Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

DPE/UNIX – system przetwarzania rozproszonego

Robert Magdziarz
Nr indeksu: 140979

Praca magisterska napisana w Instytucie Informatyki
pod kierunkiem dr Janiny Mincer-Daszkiewicz

Warszawa, wrzesień 1997

Spis treści

1 WSTĘP

1.1 Geneza

Pionierskie systemy rozproszone powstawały w instytutach naukowych i służyły badaniom naukowym. Systemy rozproszone stanowią alternatywę dla komputerów klasy *MPP* (*masively parallel processors*) – najpotężniejszych dostępnych maszyn, używanych do rozwiązywania tak złożonych problemów jak modelowanie klimatu [8].

W przedsiębiorstwach, wprowadzanie kompleksowych zmian organizacyjnych, w tym reinżynieria procesów (ang. *Business Process Reengineering*), wiąże się zazwyczaj, poza redukcją personelu, z przeniesieniem uprawnień decyzyjnych na pracowników niższych szczebli w strukturze organizacji. Jako że delegacja odpowiedzialności musi wiązać się z przekazaniem środków umożliwiających realizację zwiększonego zakresu obowiązków, naturalną tego konsekwencją są zmiany w systemie informatycznym takiej organizacji polegające na zastępowaniu dużych komputerów (typu *mainframe*) sieciami komputerów osobistych o zwiększonych możliwościach [15]. To pierwszy krok w kierunku systemów rozproszonych, których przykładami są aplikcje pracy grupowej (ang. *groupware*) umożliwiające wielu użytkownikom jednoczesną pracę nad tymi samymi danymi lub wykorzystujące sieć do ułatwienia wspólnego korzystania z informacji (przetwarzanie dokumentów, układanie harmonogramów, obsługa poczty elektronicznej i grupowej pracy etapowej – ang. *workflow*).

Tak zwane przetwarzanie kooperacyjne realizowane jest w rozproszonych systemach komputerowych, w których pewna liczba komputerów wspólnie przetwarza program lub zadanie obliczeniowe. W tym celu wprowadza się podział obciążenia, wspólne wykorzystanie plików danych i zawartości pamięci, synchronizację, rozproszone mechanizmy zabezpieczania danych, metody gwarantujące poprawność i spójność rozproszonej informacji.

1.2 Omówienie tematu

Systemy rozproszone

Najogólniej, przetwarzanie rozproszone to przetwarzanie na wielu procesorach, choć z reguły wymaga się rówież, żeby przetwarzane dane były fizycznie rozproszone. Inne cechy tych systemów są następujące: rozproszenie ludzi i informacji, wysoka wydajność przy niedużych kosztach, łatwa rozszerzalność, dostępność mimo awarii, skalowalność [16].

Systemy przetwarzania rozproszonego są dosyć złożone. Wydaje się, że podstawowe tego przyczyny są następujące:

- często są to systemy rozległe, obejmujące dużą liczbę węzłów i połączeń – wewnętrzna struktura systemu jest złożona;
- działają na heterogenicznym sprzęcie wyposażonym w różnorodne oprogramowanie systemowe – standardy są pomocne, ale nie stanowią panaceum (i nie zawsze są przestrzegane);
- zachowanie się systemu rozproszonego jest prktycznie nieprzewidywalne ze względu na olbrzymią przestrzeń stanów.

Istnieją trzy podstawowe powody, dla których tworzy się takie systemy. Po pierwsze, ze względu na wykorzystanie zasobów – jeden system uzyskuje dostęp do zasobów wszystkich węzłów. Po drugie, podział i zrównoleglenie zadań z wykorzystaniem równomiernego dzielenia obciążeń może znacznie przyspieszyć obliczenia (zwiększyć przepustowość czy średni czas odpowiedzi). Po trzecie, system rozproszony jest niezawodny w sytuacjach awarii fragmentu systemu ze względu na zwielokrotnienie danych, sprzętu i oprogramowania.

System rozproszony składa się z następujących elementów [3]:

- platforma sieciowa z różnymi protokołami;
- interfejsy między aplikacjami pozwalające na formułowanie żądań do serwerów przy użyciu mechanizmów połączeniowych działających w czasie rzeczywistym lub systemy przekazywania komunikatów zapewniające dostarczenie odpowiedzi z większym dopuszczalnym opóźnieniem;
- katalogowe usługi nazewnictwa umożliwiające rejestrację zasobów i informacji oraz miejsc ich przechowywania;
- usługi czasowe zapewniające synchronizację zdarzeń w systemach przechowujących wzajemnie powiązane informacje;
- system zarządzania bazą danych realizujący zaawansowane funkcje, jak partycjonowanie i replikacja, pozwalający na rozproszenie danych i gwarantujący ich dostępność, poprawność i ochronę;
- mechanizmy bezpieczeństwa – weryfikacja tożsamości i autoryzacja oraz możliwość komunikacji pomiędzy systemami przy zachowaniu zasady "wzajemnego zaufania", co pozwala użytkownikowi na korzystanie z wielu serwerów baz danych bez konieczności potwierdzenia swojej tożsamości przy każdorazowej próbie dostępu do odległego zasobu.

Rozproszenie danych ma zalety:

- lokalni zarządcy mają do czynienia jedynie z danymi, których przeznaczenie i istotę sami znają;
- koszt utrzymywania połączeń można zredukować (istotne w sieciach rozległych) – dane, do których odwołania z pewnego węzła są częstsze można umieszczać bliżej niego;
- pewniejsza jest ochrona przed utratą danych i skutkami awarii – dzięki replikacji, która zapewnia także nadmiarowość;

Rozproszenie danych ma jednak kilka wad:

- synchronizacja treści może być bardzo skomplikowana zależnie od wymagań przezroczystości;
- trzeba uwzględnić niejednorodność sprzętu – może powstać problem różnych reprezentacji danych, np. liczb zmiennoprzecinkowych;
- zarządzanie dostępem do danych może być skomplikowane i może wymagać bardziej złożonych mechanizmów – zwłaszcza, jeśli system powstaje z połączenia systemów już istniejących, a nie jako nowe rozwiązanie zaprojektowane od podstaw;
- lokalni zarządcy mogą odnosić wrażenie utraty panowania nad danymi i niepewności co do ich rozmieszczenia, zawartości i organizacji.

Bezpieczeństwo danych w systemie rozproszonym zapewniają następujące mechanizmy [19]:

- weryfikacja tożsamości (ang. *authentication*) umożliwia autoryzowanemu użytkownikowi uzyskanie dostępu do dowolnego lokalnego systemu, znajdującego się w dowolnym miejscu sieci, już po jednorazowym zarejestrowaniu się; procedury weryfikacji pozwalają jednemu serwerowi działać w oparciu o założenie, że drugi serwer poprawnie zidentyfikował użytkownika;
- autoryzacja pozwala na nadanie użytkownikowi przywilejów dostępu do odległych zasobów zgodnie z pełnioną przez niego funkcją i umiejscowieniem w hierarchi zarządzania; użytkownik może należeć do określonych grup z właściwymi dla nich prawami dostępu;
- mechanizmy certyfikacji pozwalają serwerom na zachowanie relacji "wzajemnego zaufania" i działanie z pewnym poziomem zaufania do użytkowników, którzy uzyskują dostęp do ich zasobów;
- podpisy elektroniczne gwarantują użytkownikom autentyczność wiadomości odebranych od innych osób;
- filtrowanie pakietów przesyłanych przez most lub ruter podłączony do sieci rozległej.

Terminologia systemów przetwarzania rozproszonego

Przedstawmy teraz stosowaną terminologię wzorowaną na pracy [10].

