne zapytania wysyłane przez dawcę *sender*. Aby tego uniknąć nowy algorytm używa informacji zebranych w procesie przepytywania do poklasyfikowania węzłów na trzy kategorie: nie wpełni wykorzystane, przeciążone i OK (z odpowiednim obciążeniem). Te dane przechowywane są w trzech listach. Na samym początku każdy komputer zakłada, że wszystkie węzły systemu, poza nim są odbiorcami zadań, jego lista odbiorców zawiera więc wszystkie komputery, pozostałe listy są puste. Uaktualniane tych list powoduje tylko małe i stałe dodatkowe obciążenie systemu, niezależne od jego rozmiarów. W zależności od sytuacji na danym węźle uaktywnia się strategia dawcy (*sender*) lub odbiorcy (*reciver*). Opiszmy je kolejno:

- Strategia dawcy (*sender*)
  Zapytuje on kolejno węzły z listy odbiorców, czy dalej nimi są. Pytany komputer najpierw usuwa identyfikator pytającego komputera z listy, na której się on aktualnie znajduje, a umieszcza go za to na samym początku listy odbiorców. A następnie informuje go o swoim aktualnym stanie. Na podstawie tej informacji uaktualniane są listy na węźle dawcy (*sender*) i jeśli węzeł potwierdził, że jest odbiorcą (*reciver*) to następuje transfer zadania. Proces pytania kończy się jeśli:

  - znaleziony zostanie odbiorca (*receiver*),
  - liczba zapytań przekroczy maksymalną ich liczbę,
  - wszystkie *hosty* z listy odbiorców zostaną przypytane.

  Jeśli nie zostanie znaleziony odpowiedni komputer dla zadania to jest ono przetwarzane lokalnie.

- Strategia odbiorcy (*receiver*)
  Komputery, które będą wykorzystywane w procesie przepytywania są wybierane w następującej kolejności:

  1. Węzły z głowy listy dawców aż do jej ogona. Najświerzsze informacje będą wykorzystane najpierw.
  2. Komputery z ogona listy *OK* aż do jej glowy. W nadzieji, że najstarsze informacje są już nieaktualne i komputer stał się dawcą (*sender*).
  3. Komputery z ogona listy odbiorców (*receiver*) aż do jej glowy. Z tych samych powodów, jak w punkcie 2.

W czasie zapytań w podobny sposób uaktualniane są listy zarówno komputera zapytywanego jak i pytającego. Jeśli znaleziony zostanie dawca (*sender*) to zadanie z niego migruje do odbiorcy (*receiver*). Warunki zakończenia tego procesu są takie same jak poprzednio.

Wprowadzenie tych ulepszeń wyeliminowało wady poprzednich algorytmów. Przy bardzo obciążonym systemie zapytania dawców (*senders*) nie odnoszą sukcesu, ale dzięki nim lista potęcjalnych odbiorców staje się pusta, tak więc w przyszłości inicjowane strategie dawców (*senders*) nie będą zalewały systemu niepotrzebnymi zapytaniami. Deaktywuje się więc w tej sytuacji strategię dawcy (*sender*), natomiast strategia odbiorcy (*receiver*) łatwo odnosi sukces. Przy słabo obciążonym systemie to właśnie ta strategia ponosi porażkę, ale dodatkowe obciążenie wprowadzone przez nią można wtedy zaakceptować, ponieważ system ma nadwyżkę mocy. Dodatkową zaletą jest to że jeśli lista odbiorców jest aktualna to dawca szybko znajdzie dla siebie partnera. Zaprezentowany algorytm został zaimplementowany przez jego autorów [12], przetestowany i porównany z innymi algorytmami. Okazał się on spośród nich najlepszy, bardzo stabilny w całym zakresie obciążenia, nawet przy 90% obciążeniu średni czas odpowiedzi na zadanie nie różnił się od tego samego czasu przy 50% obciążeniu o więcej niż 10%.

1.5 LB w obiektowych systemach rozproszonych

Większość takich systemów można rozważać jako pojedyńczą, rozproszoną pulę obiektów, powiązanych ze sobą różnorodnymi zależnościami i kontaktującymi się ze sobą w dość intensywny sposób. Pojedyńczy program składa się z wielu takich obiektów, niektóre z nich mogą być dzielone z innymi programami, inne są jego prywatną własnością. Dokładne położenie obiektu w systemie nie jest zazwyczaj znane programowi, ważne jest jedynie aby miał on do niego dostęp kiedy tylko będzie tego potrzebował. Wydaje się, że jest to środowisko wręcz wymarzone dla zastosowania LB. Aplikacja jest już podzielona na cześci, obiekty są zamkniętymi całościami, nie ważne jest zazwyczaj ich aktualne położenie. W zależności od potrzeb mogą one migrować z jednego węzła systemu na drugi. Większość opisanych powyżej strategi LB można stosować w systemach obiektowych. Istnieją jednak dwa aspekty budowy tych algorytmów, które są charakterystyczne dla sytemów rozproszonych:

- Problem wykrywania zespołów obiektów (*clumps*)
  Są to obiekty, które współpracują ze sobą bardzo intensywnie, bardzo często się komunikują, a ich rozproszenie pomiędzy wieloma węzłami systemu zwiększyłoby intensywność obciążenia sieci komputerowej łączącej system, oraz oczywiście wprowadziłoby znaczne opóźnienia. Aby temu zapobiec należy wykrywać takie zespoły obiektów i albo pozostawiać je na danym komputerze, albo przesyłać na nowy węzeł wszystkie obiekty z zespołu. Określenie członków takiej grupy obiektów nie jest sprawą łatwą, ponieważ obiekty z tego grona nie porozumiewają się tylko między sobą, ale także z innymi obiektami w systemie. O zakwalifikowaniu danego obiektu do danej grupy może decydować intensywność jego komunikacji z innymi członkami tej społeczności. Do formowania takich zespołów mogą służyć informacje o:

  - lokalnych cyklach, zebrane podczas procesu *garbage collection*,
  - spoistości zespołu, dostarczane przez specjalne liczniki aktywności komunikacyjnej, wprowadzone do obiektów,
  - preferencjach obiektów, dostarczanych przez specjalne referencje posiadane przez obiekty, są one dwóch typów:
    * *colocator* – obiekty posiadające tę referencję muszą znajdować się na tym samym węźle,
    * *contralocator* – obiekty posiadające tę referencję nie mogą znaleźć się na tym samym komputerze.

  Na podstawie tych danych i jego wielkości podejmuje się decyzję o transferze zespołu. System Bellerophon zaprezentowany w artykułach: [3] i [4] jest przykładem systemu, który implementuje taki właśnie sposób wykrywania klastrów.

- Budowa mechanizmów migracji obiektów
  Mechanizm taki powinien być dostarczany przez obiektowy system operacyjny. Moduł LB powinnien korzystać z udostępnianych mu przez ten system możliwości przesyłania i migracji obiektów. Mechanizmy takie powinny być:

  - przezroczyste (*transparent*) dla innych elementów systemu,
    to znaczy nie powinny w żaden sposób zakłócać jego pracy, ani podczas migracji obiektu, ani po zainstalowaniu go na nowym miejscu,
  - pewne (*reliable*)
    obiekt nie może zaginąć podczas przesyłania, ani nie może zostać naruszona jego wewnętrzna struktura,
  - szybkie
    transport obiektu musi przebiegać najszybciej jak to jest tylko możliwe, aby nie spowalniać pracy sytemu.

  Sprostanie tym wymaganią nie jest łatwe, tylko bardzo nieliczne systemy obiektowe oferują nieograniczone niczym mechanizmy transferu obiektów. Za przykład takiego systemu może służyć *Emerald* [6]. Ale już *ANSA* nie dostarcza takiego mechanizmu, stąd moduł LB w tym systemie musi tworzyć kopię przenoszonego obiektu na nowym miejscu, a następnie kasować stary obiekt. Należy także odpowiednio zmienić zarejestrowanie obiektu w sytemie. Cała ta procedura jest skomplikowana i kosztowna. Dwa z wielu możliwych rozwiązań tych problemów proponują systemy:

  - COOL-2 [2]
    Jest to system, który rozszerza pojęcie tradycyjnej, pojedyńczej przestrzeni adresowej na środowisko rozproszone używając techniki rozproszonej pamięci wirtualnej i tylko jednego poziomu identyfiaktorów obiektów – takie same identyfikatory lokalne i globalne. Całość koncepcji opiera się na dzielonej pamięci (*shared memery*) pomiędzy różnymi kontekstami na różnych komputera. Taka dzielona pamięć dostarcza pojedyńczej przestrzeni adresowej

dla rozproszonej aplikacji. Aby to osiągnąć musi istnieć spójny mechanizm przełączania kontekstów na wszystkich węzłach. Obiekty należące do tej samej aplikacji umieszczane są w tym samym kontekście. Może się jednak zdarzyć, że powoduje to nakładanie się na siebie dwóch różnych obiektów. Wtedy jeden z nich reprezentowany jest w tym kontekście przez zastępcę (*proxy*). Migracja obiektu wewnątrz tego samego kontekstu nie sprawia problemów, ponieważ przenosząc obiekt na nowy komputer, jesteśmy pewni, że jest dla niego miejsce w tej samym miejscu przestrzeni adresowej co na orginalnym komputerze. Dlatego nie trzeba przeadresowywać niczego wewnątrz obiektu, a dzięki globalnym identyfikatorom obiekt jest tak samo jak poprzednio dostępny w systemie. Migracja pomiędzy kontekstami polega na umieszczeniu obiektu w danym kontekscie, poprzez translację adresów (*swizzeling*), a następnie tak jak poprzednio przeniesienu go na żądany komputer.