Przez system przetwarzania rozproszonego rozumiemy zestaw oprogramowania pracującego na wielu *węzłach* połączonych siecią umożliwiający zdalne wykonywanie zadań i korzystanie z odległych

danych. Systemy takie realizuje się zazwyczaj w jednym z dwóch modeli: procesowym (system zarządza procesami na wielu maszynach) lub obiektowym (system zarządza przemieszczającymi się obiektami). System jest *przezroczysty*, jeśli to on, a nie użytkownik decyduje o wyborze węzła realizującego przetwarzanie, a użytkownik może nawet nie znać lokalizacji tego przetwarzania. System jest *generyczny*, jeśli zestaw usług może być dostosowywany do potrzeb. Przez ***usługę*** rozumiemy oprogramowanie mogące działać na jednym lub wielu węzłach, wykonujące pewną funkcję znaną *klientowi*. Usługi udostępniane są przez węzły nazywane *serwerami* i są realizowane w postaci programów wykonywalnych na tych serwerach. Wybór najlepszego serwera do zrealizowania określonej usługi i przekazywanie mu sterowania nazywamy *równoważeniem obciążenia*. Każdy węzeł może być również *serwerem danych* (*plików*), jeśli udostępnia pliki z lokalnego systemu plików innym węzłom. Oprogramowanie umożliwiające współpracę między klientami a serwerami może składać się z niezależnych *modułów*, takich jak: moduł *rozproszonego systemu plików* (zapewniający odległym klientom dostęp do lokalnych plików serwera), moduł umożliwiający *zdalne wykonanie* (realizujący żądania wykonania lokalnej usługi z innego węzła), moduł mierzący obciążenie węzła, moduł odnajdujący *węzeł optymalny* do wykonania danej usługi (jest to węzeł, który udostępnia tą usługę i na którym zostanie ona wykonana najsprawniej – kwestia miary tej „sprawności" należy do projektującego system), wreszcie moduł przechowujący listę usług udostępnianych przez dany serwer, która to lista jest fragmentem *rozproszonej bazy usług* całego systemu. Każdy moduł posiada pewien *interfejs* umożliwiający klientom komunikowanie się z modułem, zgodnie z pewnym *schematem komunikacji* właściwym dla danego modułu. Moduły systemu rozproszonego można zrealizować jako *demony*, czyli specjalne procesy działające przez cały czas pracy systemu. Schematy komunikacji mogą przyjąć przy realizacji systemu postać *protokołów komunikacyjnych* (najwyższego poziomu). Korzystanie z systemu przez użytkowników i administratorów jest możliwe przede wszystkim dzięki *programom narzędziowym*, a w celu udostępnienia możliwości systemu programistom tworzy się *biblioteki funkcji*.

Przedmiot pracy

Przedmiotem pracy jest generyczny system przezroczystego przetwarzania rozproszonego z równoważeniem obciążeń, zrealizowany w modelu procesowym. W istocie zaimplementowano te elementy systemu, które są niezbędne do funkcjonowania mechanizmu przezroczystego, zdalnego wykonywania usług na najbardziej odpowiednim węźle (tak zrealizowano równoważenie obciążenia). Autor pokazuje, że nie jest to trudne i niewielkim kosztem można stworzyć system, który dzięki swojej ogólności może służyć dalszym badaniom.

Praca obejmuje projekt systemu i metodyki tworzenia systemu, jego implementację, a także opis przykładów zastosowań. System zarządza usługami, które są udostępnianymi przez różne węzły programami. System nosi nazwę *DPE/UNIX* (*Distributed Processing Extension for UNIX* – rozszerzenie *UNIX*a o przetwarzania rozproszone).

Implementacja miała miejsce na platformie systemu *Linux*. Kod systemu zapisano w języku *C* i języku programowania powłoki Bourna *sh* (ang. *Bourne shell*).

1.3 Struktura i zakres pracy

W drugim rozdziale zatytułowanym ***Przegląd systemów*** autor przedstawia cztery znane systemy przetwarzania rozproszonego: *Linda*, *PVM*, *p4* i *Express*. Następnie omawia zagadnienie równoważenia obciążenia.

W trzecim rozdziale pt. *Metodyka tworzenia systemu* autor przedstawia argumenty przemawiające za użyciem metodologicznego podejścia przy konstrukcji systemu *DPE/UNIX* i systemów przetwarzania rozproszonego w ogólności. Istotnym elementem pracy jest bowiem prosta metodyka projektowania tego typu systemów, która w skrócie została opisana właśnie w trzecim rozdziale.

Czwarty rozdział, *Szkic definicji systemu*, jest opisem systemu jako całości oraz omówieniem jego składowych. Dokładniej przedstawione są konkretne rozwiązania i algorytmy. Celem rozdziału jest nieformalne, choć stosunkowo prezycyjne i obszerne, opisanie systemu. Kolejne etapy definicji systemu zawarto w rozdziałach od piątego do siódmego.

W piątym rozdziale, zatytułowanym *Analiza wymagań względem systemu*, autor omawia użytkowników systemu oraz specyfikuje ogólne wymagania względem systemu przetwarzania rozproszonego i szczegółowe względem systemu *DPE/UNIX*.

W szóstym rozdziale, *Specyfikacja logiczna systemu*, autor przeprowadza dekompozycję logicznego systemu na mniejsze elementy/podsystemy, takie jak: moduły, schematy komunikacyjne i narzędzia umożliwiające korzystanie z systemu i zarządzanie nim. Specyfikowane są nietrywialne algorytmy.

Kolejny rozdział pt. *Projekt fizyczny systemu* zawiera omówienie platformy, dla której przeznaczony jest system *DPE* oraz dostarcza specyfikacje potrzebne do zrealizowania składowych fizycznego systemu: demonów, protokołów komunikacyjnych, programów narzędziowych oraz bibliotek.

Ósmy rodział, *Zastosowania systemu*, to przegląd zastosowań systemu *DPE*.

Rozdział dziewiąty pt. *Podsumowanie* to zbiór różnych uwag i wniosków na temat systemu *DPE*, takich jak: wnioski z zastosowania inżynierii, wynikłe problemy, czy wskazówki dotyczące rozwoju systemu.

Praca zawiera również trzy dodatki (definicja metodyki, podręcznik administratora i podręcznik użytkownika) oraz bibliografię.

Każdy rozdział przedstawiający pewien etap konstrukcji systemu (od piątego do siódmego) rozpoczyna się przedstawieniem funkcji bieżącego etapu, struktury zadań do wykonania i struktury produktów etapu. Na potrzeby projektu systemu *DPE* metodyka została okrojona, ponieważ chodziło jedynie o wykazanie korzyści podejścia metodycznego.

2 PRZEGLĄD SYSTEMÓW

2.1 Przegląd systemów przetwarzania rozproszonego

Do najbardziej znanych systemów przetwarzania rozproszonego należą *Linda*, *PVM*, *p4* i *Express*.

2.1.1 System *Linda*

W systemie *Linda* zastosowano koncepcję wspólnej przestrzeni dla komunikacji procesów – przestrzeni krotek (ang. *tuple-space*). Jest to asocjacyjna pamięć rozproszona – alternatywa dla koncepcji przekazywania komunikatów i dla pamięci dzielonej, czyli dwóch najbardziej popularnych metod komunikacji międzyprocesowej. Charakterystyczne cechy tej pamięci to asocjacyjność i czytanie z usuwaniem zawartości. Możemy rozumieć przestrzeń krotek jako zbiór krotek (o różnych sygnaturach), które dostępne są procesom w rozproszonym systemie. Procesy te mogą wykonywać operacje *READ* (czytanie bez usuwanie), *INPUT* (czytanie z usuwaniem), *OUTPUT* (zapisanie nowej krotki) oraz wersje nieblokujące dwóch pierwszych operacji (*TRY_READ*, *TRY_INPUT*) [12].

Z punktu widzenia programisty *Linda* to zestaw programów narzędziowych i operacji umożliwiających programowanie równoległe i rozproszone [20].

W porównaniu z systemem *DPE/UNIX* w systemie *Linda* skupiono się na mechanizmie wymiany informacji między procesami – abstrakcyjnym, wysokopoziomowym. W systemie *DPE* komunikacja odbywa się na niskim poziomie: przy pomocy sygnałów, rozproszonych plików i potoków standardowego wejścia, wyjścia i błędów.

2.1.2 System *PVM*

System *PVM* (*Parallel Virtual Machine*) to jeden z pierwszych systemów dla maszyn o różnych architekturach i różnych reprezentacjach danych [14] – przyjęto jedynie, że są to stacje *UNIX*owe w sieci *TCP/IP*. System może być zainstalowany i uruchomiony przez zwykłego użytkownika (bez żadnych specjalnych uprawnień).