– Distributed Concurrent Object Oriented C [13]
  Wykorzystano tutaj technikę wielopoziomowego adresowania i propagacji adresów. System jest oparty na modelu trzypoziomowym:

  1. Obiekty
     kolekcja obiektów wykonujących się współbieżnie zgodnie z regułami określonymi przez warstwę systemu operacyjnego,
  2. System operacyjny
     system zapewniający tworzenie obiektów, ich komunikację, przydział pamięci, każda aplikacja ma swój własny system operacyjny,
  3. Jądro
     zapewnia dostęp do zasobów systemowych, komunikację pomiędzy warstwami systemów operacyjnych i pomiędzy obiektami działającymi w różnych systemach operacyjnych, na każdym komputerze działa jedno jądro.

  Migracja może mieć miejsce na wszystkich tych trzech poziomach. Adresy obiektów, systemów operacyjnych i jąder określane są względem niższych warstw. Daje to w rezultacie globalne adresy obiektów i pogrupowanie adresów ze względu na strukturę systemu. Przemieszczanie obiektów polega na zmianie przez niego sytemu operacyjnego, i to jego zadaniem jest zainstalowanie obiektu na nowym komputerze.

1.6 Podsumowanie

Wszystkie przedstawione tu rozwiązania problemu LB są strategiami heurestycznymi, takie podejście jest wybierane dlatego, że algorytm optymalny jest zbyt złożony do zaprojektowania i zaimplementowania w obecnych systemach. Chociaż zadaniem LB jest zwiększania wydajności systemu to jednak niektóre jego aspekty mogą ją pogorszyć. Jednakże jeśli tylko jest on dobrze zaprojektowany to nawet prosty algorytm wprowadza bardzo duże polepszenie wydajności. Okazuje się, że nawet wysyłanie zadań, które przybyły do zajętego komputera, przypadkowo do innych komputerów znacznie zwiększa wykorzystanie systemu. Dlatego napewno stosowanie algorytmów LB stwarza istotne możliwości poprawy efektywności pracy rozproszonych systemów komputerowych.

Rozdział 2

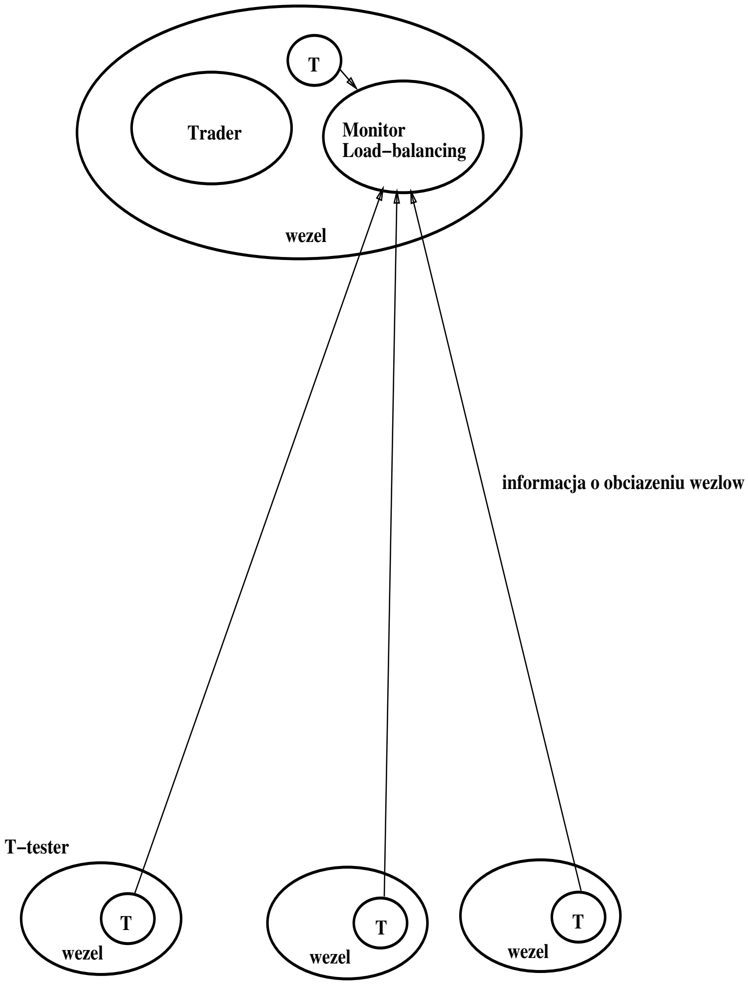
Koncepcja rozwiązania problemu LB dla platformy ANSA

2.1 Wprowadzenie

System ANSA jest platformą, dla obiektowo zorientowanego programowania, rozproszonego, posiadającą wiele mechanizmów ułatwiających programowanie rozproszone i ukrywających przed użytkownikiem jego złożoność. Celem projektu jest wzbogacenie tej platformy o moduł LB, którego nie posiada. Zadaniem tego modułu jest zwiększenie efektywności przetwarzania w systemie. Wpływając na alokację nowych obiektów programu i odpowiednio nimi zarządzając moduł ten dąży do jak najbardziej równomiernego obciążenie wszystkich węzłów sieci, na której zainstalowana jest ANSA. Praca nad rozwiązaniem tego problemu rozbija się w logiczny sposób na kilka faz, których propozycję rozwiązania przedstawiono poniżej. W wyróżnieniu tych faz oraz w koncepcji ich realizacji wykorzystano propozycje rozwiązań omówionych w rozdziale 1.

2.2 Faza monitoringu obciążenia systemu

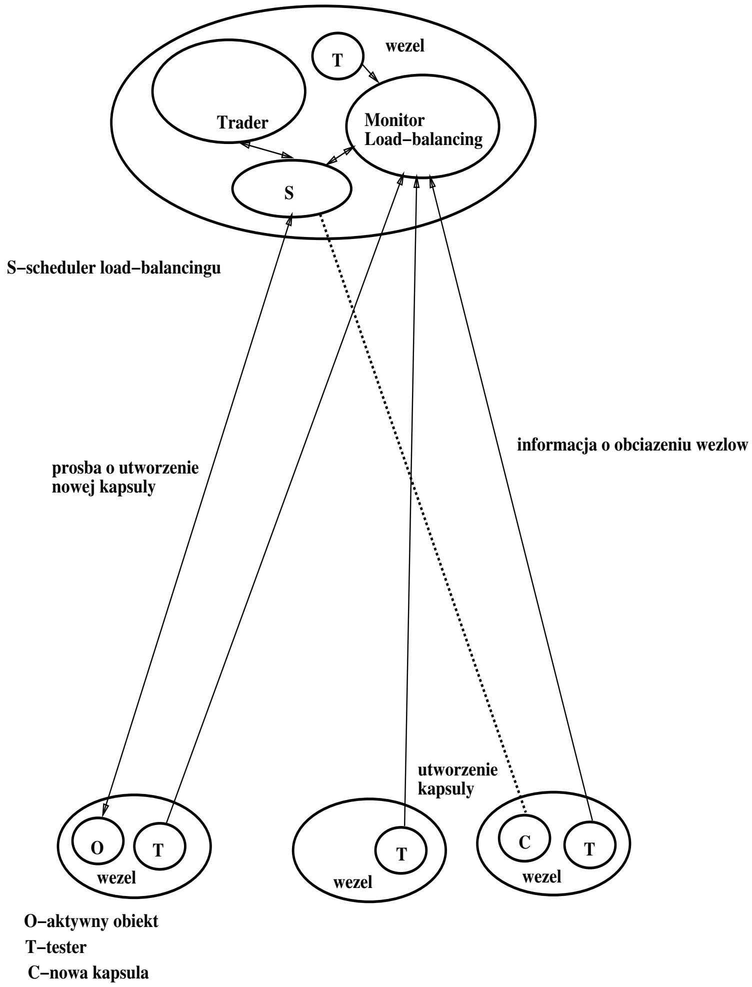
Aby móc podjąć decyzję o tym jak zarządzać systemem trzeba mieć najpierw informację o jego aktualnym stanie. Należy więc rozwiązać zadanie monitorowania systemu. Wszystkie elementy monitora systemu mogą powstać na bazie platformy ANSA. Proponuje się przyjąć jego scentralizowany model (w podobny sposób rozwiązana jest struktura platformy ANSA – jeden *Trader* zarządza obiektami na wielu węzłach). W praktyce oznacza to, że należy napisać kapsułę Monitor – Load – Balancing'u (MLB) uruchamianą na węźle, na którym znajduje się *Trader*, oraz prototyp kapsuł Testerów uruchamianych na wszystkich węzłach. Kapsuły te powinny być tworzone w momencie ładowania systemu, a ich inicjacja wprowadzona do odpowiedniego skryptu urchamiającego platformę ANSA. Zadaniem MLB jest zbieranie informacji o obciążeniu poszczególnych komputerów systemu w postaci różnych parametrów go charakteryzujących (np. procent zajętości czasu procesora, pamięci, czasie dostępu do danego węzła itp.) od Testerów działających na poszczególnych komputerach sieci. Moduł MLB przechowywuje również inne informacje, o stanie systemu, potrzebne dla algorytmów LB jak np. dane charakteryzujące możliwości węzłów (jakość procesora, ilość pamięci, czas dostępu do niego itp.).



Rysunek 2.1: Elementy systemu monitoringu

2.3 Faza konsolidacji modułu LB z systemem ANSA

Aby stworzyć możliwość korzystania z modułu LB w sposób transparentny przez dowolną aplikacje napisaną dla platformy ANSA, proponuje się przechwytywanie jej odwołania do systemu ANSA. Dotyczy to w szczególności odwołań do *Trader* i *Factory*, a być może również do innych elementów ANSA. Odwołania te kierowane są do nowej kapsuły Scheduler — Load — Balancing'u (SLB), pełniącej rolę filrta pomiędzy aplikacją a systemem ANSA i zapewniającej możliwość ich ewentualnej modyfikacji na podstawie informacji z MLB i wybranej strategi LB. W ten sposób zostaje wymuszona konsultacja z SLB podczas podejmowania działań mających istotny wpływ na rozłożenie obciążenia w systemie, a więc podczas alokacji nowych obiektów i zadań. Należy robić to jednak tak, by nie wpływało to na działanie programu, czyli było całkowicie transparentne dla programisty, jak również użytkownika (oprócz przyspieszenia działania aplikacji). Można to zrobić poprzez zamianę wywołań języka PREPC na odpowiednie wywołania procedur z SLB. W tym celu proponuje się implementację odpowiedniego preprocesora dla plików z rozszerzeniem .dpl.



Rysunek 2.2: Scheduler — Load — Balancingu w systemie ANSA

2.4 Faza podejmowania decyzji o rozdzieleniu zadań na poszczególne węzły

Pisząc SLB należy podjąć decyzję jakie dokładnie odwołania przechwytywać i w jaki sposób je przekształcać. Istnieje tutaj możliwość wyboru odpowiedniej opcji podczas uruchamiania SLB. Pojawia się tutaj kilka możliwości wynikających z tego jak głęboko wchodzi się w strukturę systemu ANSA. Rozważane poziomy ziarnistości i rodzaje podejmowanych decyzji są następujące:

- *Kapsuła* — gdzie uruchomić nową kapsułę.
- *Obiekt* — w której z istniejących kapsuł danego typu utworzyć żądany obiekt. Jeśli wszystkie kapsuły znajdują się na bardzo obciążonych węzłach to należy rozważyć czy nie utworzyć na jakimś węźle mniej obciążonym nowej kapsuły i dopiero w niej zainicjować obiekt danego typu.
- *Interfejs* — z którego obiektu należy wyeksportować dany interfejs, postępujemy tu podobnie jak dla Obiektu tylko należy rozważyć o jeden więcej poziom zagłębienia, to jest poziom obiektów a nie kapsuł.
- *Usługa* — gdzie należy uruchomić daną usługę. Jeżeli posiadany interfejs wskazuje na usługę uruchomioną na obciężonym komputerze i należałoby poszukać alternatywnego węzła, na którym znajduje się obiekt, z którego można wyeksportować nowy interfejs dla danej usług. Jeżeli nie ma takiego obiektu, albo wszystkie znajdują się na obciążonych węzłach i należy poszukać odpowiedniej kapsuły, w której można utworzyć obiekt. Może okazać się w końcu że całą strukturę: kapsuła, obiekt, interfejs trzeba będzie stworzyć od początku.

Roztrzygając problem wyboru miejsca alokacji nowego obiektu SLB wykorzystuje dane zbierane i przechowywane przez MLB. Realizowana przez SLB strategia może kłaść nacisk na różnorodne parametry obciążenia systemu. Wybór odpowiedniej strategii stanowi odrębne zagadnienie i jest przedmiotę dalszych rozważań.

2.5 Badania statystyczne i optymalizacja algorytmu LB

Po napisaniu systemu LB kolejnym etapem jest przetestowanie jego skuteczność dla różnych strategii działania SLB. Ważne jest również określenie czasu jaki zabiera systemowi utworzenie nowej kapsuły, obiektu, wyeksportowanie nowego interfejsu i na tej podstawie zdecydować się na odpowiedni poziom zagłębienia się w strukturę systemu ANSA podczas LB. Istotną rzeczą jest zbadanie stabilność napisanego modułu przy różnym obciążeniu systemu, czy ewentualnie powstające nowe kapsuły i obiekty nie blokują pracy systemu, powodując efekt odwrotny do oczekiwanego.

Rozdział 3

Monitor obciążenia systemu dla modułu LB systemu ANSA

3.1 Założenia

Zgodnie z wcześniejszymi rozważaniami celem monitora obciążenia dla modułu LB jest:

- kontrolowanie obciążenia komputerów, na których zainstalowana jest ANSA,
- gromadzenie na ten temat jak najbardziej aktualnych informacji,
- udostępnianie ich programom, które je potrzebują.

Przyjęto koncepcję scentralizowanego modelu monitoringu to znaczy informacja na temat obciążenia poszczególnych węzłów systemu zbierana jest na jednym wyróżnionym komputerze. Monitor ten został napisany całkowicie w systemie ANSA. Jego częściami składowymi są więc odpowiednio skonstruowane kapsuły komunikujące się poprzez importowane przez siebie interfejsy. Opis tych kapsuł stanowi przedmiot niniejszego rozdziału.

3.2 Kapsuły systemu

3.2.1 MLB – Monitor Load Balancingu

Jest to główna kapsuła systemu monitoringu, jest ona uruchamiana na samym początku pracy systemu. Wynikiem jej dzialania jest:

- uruchomienie na każdym węźle kapsuły *Testera*, zbierającej dane na temat obciążenia tego komputera, na którym działa,
- stworzenie tablicy struktur zawierających informacje na temat stanu komputerów systemu ANSA,
- wyeksportowanie referencji do interfejsów umożliwiających operacje na tej tablicy (zapis, odczyt danych) i ich obsługa.

Aby wypełnić te wszystkie zadania w procedurze *body* kapsuły wykonywane są następujące akcje:

1. Odczytanie parametrów wywołania, określających wartości początkowe dwóch zmiennych monitora:
   - *Delay* – określa kwant czasu pomiędzy dwoma kolejnymi odczytami danych na temat obciążenia danego komputera (defaultowo przyjęto 5 sekund),
   - *Delimiter* – określa jaka zmiana tych danych spowoduje wysłanie ich do kapsuły *MLB* (defaultowo przyjęto 2 procent).
2. Wywołanie operacji *Tradera*, *Lookup* z nazwą usługi *Factory*, znajdującej wszystkie referencje do tej usługi. Umożliwia to określenie liczby i nazwy węzłów systemu ANSA.
3. Stworzenie, z odpowiednim parametrem współbieżności i wyeksportowanie interfejsów do usług oferowanych przez tę kapsułę.
4. Instalacja funkcji przechwytującej sygnał systemowy *kill*. W takim przypadku zainstalowana funkcja niszczy kapsuły *Testerów*, interfejsy i zwalnia przydzieloną pamięć.
5. Na podstawie znanej już liczby węzłów systemu przydzielenie pamieci do tablicy *Węzły* zawierającej struktury o postaci:

   ```
   typedef struct _InfStat{
            char hostname[20];/* Nazwa komputera                    */
            int  stan_Testera;/* Czy Tester poprawnie zainstalowany ? */
            TCapsule *capsule;/* Struktura z referencjami do Testera  */
            HostInfo stan;    /* Struktura z danymi o obciazeniu      */
        } InfoStat;
   ```

   gdzie *HostInfo* jest następującym rekordem danych opisujących obciążenie węzła:

   ```
   HostInfo : TYPE = RECORD [
           Index  :INTEGER, /* Numer informacji                  */
           CPU    :INTEGER, /* Procent zajetosci czau procesora  */
           Load   :INTEGER, /* Sredne obciazenie komputera       */
           Swap   :INTEGER  /* Ilosc procesow zapisanych na dysk */
                     ];
   ```
6. Instalacja *testerów* kolejno na wszystkich węzłach systemu przez funkcję *InstalujTester*, która przy pomocy znanej już referencji do usługi *Factory* na danym komputerze, tworzy kapsułę *testera* i inicjuje jego pracę, czyli odczyt stanu węzła.

W trakcie wykonywania tych wszystkich działań do *stderr* wypisywane są komunikaty o aktualnie wykonywanych akcjach i ewentualnych błędach. Po zakończeniu wykonywania procedury *body*, system monitoringu już pracuje, a informacje zbierane przez niego są dostępne dla wszystkich programów, które zaimportują referencje do interfejsu *SlmAcd* tej kapsuły.