Wymiana informacji pomiędzy rozprosznymi elementami zadania (z reguły są to procesy) odbywa się przez asynchroniczne komunikaty. Dane mogą być kodowane w standardzie *XDR*. Każde zadanie może zostać zlecone do wykonania w jednym z trzech trybów: z automatycznym wyborem najlepszego węzła; z wyborem najlepszego węzła, ale wśród węzłów o wskazanej architekturze, wykonywane na określonym węźle.

Powstało liczne oprogramowanie rozszerzające możliwości systemu [20]: *DoPVM* (dla programujących w *C++*), *Xab* (graficzna nakładka), *XPVM* (nowe możliwości i graficzny interfejs w języku *Tcl/Tk*), *HeNCE* (*Heterogenous Network Computer Environment*). *HeNCE* to właściwie nowy system przetwarzania rozproszonego, choć został oparty głównie na systemie *PVM*. Posiada bardzo dobry interfejs graficzny oraz udoskonalone algorytmy mechanizmów umożliwiających programowanie równoległe.

System *PVM* staje się standardem dla programujących rozproszone równoległe programy przede wszystkim ze względu na łatwość zastosowania przez każdego użytkownika, funkcjonalność, znakomitą przenośność oraz stale rosnącą liczbę oprogramowania rozszerzającego.

Projektując system *DPE* autor wzorował się częściowo na systemie *PVM* stąd architektury obu systemów są podobne. Węzły w systemie *PVM* łączone są logicznie w pierścień lub gwiazdę tworząc maszynę wirtualną (w *DPE* struktura może być dowolna). Na każdym komputerze działa demon *pvmd* oraz dostępna jest obszerna biblioteka funkcji *C*.

System *PVM* jest już wygodnym środowiskiem rozwijanym od siedmiu lat. Autor dostrzega jednak przewagę *DPE/UNIX* w większej ogólności podstaw konstrukcji systemu:

- wybór węzła optymalnego odbywa się na podstawie bardzo ogólnych kryteriów dotyczących wymagań wobec zasobów węzła;
- nie ma ograniczeń dotyczących struktury węzłów;
- zadaniem może być dowolny binarny program i nie musi być on przystosowany do środowiska (np. przez zmodyfikowanie kodu źródłowego czy nawet rekompilację).

2.1.3 System *p4*

Środowisko *p4* umożliwia programowanie rozproszone w dwóch modelach [14]:

- pamięci dzielonej – programowanie oparte na lokalnych monitorach,
- pamięci rozproszonej – programowanie oparte na rozproszonych procesach przekazujących sobie komunikaty.

Programowanie oparte na monitorach jest wspierane przez zestaw użytecznych monitorów i procedur umożliwiających programiście tworzenie własnych monitorów.

W modelu pamięci rozproszonej, *p4* dostarcza operacje do przekazywania komunikatów i narzędzia umożliwiające tworzenie rozproszonych struktur procesów w sposób statyczny (dynamiczne tworzenie procesów może odbywać się jedynie w obrębie danego węzła). Struktura procesów ma postać hierarchii. Operacje przekazywania komunikatów mogą działać w trybie rozgłaszania (ang. *broadcast*).

W porównaniu z systemem *DPE/UNIX* środowisko *p4* udostępnia narzędzia ułatwiające zapisanie konkretnych problemów w postaci komunikujących się zadań do rozproszonego wykonania, ale musi to być zrobione jawnie, a struktura procesów jest mało elastyczna. Ciekawą cechą systemu *p4* jest to, że potrafi on badać topologię sieci w celu optymalizacji wydajności.

Rozszerzoną wersją systemu *p4* jest *Parmacs* [20].

2.1.4 System *Express*

Express to zbiór narzędzi ułatwiających tworzenie współbieżnych wersji sekwencyjnych zadań i środowisko do wykonywania ich w sposób rozproszony [14]. Narzędzia *VTOOL* i *FTOOL* umożliwiają obserwację zadania sekwencyjnego i dostarczają informacje na temat takich struktur kodu i danych, które mogą być wydzielone z tego zadania i wykonywane jako współbieżne zadania pomocnicze. Narzędzie *ASPAR* tworzy współbieżne (i gotowe do rozproszonego wykonania w środowisku *Express*) wersje algorytmów zapisanych w języku *C* lub *Fortran*.

Środowisko tworzą również biblioteki komunikacyjne i wejścia/wyjścia, w tym operacje do przesyłania komunikatów i rozpraszania danych.

2.2 Zagadnienie równoważenia obciążenia

Na szczególną uwagę zasługuje zagadnienie równoważenia obciążenia w systemie przetwarzania rozproszonego, ponieważ dobór odpowiedniego rozwiązania ma kluczowe znaczenie dla wydajności i skalowalności systemu [17].

Typowy model, w którym rozpatruje się różne metody równoważenia obciążenia, to spójna sieć autonomicznych węzłów, częściowo połączona. Z każdym węzłem związana jest kolejka zadań do wykonania i kolejka zadań wykonywanych. Celem równoważenia obciążenia jest z reguły minimalizacja średniego czasu odpowiedzi.

Najczęściej wyróżnia się trzy klasy metod równoważenia obciążenia [1, 7]:

- dynamiczne, które używają informacji o obciążeniu węzłów;
- statyczne – nie używają tych informacji, zatem wybór węzła może być deterministyczny lub losowy;

- adaptacyjne (czasem rodzaj metod dynamicznych) dobierają najlepszy w danej chwili algorytm.

Mierzenie obciążenia węzłów może być przeprowadzane na wiele sposobów. Najbardziej użyteczne to [7]:

- pomiar długości kolejki zadań dla procesora;
- pomiar wielkości wolnej pamięci;
- pomiar częstotliwości operacji zmiany kontekstu lub wywołań systemowych;
- pomiar współczynnika wykorzystania czasu procesora.

Badania dowodzą, że w tak ogólnie postawionym problemie najlepsze rezultaty uzyskuje się stosując metodę pomiaru długości kolejki zadań.

Jakąkolwiek metodę by nie zastosować, trzeba rozwiązać problem wiarygodności pomiaru, który może mieć niewielką wartość, jeśli obciążenie węzła ulega gwałtownym zmianom. Najczęściej stosowanym rozwiązaniem jest uśrednianie wyników kilku ostatnich pomiarów.

Równoważenie z wywłaszczaniem, trudne w implementacji lecz konieczne, gdy oczekuje się występowania okresów wysokiego obciążenia systemu, umożliwia przeniesienie zadania wykonującego się na pewnym węźle na inny, mniej obciążony. Równoważenie bez wywłaszczania, zwane też rozmieszczaniem zadań, jest metodą trwałego przydziału zadania węzłowi przed rozpoczęciem wykonywania.

Równoważenie zcentralizowane to takie, w którym decyzje wyboru węzła dokonywane są przez ustalony węzeł – stwarza to problem stabilności systemu, którego ten wybrany węzeł staje się bardzo słabym punktem. Spośród metod rozproszonych, najpopularniejsze są metody hierarchiczne wykorzystujące strategię “dziel i rządź”.

Węzłem inicjującym operację równoważenia może być węzeł nadmiernie obciążony (metody aktywne lub inicjowania przez nadawcę, ang. *sender-initiated algorithms*), który usiłuje przesłać zadanie na mniej obciążony węzeł. Równoważenie obciążenia może inicjować węzeł mało obciążony (metody pasywne lub inicjowania przez odbiorcę, ang. *receiver-initiated algorithms*), który poszukuje węzła, od którego mógłby przejąć zadanie. Pierwsze podejście sprawdza się lepiej w przeciętnych warunkach, zaś w sytuacjach bardzo dużego obciążenia systemu warto zdecydować się na drugie rozwiązanie, gdyż wtedy mniej obciążony węzeł poszukujący zawsze znajdzie niewielkim kosztem węzeł bardziej obciążony. Wyniki te sugerują celowość rozwiązań łączących obie strategie (metody mieszane lub symetrycznego inicjowania, ang. *symmetrically initiated algorithms*).

Decyzja o migracji zadania ma sens, jeśli koszt operacji równoważenia jest niższy niż zysk wynikły z wykonywania zadania na mniej obciążonym węźle. Koszt migracji jest znaczący w przypadku równoważenia z wywłaszczaniem, gdyż odtworzenie kontekstu zadania na innym węźle jest zazwyczaj trudne.