3.2.2 Tester

Kapsuła ta uruchamiana jest przez *MLB* na wszystkich węzłach systemu ANSA. Po zainicjowaniu jej wywołaniem odpowiedniej funkcji jej interfejsu, w nieskończonej pętli odczytuje ona parametry obciążenia komputera, na którym działa. Jej działanie określają dwie zmienne: *Delay i Delimiter* odpowiedzialne odpowiednio za czas pomiędzy kolejnymi odczytami oraz za to czy zebrane dane zostaną przekazane do *MLB*. Ma to miejsce wtedy gdy zmiana obciążenia jest większa od założonego poziomu, obecnie decyduje o tym zmiana zajętości czasu procesora. Jeśli należy przesłać dane wywoływana jest odpowiednia funkcja interfejsu *SlmDc* kapsuły *MLB*.

3.2.3 PM – presentation manager

W końcowym etapie prac nad programem będzie to kapsuła, która w graficzny sposób, z użyciem interfejsu X Windows, prezentować będzie dane zebrane przez monitor. Obecnie jest to prosty program, który importuje referencje do interfejsu *SlmAcd* kapsuły *MLB* i przy pomocy funkcji tego interfejsu wypisuje dane o jednym określonym komputerze lub o całym systemie.

3.3 Opis interfejsów kapsuł

3.3.1 Interfejsy kapsuły *MLB*

Kapsuła ta posiada trzy interfejsy, z których każdy zawiera inny zbiór usług i jest wykorzystywany przez używające go programy w innym celu.

1. *SlmDc* – System loading monitor - Data collecting
   Interfejs ten oferuje funkcje służące do gromadzenia danych o obciążeniu komputerów, a więc funkcje zapisujące te dane do tablicy *Węzły*. Jest tworzony z współczynnikiem współbieżności równym ilości węzłów w systemie. Ma on postać:

```
SlmDc : INTERFACE =
BEGIN
-- Constructed data types
   HostInfo : TYPE = RECORD [
                     Index  :INTEGER,
                     CPU    :INTEGER,
                     Load   :INTEGER,
                     Swap   :INTEGER
                     ];
-- Operations
   SetHostInfo : OPERATION [
                             HNumber : INTEGER;
                             StatData: HostInfo ]
                 RETURNS [ ];
END.
```

W interfejsie tym zdefiniowano opisany już typ danych *HostInfo* oraz funkcję zapisującą do tablicy *Węzły* w wierszu *HNumber* dane z *StatData*. Oblicza ona również współczynniki szybkości zmian obciążenia danego hosta, jak również ich wartości średnie. Na obecnym etapie prac każdorazowe wywołanie tej funkcji kończy się wypisaniem nowej zawartości tablicy do *stderr*.

2. *SlmAcd* – System loading monitor - Access to collecting data
   Interfejs ten oferuje funkcje umożliwiajęce programą dostęp do zebranych przez monitor danych. Jest tworzony z współczynnikiem współbieżności równym 4. Ma on postać:

```
SlmAcd : INTERFACE =
NEEDS SlmDc;
BEGIN
-- Constructed data types
   SystemInfo : TYPE = SEQUENCE OF HostInfo;

-- Operations
   GetHostInfo    : OPERATION [
                          HName: STRING]
                    RETURNS [ HostData: HostInfo ];

   GetSystemInfo  : OPERATION [ ]
                    RETURNS [ SystemData: SystemInfo ];

END.
```

   W interfejsie tym zdefiniowano nowy typ danych *SystemInfo* jako sekwencję, czyli ciąg dowolnej długości rekordów *HostInfo*, może zawierać on dane na temat obciążenia całej sieci, wszystkich komputerów. Zdefiniowane funkcje umożliwiają uzyskanie informacji z tablicy *Węzły*, albo tylko o danym komputerze, którego nazwę podamy, lub o całym systemie.

3. *SlmCp* – System laoding monitor - Change parameters
   Interfejs ten oferuje funkcje umożliwiające zmianę parametrów działania monitora. Jest tworzony z współczynnikiem współbieżności równym 1. Ma on postać:

```
SlmCp : INTERFACE =
BEGIN
-- Constructed data types

-- Operations
   ChangeDelay : OPERATION [
                           Delay :INTEGER]
                 RETURNS [ ];

   ChangeDelimiter : OPERATION [
```

```
                                Delimiter : INTEGER]
                      RETURNS [ ];

END.
```

W interfejsie tym nie zdefiniowano żadnego nowego typu danych. Natomiast jego funkcje pzwalają zmienić wartości zmiennych monitora: *Delay* i *Delimiter* na podane wartości. Umożliwia to, przynajmniej w pewnym zakresie, sterowanie pracą systemu.

3.3.2 Interfejsy kapsuły *TESTER*

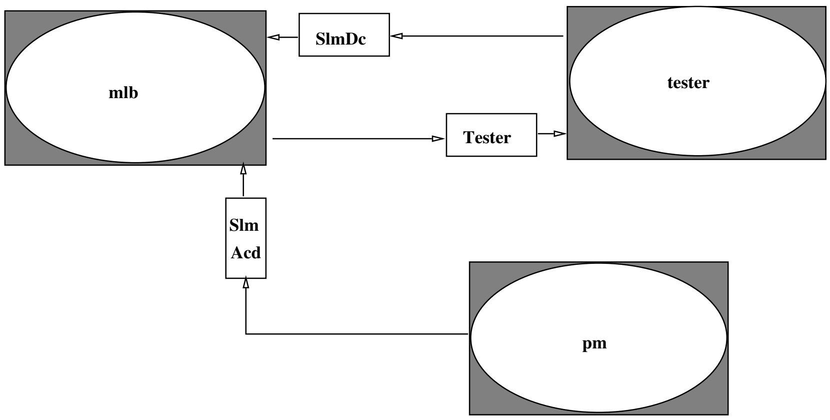
Kapsuła ta posiada tylko jeden interfejs.

1. *Tester* – umożliwia zainicjowanie pracy *testera* i wpływ na jego działanie. Interfejs ten tworzony jest z współczynnikiem współbieżności równym 2. Ma on postać:

```
Tester : INTERFACE =
NEEDS SlmDc
BEGIN
-- Operations
   InitTester  : ANNOUNCEMENT OPERATION [
                             HNumber  :INTEGER;
                             HName    :STRING;
                             Delay    :INTEGER;
                             Delimiter:INTEGER]
                 RETURNS [ ];

   GetHostInfo : OPERATION [
                             HName : STRING]
                 RETURNS [ HostInfo ];

   ChangeDelay : OPERATION [
                            Delay :INTEGER]
                 RETURNS [ ];

   ChangeDelimiter : OPERATION [
                                Delimiter : INTEGER]
                      RETURNS [ ];
END.
```

W interfejsie tym nie zdefiniowano żadnej nowej struktury danych. Pierwsza z udostępnianych funkcji inicjuje proces zbierania przez *testera* informacji o obciążeniu. Argumentami jej wywołania są: przydzielony komputerowi numer, jednocześnie numer wiersza odpowiadającego mu w tablicy *Węzły*, nazwa komputera oraz dwie wartości inicjujące zmienne *Delay* i *Delimiter*. Funkcja ta jest typu *ANNOUNCMENT* co oznacza, że wywołująca ją kapsuła nie musi czekać na jej zakończenie, w rzeczywistości nigdy się ona nie zakończy, ponieważ w jej wnętrzu wykonuje się nieskończona pętla odczytująca obciążenie węzła. Kolejna funkcja umożliwia odczytanie przez zewnętrzny program aktualnych danych o stanie tego, konkretnego komputera. Dwie ostatnie funkcje pozwalają zmienić wartości parametrów pracy tej kapsuły.

3.4 Komunikacja miedzy kapsułami



Rysunek 3.1: Powiązania pomiędzy kapsułami monitora

Współdzianie kapsuł monitora w przepływie informacji o obciążeniu systemu od kapsuł *TESTER* do *MLB* a stąd na zewnętrz do kapsuły *PM*, przedstawiono na rys. 3.1. Oczywiście informacje te mogą być wykorzystywane również przez inne kapsuły, w szczególności przez moduł zajmujący się takim rozłożeniem kapsuł na poszczególnych węzłach aby zminimalizować różnice w obciążeniu komputerów.

3.5 Metoda uzyskiwania informacji o obciążeniu węzłów

Obecnie do uzyskiwania parametrów obciążenia komputera używana jest funkcja *rstat*, która zwraca strukturę *statstime*. Struktura ta zawiera wszystkie możliwe dane o stanie hosta. Na ich podstawie i danych z poprzedniego odczytu obliczane są przez funkcje zdefiniowane w pliku *fstat.c* parametry obciążenia, wyrażające się najczęście w procentach wykorzystania danego zasobu komputera w ciągu czasu, który minął między tymi dwoma pomiarami. Obecna metoda wydaje się być jednak niezbyt pasująca dla programu napisanego w ANSA, ponieważ wykorzystuje mechanizm *RPC*. Dlatego są podejmowane próby zastępienia wywołania *rstat* bezpośrednim odczytem tych informacji z jądra systemu SUNOS.

3.6 Uwagi

Ponieważ głównym celem tworzonego MLB jest dostarczanie danych dla modułu LB, może okazać się że obecną wersję trzeba będzie wzbogacić o możliwść dostarczania innych danych o działaniu systemu oraz uzupełnienie jego funkcji. Zbudowany jednak został główny szkielet komunikacyjny MLB umożliwiający w stosunkowo łatwy sposób jego rozbudowę i modyfikacje.

3.7 Pliki systemu

Poniżej przedstawiono wszystkie pliki, z ktòrych składa się napisany MLB. Są to zaròwno pliki napisane w języku ANSA, jak i pliki z zbiorami funkcji pomocniczych w języku C. Odpowiednią kompilację i konsolidację ich wszystkich zapewnia odpowiednio skonstruowany plik imakefile.

- Plik imakfile — dodatek A.
- Pliki z interfejsami: SlmDc.idl SlmAcd.idl SlmCp.idl Tester.idl
  Podane uprzednio.
- Pliki kapsuł: *MLB*, *TESTER*, *PM* — dodatki B, C, D.
- Plik w ANSA z definicjami funkcji pomocniczych — dodatek E.
- Pliki z predefinicjami — dodatek F.
- Plik z definicjami funkcji statystycznych — dodatek G.

W wyniku kompilacji wszystkich tych plików uzyskujemy programy wykonywalny: mlb i pm.

3.8 Weryfikacja działania MLB

Przedstawiony oprogramowanie MLB zostało uruchomione i przetestowane na sieci stacji roboczych SUN. Testy wykonywano na zespole komputerów, który liczył od 1 do kilku stacji roboczych. System pracował prawidłowo, nie stwierdzono nieprawidłowości w jego działaniu, informacje na temat obciążenia bez zakłóceń były zbierane przez MLB. Można było regulować intensywność napływania informacji poprzez podanie odpowiedniech wielkości parametrów, progu i czasu przy uruchomieniu programu. Również zakończenie pracy przez MLB i niszczenie jego kapsuł usytuowanych na różnych węzłach kończyło się sukcesem. Uruchamiany system nie wpływał w widoczny sposób na obciążenie systemu.

3.9 Prace obecne

Po napisaniu MLB prace koncentrują się obecnie na napisaniu SLB, zaprojektowaniu dla niego różnych strategii podejmowania decyzji oraz napisaniu preprocesora zmieniającego pliki dpl. Zawansowanie tych prac jest już dość znaczne i wkrótce rozpoczną się badania nad efektywnością różnych wariantów pracy modułu LB. Wyniki tych prac będą przedmiotę osobnego opracowania.

Literatura

[1] *ANSA — Reference Manual.*, Cambridge (1990).

[2] P. Amaral, C. Jacquemot, P. Jensen, R. Lea, A. Mirowski, *Transparent object migration in COOL–2.* Chorus Systemes (1992).

[3] P. Dickman, *Effective Load Balancing in a Distributed Object Support Operating Systems.* University of Cambridge (1991).

[4] P. Dickman, *Matching Dinamic Object Clumps to Current Load Imbalances: A Proposal.* ERCIM and INESC (1992).

[5] D.Epema, *A New Objective and Performance Metric for Load Balancing Policies in Distributed Systems.* Delf University of Technology (1989).

[6] E.Jul, H. Levy, N. Hutchinson, A. Black, *Fine-Grain Mobility in the Emerald System.* ACM Transactions on Computer Systems, Vol 6(1) 109 — 133 (1988).

[7] M.Ibanez-Espiga, M. Wiliams, S. Zhou, *Data Placement Strategy for Parallel Database System.* Heriot-Watt University in Edinburgh (1991).

[8] C. Gary Rommel, *The Probability of Load Balancing Success in a Homogenesous Network.* IEEE Tran. on Software Engineering, Vol 17(9) 922 — 933 (1991).

[9] A. Gerns, J. Lie, S. Steiner, G. Stiege , *A Testbed for the Evaluation and the Comparison of Object Migration Strategies.* Universitat Hildesheim (1991).

[10] R. Krahl, F. Schon, *Towards an Integration of Monitoring and Load Balancing by Object Orientation.* Technical University of Berlin (1992).

[11] D. Milojičić, W. Zint, *Coarse Grained Load Distribution on the top Mach Microkernel.* University of Kaiserslautern (1992).

[12] N. Shiviratri, P. Krueger, M. Singhal, *Load Distributing for Locally Distributed Systems.* Computer, Vol 25(12) 33 – 44 (1992).

[13] R. Trehan, A. Morishita, *Multi-Layered Addressing and Address Propagation for Distributed Concurrent Object Oriented 'C'.* Toshiba Research and Development Center (1992).

Dodatek A

imakefile

Plik imakefile dla oprogramowania MLB ma postać :

```
        DEFINES =
       INCLUDES =
       IDLFLAGS =
       DPLFLAGS =
       LINTLIBS = $(LINTANSALIB) -lc
       LOCALLIB = $(ANSALIB) $(ANSASTUBLIB)
      LOCALLIBD = $(ANSALIBD) $(ANSASTUBLIB)

IDLFILES = SlmDc.idl SlmAcd.idl Tester.idl SlmCp.idl
SIFFILES = SlmDc.sif SlmAcd.sif Tester.sif SlmCp.sif
DPLFILES = mlb.dpl tester.dpl fundata.dpl pm.dpl
SRCS     = fstat.c
HDRS     = fstat.h fundata.h
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES) $(SRCS) $(HDRS)
Mlb_OBJS = mlb.o mSlmDc.o mSlmAcd.o cTester.o sSlmDc.o
sSlmAcd.o sSlmCp.o fundata.o
Tes_OBJS = tester.o mSlmDc.o cSlmDc.o sTester.o fstat.o
Pm_OBJS  = pm.o mSlmAcd.o cSlmAcd.o mSlmDc.o

RpcSvcLIB   = /lib/librpcsvc.a

PROGS =  mlb tester pm

# compile idl files
all:: $(SIFFILES)

# compile dpl files
DPLDepend(mlb)
DPLDepend(tester)
DPLDepend(fundata)
DPLDepend(pm)
```

```
all:: $(PROGS)

IDLDepend(SlmDc)
IDLDepend(SlmAcd)
IDLDepend(SlmCp)
IDLDepend(Tester)

SingleProgramTarget(mlb,$(Mlb_OBJS),$(LOCALLIB),)
SingleProgramTarget(tester,$(Tes_OBJS) $(RpcSvcLIB),$(LOCALLIB),)
SingleProgramTarget(pm,$(Pm_OBJS),$(LOCALLIB),)

InstallIDL(SlmDc)
InstallIDL(SlmAcd)
InstallIDL(SlmCp)
InstallIDL(Tester)

InstallProgram(mlb,$(TEMPLATEDIR)/mlb)
InstallProgram(tester,$(TEMPLATEDIR)/tester)
InstallProgram(pm,$(TEMPLATEDIR)/pm)

NormalLintTarget(*.c)
DependTarget()

IDLCleanList($(IDLFILES))
DPLCleanList($(DPLFILES))
```