W powyższych rozważaniach przyjęto, że równoważenie obciążenia polega na migracji zadań, być może też danych wykorzystywanych przez te zadania. Zadaniami są zazwyczaj procesy.

Pomysł ten można również zastosować do obiektowego modelu obliczeniowego, w którym równoważenie obciążenia polegałoby na przemieszczaniu obiektów. Obiekty to struktury danych (atrybuty obiektu) i kodu (metody obiektu) powielonego na wszystkich węzłach. Zazwyczaj obiekty są tworzone dynamiczne, wszystkie posiadają niepowtarzalne nazwy (identyfikatory), komunikują się przez zdalne wykonywanie metod. Liczba obiektów bywa znacznie większa od liczby procesorów.

Wywołanie metody, zwane częściej przesłaniem komunikatu, polega na uruchomieniu procesu, który wykona kod tej metody, korzystając z zestawu danych obiektu, parametrów i ewentualnie innych danych. Komunikat nazywa się synchroniczny, jeśli nadawca oczekuje zakończenia wykonywania zdalnego procesu i ewentualnie korzysta z udostępnionej wartości wynikowej. Komunikat jest asynchroniczny, jeśli po wysłaniu komunikatu nadawca działa dalej (nie czeka na wynik).

W systemie *DPE/UNIX*, zaprojektowanym według modelu procesowego, zastosowano metodę dynamiczną, bez wywłaszczania, hierarchiczną, w której węzłem inicjującym jest węzeł bardziej obciążony. Dobór węzła odbywa się spośród zbioru węzłów spełniających kryterium wskazane przez węzeł inicjujący i deklarujących się jako węzły gotowe do przyjęcia zadania. Najlepszy węzeł to ten, dla którego określone wyrażenie (nazywane w zagadnieniach optymalizacji funkcją celu) osiąga wartość maksymalną, przy czym elementami tego wyrażenia mogą być dynamicznie wyliczane parametry węzła, takie jak wykorzystanie procesora czy wielkość wolnej pamięci. Autor uogólnił więc rozumienie

obciążenia węzła, które może być różnie liczone zależnie od wymagań zadania zlecanego przez węzeł inicjujący.

3 METODYKA TWORZENIA SYSTEMU *DPE/UNIX*

Z racji wspomnianej złożoności systemów przetwarzania rozproszonego, ich konstruowanie jest dosyć skomplikowane, a implementacja powinna być poprzedzona wykonaniem dokładnej specyfikacji projektowej. Specyfikacja musi być precyzyjna, jednak – zdaniem autora – nie należy podejmować się stworzenia jej przy pomocy formalizmu (a już na pewno nie w całości), lecz należy zastosować odpowiednią metodykę, ponieważ:

- koszty wykonania formalnej specyfikacji są wysokie; liczba szczegółów, które należy rozważyć, oraz drobnych problemów, które należy rozwiązać, jest bardzo duża – trudno je wszytkie wyrazić formalizmem, a specyfikacja byłaby bardzo obszerna;
- formalna specyfikacja jest rzeczywiście celowa w przypadku, gdy użytkownikom systemu zależy na stworzeniu systemu o gwarantowanej, logicznie dowiedzionej poprawności (np. jeśli jest to system krytyczny, w którym podstawowym wymaganiem jest bezpieczeństwo); jeśli tak nie jest (jak w przypadku naszego systemu), podejście to – przynajmniej obecnie – jest niepotrzebne.

Na potrzeby systemu *DPE/UNIX* autor opracował metodykę, która może być wykorzystana do tworzenia innych, w szczególności mniej ogólnych systemów przetwarzania rozproszonego.

W niniejszej pracy słowa metodologia i metodyka używane są zamiennie i w znaczeniu w jakim używa się ich zwykle w konstrukcji systemów bazodanowych.

3.1 O inżynierii systemów

Zanim powstały metodologie, stosowano już różne techniki modelowania systemów, np. systemów baz danych, jak modelowanie przepływu danych (lub procesów, ang. *Data Flow Model*) i modelowanie encji (ang. *Entity-Relationship Model*). Zasady użycia tych technik usystematyzowano np. w metodologiach *SSADM*, *Merise* i *Euromethod* [5].

Metodologia to precyzyjny przepis określający sposób tworzenia oprogramowania. Może on ograniczać się do wskazówek i naszkicowania wizji procesu tworzenia systemu, albo dokładnie przedstawiać ten proces: krok po kroku, dzień po dniu, procedura po procedurze. Podejście metodologiczne zakłada więc, że projektowanie systemów informatycznych to proces formalny i jako taki stanowi przedmiot studiów i udoskonaleń.

Skoro przy tworzeniu systemu informatycznego można stosować podejście inżynierskie, narzuca się myśl o zastosowaniu narzędzi informatycznych, które wspomagałyby takie postępowanie – takie narzędzia to oczywiście systemy *CASE*.

Metodologia jest opisem procesu tworzenia systemów informatycznych. W przypadku konkretnego projektu dostarcza ona struktury zadań do wykonania i produktów, które muszą zostać stworzone, by można było realizować kolejne zadania. Ale nie mówi nic o terminach i ludziach, wyjąwszy to, że zadanie *X* ma zostać wykonane przez osobę zdolną do robienia *A* zanim osoba o umiejętności *B* rozpocznie pracę nad zadaniem *Y*. Z reguły nie mówi nic o ocenie jakości.

W ogólności kontrola jakości obejmuje nie tylko ocenę produktów jako wyników realizacji zadań, ale również ocenę procesów produkcyjnych w celu ich doskonalenia. Myślenie o doskonaleniu metodologii wywodzi się z współczesnej kultury zarządzania i takich jej elementów jak totalne zarządzanie jakością (ang. *Total Quality Management*, *TQM*) i rein¿ynieria (ang. *Business Process Reengineering*) [15].

Jednym z podstawowych założeń zarządzania jakością jest to, że błędy nie występują na końcu procesu produkcyjnego. Dlatego też zgodne z nim skuteczne zarządzanie projektem wymaga przygotowania dwóch planów: planu projektu (który winien uwzględniać ludzi i przedstawiać terminy) oraz planu kontroli jakości. W szczególności z testowaniem powstającego systemu nie należy czekać do chwili jego powstania.

Popularną klasyfikacją metodyk jest podział na zstępujące (ang. *top-down*) i wstępujące (ang. *bottom-up*). W metodologiach zstępujących, zwanych czasem klasycznymi czy wodospadowymi (ang. *waterfall*), konstrukcja systemu przebiega według ściśle określonych etapów, a w każdym z nich powstaje pewna liczba produktów docelowych (ang. *external deliverables*) lub pośrednich, potrzebnych

do realizacji kolejnych faz (ang. *internal deliverables*) – zaletą jest możliwość pełnej kontroli nad przebiegiem prac, wadą: mała elastyczność – produkty docelowe powstają późno, więc istnieje duże ryzyko niezgodności z oczekiwaniami użytkowników. System tworzony zgodnie z czystą metodologią wstępującą powstaje przez prototypowanie – poszczególne jego części powstają równolegle. Tego typu metodyki stosuje się, gdy wymagania użytkownika trudno jest zdefiniować; zaletą jest duża elastyczność, wadą – słaba kontrola [6].

Popularne współczesne metodologie (jak *Client/Server*, *Rapid Application Development,* czy metodyki korporacyjne jak np. *Software Life Cycle* firmy *EDS*) oparte są na podejściu wyważonym (analiza, iteracyjnie przebiegająca konstrukcja, wdrożenie).

3.2 Szkic metodyki

Pomysł na metodykę jest następujący: należy określić moduły systemu i schematy ich wzajemnej komunikacji, następnie zaimplementować moduły jako procesy-demony, zaś schematy komunikacji jako protokoły komunikacyjne; dodatkowo należy zaprojektować i zrealizować biblioteki i narzędzia umożliwiające korzystanie z systemu.

Konstrukcja systemu składa się z sześciu etapów:

1. Analiza wymagań dla systemu obejmująca zdefiniowanie katalogu użytkowników, stworzenie katalogu wymagań jakościowych i ilościowych.
2. Specyfikacja logiczna systemu: zdefiniowanie logicznych modułów, schematów komunikacji, narzędzi i opracowanie algorytmów.
3. Projekt fizyczny systemu: specyfikacja demonów, protokołów, programów narzędziowych i bibliotek; także specyfikacja testów tych komponentów.
4. Implementacja systemu, czyli realizacja i opis realizacji demonów, protokołów, programów narzędziowych i bibliotek.
5. Testy wszystkich składników systemu oraz testy integralności.
6. Przegląd i korekta systemu.