Dodatek B

mlb.dpl

Plik źródłowy kapsuły MLB.

```
/* dolanczane pliki */
#include "AWstdio.h"

#include <string.h>
#include "fundata.h"

/* stale programu */
#define NIEAKTUALNE         0
#define Bad                 0
#define Good                1
#define SlmAcd_Concurrency  4
#define DefaultDelay        5
#define DefaultDelimiter    2 /* procent */

/* uzywane interfejsy */
! USE Trader
! USE Capsule
! USE Factory
! USE Object

! USE Tester
! USE SlmDc
! USE SlmAcd
! USE SlmCp

/* deklaracja wskaznikow do interfejsow */
! DECLARE { FacRef } : Factory CLIENT
! DECLARE { CapRef } : Capsule CLIENT
! DECLARE { ObjRef } : Object  CLIENT

! DECLARE { ir_SlmDc    } :SlmDc  SERVER
```

```
! DECLARE { ir_SlmAcd   } :SlmAcd SERVER
! DECLARE { ir_SlmCp    } :SlmCp SERVER
! DECLARE { Res.data[0] } :Tester  CLIENT

/* referencje do Factory,  Capsule,  Object */
ansa_InterfaceRef FacRef;
ansa_InterfaceRef CapRef;
ansa_InterfaceRef ObjRef;

/* identyfikator kapsuly i referencje do udostepnianych
    przez obiekt uslug */
ansa_CapsuleId Cid;
InstantiateResult Res;

/* referencje do SlmDc, SlmAcd */
ansa_InterfaceRef ir_SlmDc, ir_SlmAcd, ir_SlmCp;

/* rezultat operacji Lookup */
LResult Lookup_Res;

/* struktura przechowujaca dane na temat kapsuly Testera */
typedef struct {
            ansa_InterfaceRef FacRef;
            ansa_InterfaceRef CapRef;
            ansa_InterfaceRef ObjRef;
            ansa_CapsuleId    Cid;
            ansa_InterfaceRef TesRef;
        } TCapsule;

/* struktury przechowujaca dane statystyczne o wezle */
typedef struct _InfStat{
            char hostname[20];
            int  stan_Testera;
            TCapsule *capsule;
            HostInfo stan;
        } InfoStat;

/* ilosc wezlow w systemie */
int IleWezlow;

/* wskaznik na tablice przechowujaca dane statystyczne o wezlach */
InfoStat *Wezly;

/* bufor properties */
#define PROPSIZE 1024
char propbuf[PROPSIZE];
```

```
/* opoznienie */
int Delay;

/* ogranicznik */
int Delimiter;

/* czy pierwszy raz wywolano signal_handler */
int first;

/* Funkcja usuwajaca Tester z danego wezla */
void KillTester(Index)
int Index;
{
 int i;
 ansa_Boolean r;

 /* Interfejs Wezly.TCapsule.TesRef nie bedzie wiecej uzywany */
 ifref_copyRef(&(Res.data[0]), &(Wezly[Index].capsule->TesRef));
!{}  <- Res.data[0]$Discard

 /* Niszcz obiekt ObjRef
 ifref_copyRef(&(ObjRef), &(Wezly[Index].capsule->ObjRef));
 !{}  <- ObjRef$Terminate() */

 /* Interfejs ObjRef nie bedzie wiecej uzywany
 !{}  <- ObjRef$Discard */

 /* Interfejs CapRef nie bedzie wiecej uzywany */
 ifref_copyRef(&(CapRef), &(Wezly[Index].capsule->CapRef));
!{}  <- CapRef$Discard

 /* Niszcz kapsule o identyfikatorze Cid */
 ifref_copyRef(&(FacRef), &(Wezly[Index].capsule->FacRef));
!{r} <- FacRef$Terminate(Wezly[Index].capsule->Cid)

 if (!r)
  (void)fprintf(stderr, "Nie zniszczono kapsuly na: %s\n
            Numer procesu: %d\n", Wezly[Index].hostname,
              Wezly[Index].capsule->Cid);

 /* Zwolnij pamiec */
 for (i=0; i<IleWezlow; i++)
  free((char *)Wezly[i].capsule);
 cfree((char *)Wezly);
}

/* Funkcja usuwajaca wszystkie Testery i interfejsy */
```

```
void KillAll()
{
 int i;

 fprintf(stderr, "\n     Koniec   \n");

 /* usun kolejno wsystkie Testery */
 for (i=0; i<IleWezlow; i++)
  if (Wezly[i].stan_Testera)
   KillTester(i);

 /* usun interfejs SlmDc */
!{} <- traderRef$Withdraw(ir_SlmDc)
!{} :: SlmDc$Destroy(ir_SlmDc)

 /* usun interfejs SlmAcd */
!{} <- traderRef$Withdraw(ir_SlmAcd)
!{} :: SlmAcd$Destroy(ir_SlmAcd)

 /* Ustaw parametr zwalniania pamieci */
 stub_setFreeCltMem(ansa_TRUE);
}

/* Funkcja przechwytujaca sygnal kill */
static SIGNAL_RETURN_TYPE signal_handler( sig )
int sig;
{

 if (first){
  first = 0;
  KillAll();
 }

 system_catch( sig );
 return SIGNAL_RETURN_VALUE;
}

/* Funkcja znajduje nazwe hostu, na ktorym znajduje sie dana
     usluga Factory */
char *h_name(i)
 int i;
 {
  int j;

  for(j=0;(strcmp(Lookup_Res.u.u_S_S.data[i].ifr_props.data[j].p_name
             ,"Node")); j++);
```

```
  return (Lookup_Res.u.u_S_S.data[i].ifr_props.data[j].p_value);
 }

/* Funkcj instalujaca Testery na danym wezlach */
int instaluj_Tester(i)
 int i;
 {
  /* Znajdz nazwe wezla */
  strncpy(Wezly[i].hostname, h_name(i), 20);

  /* Kopjuj aktualny interfejs Factory */
  ifref_copyRef(&FacRef, &(Lookup_Res.u.u_S_S.data[i].ifr_ref));

  /* Tworz kapsule na odpowiednim wezle */
! {CapRef, Cid} <- FacRef$Instantiate \
         ("/home/tulip/uszok/mlb", "tester", "", "", nullRef, 0)


  /* Sprawdz czy Factory utworzyla nowa kapsule */
  if (!Cid){
   fprintf(stderr, "Blad: utworzenia nowej kapsuly na wezle:%s\n",
            Wezly[i].hostname);
   return(Bad);
  }

  /* Tworz obiekt w kapsule */
! {ObjRef, Res} <- CapRef$Instantiate(nullRef, nullRef,\
                          "Tester", "", "")


  /* Sprawdz czy utworzono obiekt w kapsule */
  if (ObjRef.length == 0){
   fprintf(stderr, "Blad: utworzenia obiektu w kapsule
              na wezle:%s\n", Wezly[i].hostname);
   return(Bad);
  }

  /* Inicjuj Tester */
! {} <- Res.data[0]$InitTester(i, Wezly[i].hostname, Delay,
               Delimiter)

  /* Przydziel pamiec i wypelnij strukture TCapsula dla Wezla i */
  Wezly[i].capsule = (TCapsule *)malloc(sizeof(TCapsule));
  ifref_copyRef(&(Wezly[i].capsule->FacRef), &FacRef);
  ifref_copyRef(&(Wezly[i].capsule->CapRef), &CapRef);
  ifref_copyRef(&(Wezly[i].capsule->ObjRef), &ObjRef);
  ifref_copyRef(&(Wezly[i].capsule->TesRef), &(Res.data[0]));
```

```
  Wezly[i].capsule->Cid = Cid;

  /* Zwroc Good */
  return(Good);
 }

/* Funkcja wypisujaca wzor wywolania mlb */
void pomoc()
 {
  (void)fprintf(stderr," Poprawne wywolanie:\n \t\t
                             mlb [delay delimiter]");
 }

/* Funkcja pobierajaca opcje wejsciowe */
void opcje(argc, argv)
 int argc;
 char *argv[];
 {
  int Pomoc;

  Pomoc = 0;
  if (argc == 3){
   if (!(Delay = atol(argv[1]))){
    Delay = DefaultDelay;
    (void)fprintf(stderr, "Przyjeto Delay= %l\n", DefaultDelay);
    Pomoc = 1;
   }
   if (!(Delimiter = atol(argv[2]))){
    Delimiter = DefaultDelimiter;
    (void)fprintf(stderr, "Przyjeto Delimiter= %l\n",
                   DefaultDelimiter);
    Pomoc = 1;
   }
  }
  else
   Pomoc = 1;

  if (Pomoc) pomoc();
  return;
 }

/* Funkcja glowna monitora load balancingu */
void body(argc, argv, envp)
 int argc;
 char *argv[];
 char *envp[];
 {
```

```
  int i;

  (void)fprintf(stderr,"\n\n
                     *** Monitor Load Balancing'u ***\n");

  /* Pobierz parametry wejsciowe */
  if (argc > 1)
   opcje(argc, argv);
  else{
   Delay = DefaultDelay;
   Delimiter = DefaultDelimiter;
  }

  /* Przeszukaj baze Trader'a w poszukiwaniu uslug Factory */
! {Lookup_Res} <- traderRef$Lookup("Factory", "/", "", Lookup_All)

  IleWezlow = Lookup_Res.u.u_S_S.length;

  (void)fprintf(stderr,"\n Liczba wezlow, na ktorych
                zainstalowana jest ANSA:");
  (void)fprintf(stderr," %d\n  Sa to wezly:\n", IleWezlow);

  /* Wypisz te wezly */
  for (i = 0; i < IleWezlow; i++)
   (void)fprintf(stderr,"     Wezel%i:   %s\n",i+1, h_name(i));
  (void)fprintf(stderr, "\n");

  /* Jesli sa wezle z usluga Factory */
  if (IleWezlow > 0){

   /* Zwieksz ilosc taskow */
   nucleus_tasks(1, 0);

   /* Ustaw parametr dla zwalniania pamieci */
   stub_setFreeCltMem( ansa_FALSE );

   /* Instaluj interfejs SlmDc z mozliwa wspolbieznoscia
          rowna liczbie wezlow */
!  {ir_SlmDc} :: SlmDc$Create(IleWezlow)
   (void) system_init_properties(propbuf, PROPSIZE, argc, argv);
!  {} <- traderRef$Export("SlmDc", "/ansa/testservices", \
            propbuf, ir_SlmDc)

   (void)fprintf(stderr," Interfejs SlmDc zainstalowany. \n\n");

   /* Instaluj interfejs SlmAcd z mozliwa wspolbieznoscia rowna
       SlmAcd_Concurrency */
```

```
!  {ir_SlmAcd} :: SlmAcd$Create(SlmAcd_Concurrency)
   (void) system_init_properties(propbuf, PROPSIZE, argc, argv);
!  {} <- traderRef$Export("SlmAcd", "/ansa/testservices", \
            propbuf, ir_SlmAcd)

   (void)fprintf(stderr," Interfejs SlmAcd zainstalowany. \n\n");

   /* Instaluj interfejs SlmCp z mozliwa wspolbieznoscia
       rowna jeden */
!  {ir_SlmCp} :: SlmCp$Create(1)
   (void) system_init_properties(propbuf, PROPSIZE, argc, argv);
!  {} <- traderRef$Export("SlmCp", "/ansa/testservices", \
            propbuf, ir_SlmCp)

   (void)fprintf(stderr," Interfejs SlmCp zainstalowany. \n\n");


   /* Instaluj funkcje przechwytujaca sygnal kill */
   first = 1;
   system_signal(signal_handler);

   /* Przydziel pamiec do tablicy Wezly */
   Wezly = (InfoStat *)calloc(IleWezlow, sizeof(InfoStat));

   /* Inicjuj pole Index w strukturze InfoStat*/
   for (i = 0; i<IleWezlow; i++)
    Wezly[i].stan.Index = NIEAKTUALNE;

   (void)fprintf(stderr, " Rozpoczynam instalacje Testerow. \n\n");

   /* Instaluj Testery na wszystkich wezlach ANSY */
   for (i = 0; i < IleWezlow; i++)
    /* Czy Tester poprawnie zainstalowany */
    if (!(Wezly[i].stan_Testera = instaluj_Tester(i))){
     (void)fprintf(stderr, " Sprawdz wezel:%s\n", Wezly[i].hostname);
     (void)fprintf(stderr, "  Nie zainstalowano na nim Testera\n");
    }
    else
     (void)fprintf(stderr, " Zainstalowano Tester na wezle: %s\n",
                  Wezly[i].hostname);

  }
  else{
   (void)fprintf(stderr," Blad: ANSA nie jest poprawnie
                 zainstalowana !\n");
   (void)fprintf(stderr,"   Serwis Factory nie jest dostepny
                 na zadnym wezle\n");
```

```
  }

  (void)fprintf(stderr,"\n Koniec Instalacji Testerow\n");
  (void)fprintf(stderr,"\n Modul load - balancing'u pracuje\n");
}

/*
               Funkcje interfejsu SlmDc
*/

/* Funkcja drukujaca stan wezlow systemu ANSA */
void print_stan_wezlow()
 {
  int i;

  (void)fprintf(stderr, " Nazwa Wezla \t Nr informacji \t
        CPU \t Load \t Swap\n");
  for(i = 0; i < IleWezlow; i++)
   if (Wezly[i].stan.Index >NIEAKTUALNE)
    if (strlen(Wezly[i].hostname) > 4)
     (void)fprintf(stderr, "  %s \t %i \t\t %i \t %i \t %i\n",
            Wezly[i].hostname,Wezly[i].stan.Index,Wezly[i].stan.CPU,
            Wezly[i].stan.Load, Wezly[i].stan.Swap);
    else
     (void)fprintf(stderr, "  %s \t\t %i \t\t %i \t %i \t %i\n",
            Wezly[i].hostname,Wezly[i].stan.Index,Wezly[i].stan.CPU,
            Wezly[i].stan.Load, Wezly[i].stan.Swap);
  fprintf(stderr, "\n");
  return;
 }

/* Funkcja zmieniajaca dane w tablicy okreslajacej stan hosta na
     dane aktualne */
int SlmDc_SetHostInfo(_attr, HostNumber, StatData)
 ansa_InterfaceAttr *_attr;
 ansa_Integer HostNumber;
 HostInfo StatData;
 {
  /* Uaktualnij dane w tablicy Wezly na odpowiedniej pozycji */
  copyHI(&(Wezly[HostNumber].stan), &StatData);

  /* Wyswietl stan wezlow */
  print_stan_wezlow();

  return(1);
 }
```

```
/************** Koniec funkcji interfejsu SlmDc **************/

/*
               Funkcje interfejsu SlmAcd
*/

int SlmAcd_PmRegistration(_attr, ContextName)
 ansa_InterfaceAttr *_attr;
 ansa_String ContextName;
 {
   /* Na razie nie zaimplementowane */
 }

int SlmAcd_GetHostInfo(_attr, HName, result)
 ansa_InterfaceAttr *_attr;
 ansa_String HName;
 HostInfo *result;
 {
  int i;

  /* Szukaj hosta w tablicy Wezly */
  for(i=0; strcmp(HName, Wezly[i].hostname) && (i < IleWezlow);
       i++);

  /* Jesli znalazles */
  if (i<IleWezlow)
   /* Kopjuj dane statystyczne o tym wezle */
   copyHI(result, &(Wezly[i].stan));
  else
   result->Index = 0;

  return SuccessfulInvocation;
 }

int SlmAcd_GetSystemInfo(_attr, result)
 ansa_InterfaceAttr *_attr;
 SystemInfo *result;
 {
  int i;

  /* Przydziel pomiec */
  result->data=(HostInfo*)calloc(IleWezlow, sizeof(HostInfo));

  /* Kopjuj dane statystyczne o wszystkich wezlach */
  for (i=0; i < IleWezlow; i++){

   copyHI(&(result->data[i]), &(Wezly[i].stan));
```

```
  }

  result->length = IleWezlow;

  return SuccessfulInvocation;
 }

/************** Koniec funkcji interfejsu SlmAcd *************/

/*
                Funkcje interfejsu SlmCp
*/

/* Funkcja zmieniajaca Delay testerow na wszystkich wezlach */
int SlmCp_ChangeDelay(_attr, Delay)
 ansa_InterfaceAttr *_attr;
 ansa_Integer Delay;
 {
  int i;

  /* Dla wszystkich wezlow zmien Delay */
  for (i=0; i < IleWezlow; i++){
   /* Kopjuj referencje do testera na danym wezle */
   ifref_copyRef(&(Res.data[0]), &(Wezly[i].capsule->TesRef));

   /* Wywolaj operacje zmiany Delay */
!  {} <- Res.data[0]$ChangeDelay(Delay)
  }
  return SuccessfulInvocation;
 }

/* Funkcja zmieniajaca Delimiter testerow na wszystkich wezlach */
int SlmCp_ChangeDelimiter(_attr, Delimiter)
 ansa_InterfaceAttr *_attr;
 ansa_Integer Delimiter;
 {
  int i;

  /* Dla wszystkich wezlow zmien Delay */
  for (i=0; i < IleWezlow; i++){
   /* Kopjuj referencje do testera na danym wezle */
   ifref_copyRef(&(Res.data[0]), &(Wezly[i].capsule->TesRef));

   /* Wywolaj operacje zmiany Delimiter'a */
!  {} <- Res.data[0]$ChangeDelimiter(Delimiter)
  }
```

```
  return SuccessfulInvocation;
 }

/************** Koniec funkcji interfejsu SlmCp **************/
```