W konstrukcji systemu stosuje się podejście zstępujące: każdy z produktów wyjściowych określa jakie produkty są potrzebne do jego urzeczywistnienia w systemie docelowym, np. realizacja pewnego modułu systemu może wymagać implementacji określonego demona, protokołu(ów) oraz bibliotek. (Specyfikacja systemu staje się strukturą definicji składników, powiązaną polem **Pochodzenie**).

Dokładny opis metodyki znajduje się w dodatku *Definicja metodyki*.

3.3 Wybór narzędzi

Ogromnym ułatwieniem dla projektującego system komputerowy jest zastosowanie odpowiedniego narzędzia *CASE*, które nie tylko przechowuje całą dokumentację projektową (jako tzw. repozytorium), ale również ułatwia rozwijanie systemu w sposób zgodny z przyjętą metodyką. Autor rozważał możliwość stworzenia prostego narzędzia, ale zdecydował się na zastosowanie edytora *Microsoft Word*, który okazał się bardzo wygodny dzięki łatwości operowania na dokumentach o złożonej hierarchicznej strukturze – należało przede wszystkim odwzorować strukturę zadań opracowanej metodyki na strukturę dokumentu.

4 SZKIC DEFINICJI SYSTEMU *DPE/UNIX*

4.1 Ogólne wymagania względem systemu

System ma działać jako środowisko umożliwiające głównie programom aplikacyjnym, choć nie koniecznie tylko takim, działanie w sposób rozproszony. Jako że ma tworzyć warstwę pomiędzy kodem systemu operacyjnego a kodem aplikacji, należy oczekiwać od niego tych cech, których oczekuje się od usług systemu operacyjnego: umożliwić wygodne i wydajne tworzenie oraz wykonywanie oprogramowania [16].

Dobrze zaprojektowany system realizuje priorytety. Takim priorytetowym wymaganiem w przypadku systemu *DPE* jest maksymalizacja siły przetwarzania systemu (jak się ją określa, opisano dalej).

Wymagania dotyczące systemów operacyjnych mogą dotyczyć mechanizmów, które powinny być w systemach zaimplementowane. Wymagania związane z niezależnością od sprzętu mogą dotyczyć transformacji kodu lub formatu danych. Oczekuje się możliwie mało kodu zależnego od platformy, nie ingerowania w jądro systemu operacyjnego – w tym w mechanizmy sieciowe. Spełnianie wszystkich tych wymagań zapewni przenośność systemu.

Przetwarzanie w systemie rozproszonym *DPE/UNIX* ma przebiegać na autonomicznych, w pełni równuoprawnionych węzłach. Nie wprowadza się żadnych struktur węzłów. Każdy węzeł zna swoich „sąsiadów" i tylko te węzły (nie powinien przechowywać listy wszystkich węzłów w systemie, ponieważ, po pierwsze lista mogłaby być bardzo długa, po drugie aktualizacja wszystkich list w systemie byłaby kosztowna, a po trzecie nie ma takiej potrzeby – kosztem dodatkowej komunikacji można dostać się do każdego węzła systemu). Są to wybrane węzły, z którymi może się on bezpośrednio komunikować np. w celu wybrania węzła optymalnego do wykonania danej usługi. W sieci komunikacyjnej może więc działać wiele niezależnych systemów *DPE*, które nie są w ten sposób połączone, a każdy zbiór węzłów powiązanych wspomnianą relacją sąsiedztwa wyznacza pewien graf (nazwiemy go grafem systemu).

Nie należy ograniczać liczby węzłów. Każdy węzeł może udostępniać innym zasoby (usługi i pamięć na ich wykonanie, pliki i miejsce na dysku na repliki danych), ale kontrola nad zasobami jest zawsze lokalna. Zarządzanie usługami obejmuje rejestrowanie i wyrejestrowywanie usług, wykonywanie usług, przerywanie ich pracy oraz konfigurowanie (np. w celu uzyskania większej wydajności, czy ustalanie domyślnych parametrów). Wykonywanie i przerywanie pracy (niszczenie) usług można przeprowadzać zdalnie. Nie nakłada się żadnych ograniczeń na dostępność usług, tzn. każdy użytkownik systemu może zlecić wykonanie dowolnej usługi, której proces może być uruchomiony potencjalnie na każdym węźle (można zaimplementować takie ograniczenia przy pomocy mechanizmu dojść).

Zarządzanie pamięcią odbywa się lokalnie – element globalnego zarządzania pamięcią pojawia się przy wyborze węzła optymalnego do wykonania określonej usługi i przy migracji danych, kiedy to w systemie plików węzła docelowego umieszcza się replikę pliku z węzła, z którego on pochodzi [2].

Przezroczystość pewnej funkcji systemu oznacza, że jej realizacja jest niewidoczna dla jej użytkownika, który jest nieświadomy pewnych jej aspektów ukrywanych przez system. Ogólnie w systemie rozproszonym możemy mieć do czynienia z przezroczystością dostępu, umiejscowienia, współbieżności, replikacji, błędów, migracji, wydajności. Oczekuje się zapewniania kolejności komunikatów, wykrywania i radzenia sobie ze zduplikowanymi komunikatami oraz gwarancji dostarczania komunikatów. Dane w systemie rozproszonym są dzielone i mogą być replikowane, gdy potrzeba (zwykle realizuje się tą cechę w postaci rozproszonego systemu plików). Aby zapewnić łatwą rozszerzalność systemu można zaimplementować go tak, by na nowym węźle wystarczyło lokalnie uruchomić odpowiednie moduły systemu.

Chcielibyśmy – bez ingerencji w jądro systemu operacyjnego – dać możliwość korzystania z rozproszonego systemu plików takim procesom, jak np. standardowe polecenia systemowe *wc*, czy *grep* – w ogólności takim, które przetwarzają plik, bo mają jego deskryptor, ale nic nie wiedzą o jego nazwie czy lokalizacji (w systemie *DPE* plik ten może być zdalny, tj. znajdować się na innym węźle). Przy dostępie do plików zdalnych wygodnie jest przyjąć zasadę 'wzajemnego zaufania' (ang. *trusted security*) – zdalny system przyjmuje, że lokalny zweryfikował użytkownika pozytywnie [19].

Synchronizacja powinna odbywać się przez mechanizmy udostępniane przez platformę, np. sygnały. Potrzebny jest mechanizm umożliwiający taką komunikację przez sieć. Wydajność systemu komunikacji jest ważna – od jego sprawności zależy wydajność całego systemu.

Ponieważ jedną z przyczyn nieudostępniania danych lokalnych mogą być względy bezpieczeństwa, dane przesyłane przez sieć przy zdalnym wykonywaniu powinny być szyfrowane. W implementacji pominięto to wymaganie.

W systemach rozproszonych przywiązuje się dużą wagę to niezawodności, w tym do unikania zastoju oraz wykrywania i podnoszenia się z błędów (np. przez reelekcja serwera i ponowne rozpoczęcie przetwarzania) [3, 9]. W systemie *DPE* pominięto to zagadnienia ze względu na ich złożoność, a stąd trudność implementacji.

4.2 Omówienie platform

System *UNIX* był systemem pionierskim w wielu rozwiązaniach. Posiada on unikatowe zestawienie takich cech jak: przenośność, dobre mechanizmy komunikacji, bogate oprogramowanie narzędzi systemowych, modularność. Warto przypomnieć, że to twórcy *UNIXa* wprowadzili pomysł współpracy wielu procesów w celu realizacji wspólnego celu [18].

W celu uzyskania możliwie dużej przenośności, w systemie *DPE* nie używa się rzadko spotykanych funkcji systemowych i bibliotecznych, definicji danych, opcji, sygnałów. Nie wykorzystuje się *RPC*, ale tylko gniazda *BSD* (wywołania *RPC* są czasochłonne i niedostępne w niektórych wersjach *UNIX*a).

Dla potrzeb *DPE* nie stworzono systemu nazw, wykorzystuje się *DNS*.

Przekazywanie komunikatów zlecono oprogramowaniu sieciowemu *TCP*. Nie korzysta się z protokołu *UDP* – ma on większą przepustowość niż *TCP*, ale w *DPE* niewiele danych jest przesyłanych. Komunikaty *TCP* są dostarczane w kolejności wysłania, połączenie jest ustalane raz i nie trzeba już adresować komunikatów [13].

Dopuszczenie heterogeniczności sprzętu (różne procesory, różne reprezentacje danych) umożliwia zwiększenie wydajności dzięki potencjalnie większemu rozproszoniu oraz rozszerza zakres usług dostępnych z wybranej stacji na takie, które wykonywane mogą być tylko na komputerach o innej architekturze. Heterogeniczność sieci (różne media transmisji, różne protokoły na poziomie łącza danych, różne interfejsy sprzętowe) nie stanowi problemu, ponieważ wykorzystywane są protokoły *TCP* i *IP.* Heterogeniczność systemów operacyjnych i sprzętu wprowadza konieczność wyodrębnienia małej biblioteki funkcji zależnych od systemu operacyjnego i sprzętu.

4.3 Usługi

O każdej usłudze udostępnianej przez dany serwer zapamiętane są następujące informacje:

- nazwa usługi;
- pełna lokalna ścieżka dostępu do kodu binarnego usługi;
- domyślne parametry wywołania usługi;
- wartość używana do skorygowania priorytetu procesu usługi (ang. *nice value*);
- tzw. warunek lokalny (tj. taki, który musi spełniać lokalny węzeł, by usługa mogła być na nim wykonana z innego węzła);
- tzw. warunek globalny (tj. taki, który musi spełniać każdy z węzłów branych pod uwagę przy wyborze najlepszego węzła przeznaczonego do wykonania danej usługi);
- wymagania wobec zasobów węzła.

Dodanie usługi polega na:

- skompilowaniu kodu programu usługowego na danym węźle;
- zapisaniu wyżej wymienionych informacji na temat usługi w bazie danych na danym węźle.

Gdy usługa wykonywana jest na węźle zdalnym (tzn. różnym od tego, z którego pochodzi jej żądanie wykonania), dane wejściowe i wyjściowe przekazywane są przez sieć protokołem *TCP*. Inne dane, do których usługi mogą mieć dostęp, to oczywiście pliki. Ponieważ każda usługa może być wykonana potencjalnie na dowolnym, często nieznanym zlecającemu, węźle w sieci *DPE*, należy zaimplementować mechanizm dostępu do zdalnych plików. Dane wielokrotnie czytane lub modyfikowane warto skopiować. Jeśli miałyby być mało czytane, to nie warto, jednak nie da się tego przewidzieć – chyba, że proces jawnie zadeklaruje, że w ten sposób chce użyć pliku. W systemie *DPE* nie zaimplementowano tego mechanizmu.

4.4 Architektura systemu

Modularność

System *DPE* ma architekturę modularną. Moduły systemu są implementowane w postaci *UNIX*owych procesów-demonów. Są to:

- *dpeinfod* – demon podający informacje o obciążeniu węzła;
- *dpefsysd* – demon rozproszonego systemu plików;
- *dpesvcsd* – demon rozproszonej bazy usług;
- *dpeseld* – demon wybierający węzeł optymalny;
- *dpeexecd* – demon zdanego wykonywania usług.

Ostatni z modułów został zaimplementowany jako program *dpestub* – inicjuje on wykonanie zdalnej usługi.

System ma również architekturę wielowarstwową. Użytkownik, który zleca wykonanie usługi zwraca się do najwyższej warstwy jaką jest interfejs modułu *dpestub*. Moduł ten komunikuję się z drugą warstwą: kieruje żądanie znalezienia najlepszego węzła do *dpeseld*, następnie zleca wykonanie usługi *dpeexecd*. Na najniższym poziomie, najbliższym systemowi operacyjnemu, działają moduły: *dpeinfod* i *dpesvcsd* (umożliwiają wybranie optymalnego węzła) oraz *dpefsysd* (wraz z *dpesvcsd* umożliwia wykonanie usługi).

Podstawową korzyścią modularności jest abstrakcja kodu – każdy moduł można wymienić nie zmieniając interfejsu. Z demonem można komunikować się poprzez gniazdo związane z ustalonym portem – wykorzystuje się protokół *TCP* (prostszy niż np. *RPC*). Gniazdo wraz z protokołem komunikacji przez nie jest zatem interfejsem demona. Każdy z demonów ma zarezerwowany dla siebie jeden port komunikacyjny. Każdy z demonów może jednocześnie obsługiwać wiele procesów-klientów (serwer wielowątkowy, ang. *multithreaded server*) [13].

Równoważenie obciążenia

Przydział procesora (ang. *load sharing*), czyli wybór węzła optymalnego odbywa się poprzez zorganizowanie węzłów w hierarchię (wyznaczenie drzewa rozpinającego graf systemu) oraz wybór w każdym węźle (zatem korzeniu poddrzewa) najlepszego węzła w poddrzewie. Algorytm ma własność stopu, ponieważ każdy węzeł odwiedzany jest tylko raz przy dystrybucji zapytania (osiągnięto to w ten sposób, że w węzłach zapamiętywana jest informacja identyfikująca zapytanie).

Informacja dotycząca zasobów węzła zapamiętana jest w danym węźle i uaktualniana co pewien czas – każdy sąsiad może ją w dowolnej chwili uzyskać.

Analiza złożoności algorytmu odnajdywania węzła optymalnego przedstawia się następująco (e – liczba krawędzi, czyli połączeń):

1. Dystrybucja zapytania to $(e - c) + 2c = e + c$ komunikatów przesłanych do innych węzłów, przy czym c to liczba krawędzi poprzecznych – jest tak dlatego, że z każdą krawędzią poprzeczną wiąże się nieudana próba dystrybucji zapytania wgłąb (nieudana, ponieważ docelowy wierzchołek już wchodzi w skład drzewa rozpinającego o $e - c$ krawędziach). Zatem złożoność wynosi $O(e)$.
2. Złożoność zebrania wyników to liczba krawędzi w drzewie rozpinającym.

Zatem algorytm ma złożoność liniową $O(e)$ względem liczby połączeń e.

Moduły systemu

Zadaniem modułu informującego o stanie węzła jest zbieranie i udostępnianie informacji o systemie pracującym na danym węźle i o aktualnym stanie tego węzła, w tym o jego obciążeniu.

Model komunikacji to klient-serwer, z tym że na każdym komputerze pracują moduły realizujące funkcjonalność serwera i procesy z każdego komputera mogą być klientami – w tym sensie mamy więc do czynienia z modelem zintegrowanym, ponieważ na każdym komputerze działa (przynajmniej potencjalnie) to samo oprogramowanie. Problemy z modelem klient-serwer są następujące [15]:

- podatność na błędy serwera (mniejsza, gdy wiele serwerów, w niektórych przypadkach awarii serwera przetwarzanie może być wykonywane od początku na innym serwerze – reelekcja serwera, itd.);
- serwer wąskim gardłem (niekoniecznie, bo z reguły wiele serwerów);
- problem większego kosztu większej liczby serwerów, które robią to samo.

Pierwszy problem w systemie *DPE* nie został rozwiązany, tzn. przerwanie pracy powoduje zerwanie komunikacji. Serwery nie są wąskim gardłem, ponieważ są wielowątkowe. Większy koszt większej liczby serwerów nie ma znaczenia, po za tym liczność serwerów to wymaganie systemu rozproszonego.

Wydajność całego systemu zależy od wydajności systemu komunikacji. Zaimplementowane schematy komunikacji funkcjonują na poziomie warstwy aplikacyjnej (obejmuje ona m.in. schematy uzgadniania korzystania z globalnych zasobów systemu) w modelu *ISO/OSI*.

Zastosowano koncepcję pamięci podręcznej (ang. *remote cache*): lokalne kopie zdalnych plików (repliki).

Realizacja demona rozproszonego systemu plików wygląda w skrócie następująco: demon przechowuje listę lokalnych replik i obsługuje dostęp do tych plików. Pliki, do których sięga się z innego węzła, są w całości kopiowane (replikowane) na ten węzeł i umieszczane w katalogu */tmp*. Po zakończeniu wszystkich operacji modyfikacji takiego pliku jest on zwracany węzłowi macierzystemu, jeśli faktycznie został zmieniony. Podobna metoda została zaimplementowana w systemie *Andrew*.