Dodatek C

tester.dpl

Plik źródłowy kapsuły TESTER.

```
/* dolanczane pliki */
#include "AWstdio.h"
#include "fstat.h"

/* bufor properties */
#define PROPSIZE 1024
char propbuf[1024];

/* kapsula moze byc tworzona przy pomocy Factory */
! MANAGED Tester

/* uzywane interfejsy */
! USE Trader

! USE Tester
! USE SlmDc

/* deklaracja wskaznikow do interfejsow */
! DECLARE { p->ref } : Tester SERVER
! DECLARE { ir_SlmDc } : SlmDc CLIENT

/* struktura przechowujaca stan objectu */
typedef struct _ObjState{
          ansa_InterfaceRef ref;
          ansa_Boolean export;
        } ObjState;

/* Funkcja tworzaca obiekt Tester  */
ansa_StatePtr Create_Tester_Object(argc, argv, envp, results)
 int argc;
 char *argv[], *envp[];
```

```
 InstantiateResult *results;
 {
  ObjState *p;

  /* Stworz obiekt */
  p =(ObjState *)malloc(sizeof(ObjState));
! {p->ref} :: Tester$Create(3)

  /* Jesli odpowiednia zmienna ustawiona to wyeksportuj
      referencje do niego */
  if(system_getenv("ANSA_MANAGED_EXPORT", envp) !=(char *)0){
    p->export = ansa_TRUE;
    system_init_properties(propbuf, PROPSIZE, argc, argv);
!   {} <- traderRef$Export("Tester", "/ansa/testservices", \
              propbuf, p->ref)
   }
  else
   p->export = ansa_FALSE;

  /* Zwieksz ilosc taskow */
  nucleus_tasks(1, 0);

  /* Przygotuj dane wyjsciowe */
  results->length = 1;
  results->data = &(p->ref);

  return(ansa_StatePtr)p;
 }

/* Funkcja niszczaca obiekt Tester */
ansa_Boolean Destroy_Tester_Object(state)
 ansa_StatePtr state;
 {
  ObjState *p;

  p =(ObjState *)state;
  if(p->export == ansa_TRUE)
!  {} <- traderRef$Withdraw(p->ref)
! {} :: Tester$Destroy(p->ref)
  free((char *)p);

  return ansa_TRUE;
 }
/*
                 Funkcje interferjsu Tester
*/
/* Zmienne globalne interfejsu */
```

```
ansa_InterfaceRef ir_SlmDc;
HostInfo DaneStat;
int _Delay;
int _Delimiter;
int Index;

/* Funkcja zwracajaca aktualne dane o wezle */
int Tester_GetHostInfo(_attr, HName, result)
 ansa_InterfaceAttr *_attr;
 ansa_String HName;
 HostInfo *result;
 {
  /* Pobierz aktualny stan hosta */
  getstat(HName);

  /* Przygotuj dane do przekazania */
  result->Index = Index;
  result->CPU   = fcpu();
  result->Load  = fload();
  result->Swap  = fswap();

  /* Wszystko oky */
  return SuccessfulInvocation;
 }

/* Funkcja inicjujaca Tester */
int Tester_InitTester(_attr, HNumber, HName, Delay, Delimiter)
ansa_InterfaceAttr *_attr;
 ansa_Integer HNumber;
 ansa_String  HName;
 ansa_Integer Delay;
 ansa_Integer Delimiter;
 {
  int j;

  /* Inicjuj _Delay */
  _Delay = Delay;

  /* Inicjuj _Delimiter */
  _Delimiter = Delimiter;

  /* Import referencji do SlmDc */
! {ir_SlmDc} <- traderRef$Import("SlmDc", "/", "")

  /* inicjuj Index */
  Index = 0;
```

```
  /* Inicjuj hstat i DaneStat*/
  inithstat(HName);

  /* Nieskonczona petla, w ktorej badany jest stan wezla */
  while(1) {
   /* Czekaj */
   (void)sleep(Delay);

   /* Pobierz aktualny stan hosta */
   getstat(HName);

   /* Jesli zmiana obciazenia CPU wieksza niz ogranicznik to
      przekaz dane do Mlb */
   if((abs(DaneStat.CPU - fcpu()) > _Delimiter)  || (Index<1)){
    DaneStat.Index = ++Index;
    DaneStat.CPU   = fcpu();
    DaneStat.Load  = fload();
    DaneStat.Swap  = fswap();

    /* Przeslij zebrane dane */
!   ir_SlmDc$SetHostInfo(HNumber, DaneStat)
   }
  }
 }

/* Funkcja zmieniajaca _Delay */
int Tester_ChangeDelay(_attr, Delay)
 ansa_InterfaceAttr *_attr;
 ansa_Integer Delay;
 {
  /* Zmiana zmiennej: _Delay */
  _Delay = Delay;

  return SuccessfulInvocation;
 }

/* Funkcja zmieniajaca _Delimiter */
int Tester_ChangeDelimiter(_attr, Delimiter)
 ansa_InterfaceAttr *_attr;
 ansa_Integer Delimiter;
 {
  /* Zmiana zmiennej: _Delimiter */
  _Delimiter = Delimiter;

  return SuccessfulInvocation;
 }
```