Każdy program, który zdobędzie deskryptor pliku otwartego zdalnie, może na nim operować – nie wiedząc, że w rzeczywistości operuje na pliku z odległego systemu.

Przyjęte rozwiązanie jest dosyć proste w implementacji, w szczególności nie wymaga ingerencji w jądro systemu na żadnym węźle.

Nie przestrzeganie powyższych uwag może prowadzić do następujących problemów:

- nawet jeśli powstały na węźle *N2* (gdzie jest replika) tylko niewielkie zmiany w bardzo dużym pliku, to odsyła się na węzeł *N1* cały plik;
- ponieważ zamknięcie pliku *F2* (repliki) oznacza odesłanie jego treści na węzeł *N1*, możemy utracić zmiany dokonane po tej chwili przez procesy, które nie wiedzą o tym, że nie operują na zwykłym pliku, lecz na replice (np. program uruchomiony po funkcji *fork*) i zwykłe zamknięcie pliku nie wystarczy; w tym sensie semantyka operowania na plikach zdalnych różni się od semantyki operowania na plikach lokalnych (zgodnie z wymaganiami – patrz: *Remote-Files-Access-Transparency*).

Pierwszego problemu można by uniknąć w sposób następujący: niech replika (*F2*) zawiera tylko te dane, które są niezbędne i niech będą one przesyłane w miarę potrzeby (tak jak przy stronicowaniu na żądanie) – taki zresztą algorytm zaimplementowano w systemie *NFS*. Wówczas jednak tracimy którąś z zalet rozwiązania w *DPE*, gdyż albo

- umożliwiamy swobodny dostęp wszystkim programom i wtedy musimy zmodyfikować w jądrze obsługę operacji *read*, *write*, *seek* (należy wykluczyć możliwość scalenia wszystkich programów w systemie z kodem funkcji realizujących te ‘nowe’ – przystosowane do rozproszenia – odpowiedniki operacji *read*, *write*, *seek*), albo
- rezygnujemy z takiej możliwości.

W przypadku drugiego problemu podkreślmy, że naruszenie semantyki dotyczy wyłącznie procesów nieświadomych istnienia rozproszonego systemu plików, ponadto można to ograniczenie łatwo obejść przez odpowiednie zaprogramowanie procesu-ojca (mającego świadomość przetwarzania zdalnego), który powinien oczekiwać na zakończenie działania swoich potomków (tych nie mających tej świadomości, jak *grep*) i dopiero wtedy zamknąć plik-replikę.

W związku z tworzeniem replik powstaje problem z semantyką operacji czytania i pisania. Weźmy bowiem taką sytuację: dana *D* o nazwie *D1* na serwerze *S1* zostaje zreplikowana na *server S2*, gdzie ma nazwę *D2*. Proces działający na *S1* modyfikuje daną *D1* i otrzymuje ona wartość *D1'*, podobnie proces działający na *S2* modyfikuje *D2*, gdzie ma ona teraz wartość *D2'* (niech *D1'* różne od *D2*'). Pytanie: jaką wartość (D1' czy D2') powinien przeczytać proces na serwerze *S3*, który chce operować na danej *D*? Powstaje konflikt. Należy go rozwiązać przez uzgodnienie wartości danej *D*.

W systemie *DPE/UNIX* nie rozwiązano tego problemu (zatem ostatni proces zamykający plik zapisuje zmiany, określające nową treść pliku). Warto jednak przyjrzeć się temu problemowi, ponieważ występuję często w systemach rozproszonych, np. w replikacyjnych bazach danych (*Oracle*, *Sybase SQL Anywhere*). Nie znaleziono dla niego dotąd ogólnego i w pełni zadowalającego rozwiązania. Dwie metody rozwiązywania konfliktów są najbardziej popularne:

- metoda symetryczna asynchroniczna – wartości danej *D* są uzgadniane w chwili, gdy jakiś nowy proces chce uzyskać aktualną wartość *D*;
- metoda symetryczna synchroniczna – wartości danej *D* są uzgadniane okresowo, co jakiś czas.

Uzgadnianie danych odbywa się zazwyczaj na jeden z następujących sposobów:

- metoda stempli czasowych (ang. *time stamps*) – za obowiązującą wartość uznaje się tę, która została zapisana najpóźniej (powstaje potrzeba wprowadzenia globalnego zegara);
- metoda priorytetowa – za obowiązującą uznaje się wartość pochodzącą od serwera o najwyższym priorytecie;
- metoda funkcji wartości – za obowiązującą uznaje się wartość będącą pewną funkcją wartości replik, np. *min*, *max*, suma, konkatenacja, czy średnia (funkcję należy dobrać stosownie do interpretacji danej, którą to interpretację trzeba znać).

Komunikacja między demonami odbywa się dzięki ustaleniu protokołów komunikacyjnych. Protokół to zbiór reguł i konwencji dotyczących formatu i synchronizacji w czasie wymiany komunikatów między dwoma lub więcej procesami. Wprowadzanie protokołów komunikacyjnych wynika z potrzeby standaryzacji w systemach komunikacyjnych. Architektura sieci z punktu widzenia pakietu danych przesyłanych między komunikującymi się procesami ma strukturę warstwową (*ISO/OSI*), bo daje to podstawę koordynacji rozwoju standardów komunikacji między systemami, dokładnie określa dziedziny wymagające stworzenia standardów, umożliwia zachowanie zgodności między wszystkimi standardami, umożliwia łączenie systemów różnych producentów. W systemie *DPE* istnieją protokoły warstwy aplikacyjnej, a wykorzystane zostaną protokoły rodziny *TCP/IP* w niższych warstwach. Dla każdego implementowanego protokołu określono funkcję, podstawowy scenariusz (definicja podstawowych komunikatów i ich poprawnych podstawowych sekwencji) i procedury wychodzenia z błędów (defincja dodatkowych komunikatów i ich poprawnych sekwencji).

Miejsce kodu *DPE* w systemie rozproszonym to warstwa pomiędzy jądrem a aplikacją (zatem nie ingeruje się w jądro SO, a korzysta się z mechanizmów sieciowych).

5 ANALIZA WYMAGAŃ WZGLĘDEM SYSTEMU *DPE/UNIX*

Celem analizy wymagań jest określenie i udokumentowanie wymagań względem tworzonego systemu.

Analiza wymagań prowadzi do uzyskania katalogu użytkowników systemu i katalogu wymagań pochodzących od tych użytkowników. Niektóre wymagania trudno jednoznacznie sklasyfikować ze względu na dziedzinę – wtedy zwracamy uwagę na najistotniejszą. Etap analizy wymagań pełni w wielu metodologiach projektowania systemów bazodanowych kluczową rolę, ponieważ umożliwia określenie zakresu i funkcjonalności systemu – podobnie jest w przypadku systemu *DPE*.

Dziedziny wymagań mogą być następujące:

- niezależność od systemu operacyjnego (pierwszy aspekt przenośności);
- niezależność od sprzętu (drugi aspekt przenośności);
- autonomia przetwarzania;
- symetria przetwarzania;
- spójność fizyczna systemu;
- funkcjonalność systemu;
- niezawodność systemu;
- przezroczystość dostępu;
- przezroczystość replikacji.

Generalnie wymagania dzielimy na jakościowe (ogólne) i ilościowe (często stanowią sprecyzowanie wymagania jakościowego, ale niekoniecznie). Podział ten jest zbliżony do podziału wymagań na funkcyjne oraz niefunkcyjne, jaki ma miejsce w metodologii *SSADM* - odpowiednikiem analizy wymagań w *SSADM Version 4* są 2 etapy: Analiza Wymagań (ang. *Requirements Analysis Module*) oraz Specyfikacja Wymagań (ang. *Requirements Specification Module*) [5].

Każdemu wymaganiu przypisano priorytet. Priorytety mają zasadniecze znaczenie przy metodologicznej konstrukcji systemu, kiedy to często rezygnuje się z mniej ważnych wymagań np. ze względu na nadchodzący termin realizacji projektu.