Dodatek D

pm.dpl

Plik źródłowy kapsuły PM.

```
/* dolanczane pliki */
#include "AWstdio.h"

/* uzywane interfejsy */
! USE Trader

! USE SlmAcd

/* deklaracja wskaznikow do interfejsow */
! DECLARE { ir_SlmAcd } : SlmAcd CLIENT

/* referencje do SlmAcd */
ansa_InterfaceRef ir_SlmAcd;

/* Funkcja glowna monitora load balancingu */
void body(argc, argv, envp)
 int argc;
 char *argv[];
 char *envp[];
 {
  int i;

  /* Import referencji do SlmAcd */
! {ir_SlmAcd} <- traderRef$Import("SlmAcd", "/", "")

  if (argc > 1){
   HostInfo Data;

   /* Pobierz dane */
!  {Data} <- ir_SlmAcd$GetHostInfo(argv[1])
```

```
   if (Data.Index)
    (void)fprintf(stderr, "  Name = %s \t Index = %i \t
             CPU =  %i \n", argv[1], Data.Index, Data.CPU);
   else
    (void)fprintf(stderr, "  Nie ma takiego pliku ! \n");
  }
  else{
   SystemInfo Data;

   /* Pobierz dane */
!  {Data} <- ir_SlmAcd$GetSystemInfo()

   /* Drukuj informacje */
   for(i = 0; i < Data.length; i++)
    (void)fprintf(stderr, "  Index = %i \t\t CPU =  %i \n",
                     Data.data[i].Index, Data.data[i].CPU);
 }
! ir_SlmAcd$Discard

  return;
 }
```

Dodatek E

fundata.dpl

Plik źródłowy funkcji pomocniczych.

```
/* dolanczane pliki */
#include "AWstdio.h"

/*uzywane interfejsy */
! USE SlmDc

/* funkcja kopjujaca struktury HostInfo */
void copyHI(destination, source)
HostInfo *destination, *source;
{
 destination->Index = source->Index;
 destination->CPU   = source->CPU;
 destination->Load  = source->Load;
 destination->Swap  = source->Swap;

 return;
}
```

Dodatek F

fstat.h, fundata.h

Plik fstat.h zawiera predefinicje funkcji dla pliku fstat.c.

```
/* zmienne globalne */
struct statstime *hstat[3];
/* funkcja inicjujaca tablice hstat */
extern void inithstat(/* char *HName */);
/* funkcja pobierajaca aktualny stan hosta */
extern void getstat(/* char *HName */);
/*   Funkcje Obliczen Statycznych Obciazenia */
extern int fcoll();
extern int fcpu();
extern int fdisks();
extern int ferr();
extern int fierr();
extern int fextern();
extern int fipkt();
extern int fload();
extern int foerr();
extern int fopkt();
extern int fpage();
extern int fpgpgin();
extern int fpkts();
extern int fpgpgout();
extern int fswap();
extern int fpswpin();
extern int fpswpout();
extern int fswt();
extern int fsys();
extern int fuser();
```

Plik fundata.h zawiera predefinicje funkcji dla pliku fundata.dpl.

```
void copyHI();
```

Dodatek G

fstat.c

Plik z funkcjami pomocniczymi w języku C.

```
/* dolanczane pliki */
#include <rpcsvc/rstat.h>
#include <sys/param.h>

/* makrodefinicje */
#define DIF(f)  (hstat[1]->f - hstat[0]->f)

/* zmienne globalne */
struct statstime *hstat[3];

/* funkcja inicjujaca tablice hstat */
void inithstat(HName)
char *HName;
 {
  /* Inicjuj tablice hstat */
  hstat[0] =(statstime *)malloc(sizeof(statstime));
  hstat[1] =(statstime *)malloc(sizeof(statstime));
  (void)rstat(HName, hstat[1]);
 }

/* funkcja pobierajaca aktualny stan hosta */
void getstat(HName)
char *HName;
 {
  /* Zmien tablice hstat */
  hstat[2] = hstat[0];
  hstat[0] = hstat[1];
  hstat[1] = hstat[2];

  /* Pobierz aktualny stan wezla */
  (void)rstat(HName, hstat[1]);
}
```

```
/*   Funkcje Obliczajace Statystyke */

int fcoll()
{
  return(DIF(if_collisions) / DIF(curtime.tv_sec));
}

int fcpu()
{
  int i;
  int t;
  int d[CPUSTATES];

  for(t = 0, i= 0; i < CPUSTATES; i++)
      t +=(d[i] = DIF(cp_time[i]));
  return(t ?(100 *(d[CP_USER]+d[CP_NICE]+d[CP_SYS])) / t : 0);
}

int fdisks()
{
  int i;
  int t;

  for(t = 0, i= 0; i < DK_NDRIVE; i++)
      t +=(DIF(dk_xfer[i]));
  return(t / DIF(curtime.tv_sec));
}

int ferr()
{
  return(fierr() + foerr());
}

int fierr()
{
  return(DIF(if_ierrors) / DIF(curtime.tv_sec));
}

int fintr()
{
  return(DIF(v_intr) / DIF(curtime.tv_sec));
}

int fipkt()
{
  return(DIF(if_ipackets) / DIF(curtime.tv_sec));
```

```
}

int fload()
{
  return((hstat[1]->avenrun[1]) * 100/FSCALE);
}

int foerr()
{
  return(DIF(if_oerrors) / DIF(curtime.tv_sec));
}

int fopkt()
{
  return(DIF(if_opackets) / DIF(curtime.tv_sec));
}

int fpage()
{
  return((DIF(v_pgpgin) + DIF(v_pgpgout)) / DIF(curtime.tv_sec));
}

int fpgpgin()
{
  return(DIF(v_pgpgin) / DIF(curtime.tv_sec));
}

int fpkts()
{
  return(fipkt() + fopkt());
}

int fpgpgout()
{
  return(DIF(v_pgpgout) / DIF(curtime.tv_sec));
}

int fswap()
{
  return(fpswpin() + fpswpout());
}

int fpswpin()
{
  return(DIF(v_pswpin) / DIF(curtime.tv_sec));
}
```

```
int fpswpout()
{
  return(DIF(v_pswpout)  / DIF(curtime.tv_sec));
}

int fswt()
{
  return(DIF(v_swtch) / DIF(curtime.tv_sec));
}

int fsys()
{
  int i;
  int t;
  int d[CPUSTATES];

  for(t = 0, i= 0; i < CPUSTATES; i++)
      t +=(d[i] = DIF(cp_time[i]));
  return(t ?(100 * d[CP_SYS]) / t : 0);
}

int fuser()
{
  int i;
  int t;
  int d[CPUSTATES];

  for(t = 0, i= 0; i < CPUSTATES; i++)
      t +=(d[i] = DIF(cp_time[i]));
  return(t ?(100 *(d[CP_USER] + d[CP_NICE])) / t : 0);
}
```