Struktura analizy wymagań

Kroki		**Produkty wejściowe**	**Produkty wynikowe**
Analiza wymagań			Katalog użytkowników, Katalog wymagań
1	Określ użytkowników systemu.		Katalog użytkowników
2	Zdefiniuj wymagania jakościowe dotyczące niezależności od platformy, własności elementów przetwarzających, przezroczystości, bezpieczeństwa i ochrony, sytuacji awaryjnych oraz związane z użytecznością.	Katalog użytkowników	Katalog wymagań jakościowych

Struktura produktów analizy wymagań

Produkty

Katalog użytkowników

Katalog wymagań

- Katalog wymagań jakościowych
  - Katalog wymagań dotyczących niezależności od platformy
  - Katalog wymagań dotyczących własności elementów przetwarzających
  - Katalog wymagań dotyczących przezroczystości
  - Katalog wymagań dotyczących bezpieczeństwa i ochrony
  - Katalog wymagań związanych z użytecznością

5.1 Użytkownicy systemu

Przez użytkowników systemu rozumiemy osoby, które w jakikolwiek sposób korzystają z działającego systemu.

Katalog użytkowników

Użytkownicy systemu zostali opisani w następującym katalogu:

ID: *Superuser*

Nazwa: Administrator

Uprawnienia: Wszystkie

Rola użytkownika: Administrowanie systemem.

Uwagi: Użytkownik powinien być tożsamy z administratorem (ang. *superuser*) danego węzła, dla uproszczenia budowy systemu i zarządzania nim.

ID: *Engineer*

Nazwa: Inżynier

Uprawnienia: Pytanie o obciążenie, pytanie o udostępniane usługi, pytanie o węzeł optymalny, strojenie systemu, zdalne operowanie na własnych danych, zdalne wykonywanie usług

Rola użytkownika: Odpowiedzialny za wydajne działanie systemu.

Uwagi: Z punktu widzenia lokalnego systemu operacyjnego powinien to być użytkownik pracujący na zwykłych prawach.

ID: *User*

Nazwa: Zwykły użytkownik

Uprawnienia: Pytanie o obciążenie, pytanie o udostępniane usługi, pytanie o węzeł optymalny, zdalne operowanie na własnych danych, zdalne wykonywanie usług

Rola użytkownika: Klient usług.

Uwagi: Z punktu widzenia lokalnego systemu operacyjnego powinien to być użytkownik pracujący na zwykłych prawach.

Uwaga: oczywiście dowolna osoba może w systemie pełnić rolę wielu użytkowników (w powyższym sensie), np. administrator systemu może spełniać faktycznie funkcje administracyjne jako *Superuser*, ale w wielu sytuacjach może być traktowany jako zwykły użytkownik.

5.2 Wymagania jakościowe

Wymagania jakościowe są wymaganiami ogólnymi.

5.2.1 Wymagania dotyczące niezależności od platformy

Katalog wymagań dotyczących niezależności od platformy

Następujące wymagania dotyczą niezależności od platformy sprzętowej i systemowej:

ID: *Operating-And-Network-System-Properties*

Nazwa: Oprogramowanie węzłów i sieci

Pochodzenie: *Superuser, Engineer, User*

Dziedziny: Niezależność od systemu operacyjnego

Typ: Jakościowe

Priorytet: Bardzo wysoki

Treść wymagania: System ma pracować na węzłach, na których działa system operacyjny *UNIX* z zaimplementowanymi mechanizmami pracy w sieci *TCP/IP*. Oprogramowanie systemu *DPE* pracujące na każdym węźle powinno móc być rozpoznane przez inne węzły.

Uwagi: Wymaga się, aby w systemie operacyjnym węzłów zawarta była obsługa gniazd, nie oczekuje się obsługi *RPC*. Każdy węzeł powinien na żądanie udzielać informacji na temat swojego oprogramowania (SO i komunikacyjne).

ID: *Hardware-Properties*

Nazwa: Wymagania dotyczące sprzętu

Pochodzenie: *Superuser, Engineer, User*

Dziedziny: Niezależność od sprzętu

Typ: Jakościowe

Priorytet: Bardzo wysoki

Treść wymagania: Architektura sprzętowa każdego węzła może być dowolna byle była nowoczesna.

Uwagi: Każdy węzeł powinien na żądanie udzielać informacji na temat swojej architektury.

ID: *Hardware-Dependent-Code-Minimized*

Nazwa: Minimalizacja kodu zależnego od sprzętu

Pochodzenie: *Superuser, User*

Dziedziny: Niezależność od sprzętu

Typ: Jakościowe

Priorytet: Wysoki

Treść wymagania: System nie powinien zawierać kodu zależnego od sprzętu, jeśli funkcję realizowaną przez ten kod można zaimplementować w sposób niezależny od sprzętu.

ID: *Operating-Systems-Untouched*

Nazwa: Nietykalność kodu systemów operacyjnych

Pochodzenie: *Superuser, User*

Dziedziny: Niezależność od systemu operacyjnego

Typ: Jakościowe

Priorytet: Bardzo wysoki

Treść wymagania: Nie należy modyfikować kodu należącego do systemów operacyjnych węzłów.

5.2.2 Wymagania dotyczące własności elementów przetwarzających

Katalog wymagań dotyczących własności elementów przetwarzających

Następujące wymagania dotyczą własności węzłów jako elementów przetwarzających w systemie:

ID: *Autonomy*

Nazwa: Autonomia węzłów

Pochodzenie: *Superuser, Engineer, User*

Dziedzina: Autonomia przetwarzania

Typ: Jakościowe

Priorytet: Bardzo wysoki

Treść wymagania: Każdy węzeł stanowi samodzielny element przetwarzający w systemie i może funkcjonować niezależnie od innych węzłów.

ID: *Symmetry*

Nazwa: Symetria węzłów

Pochodzenie: *Superuser, User*

Dziedzina: Symetria przetwarzania

Typ: Jakościowe

Priorytet: Bardzo wysoki

Treść wymagania: Każdy węzeł jest jednakowo uprzywilejowany w systemie.

Uwagi: W szczególności każdy może być serwerem i klientem. Ponadto węzłom nigdy nie należy przypisywać priorytetów *ad hoc*.

ID: *Consistency*

Nazwa: Spójność fizyczna systemu

Pochodzenie: *Superuser, User*

Dziedzina: Spójność fizyczna systemu

Typ: Jakościowe

Priorytet: Bardzo wysoki

Treść wymagania: Każdy węzeł potrafi w dowolnej chwili skomunikować się z pewną liczbą innych węzłów – w szczególności zna on nazwy tych węzłów należących do sieci *DPE.*

Uwagi: Wymaganie stosuje się o ile nie wystąpiła awaria na którymś z tych węzłów.

ID: *Server-Liabilities*

Nazwa: Powinności serwera

Pochodzenie: *Superuser, Engineer, User*

Dziedzina: Funkcjonalność systemu

Typ: Jakościowe

Priorytet: Wysoki

Treść wymagania: Węzeł udostępniający pewną usługę powinien umożliwiać zdalne uruchamianie jej i przerywanie jej wykonywania.

ID: *Client-Liabilities*

Nazwa: Powinności klienta

Pochodzenie: *Superuser, Engineer, User*

Dziedzina: Funkcjonalność systemu

Typ: Jakościowe

Priorytet: Wysoki

Treść wymagania: Klient zdalnej usługi powinien dawać możliwość dostępu do danych dla usługi z węzła, na którym wykonywana jest usługa.

5.2.3 Wymagania dotyczące przezroczystości

Katalog wymagań dotyczących przezroczystości

ID: *Execution-Placement-Transparency*

Nazwa: Przezroczystość umiejscowienia wykonania

Pochodzenie: *Superuser, Engineer, User*

Dziedzina: Przezroczystość dostępu

Typ: Jakościowe

Priorytet: Bardzo wysoki

Treść wymagania: Klient usługi nie musi określać miejsca wykonania danej usługi.

ID: *Local-Files-Access-Transparency*

Nazwa: Przezroczystość dostępu do danych lokalnych

Pochodzenie: *Superuser, Engineer, User*

Dziedziny: Przezroczystość dostępu, Przezroczystość replikacji

Typ: Jakościowe

Priorytet: Wysoki

Treść wymagania: Przewarzanie udostępnianego pliku lokalnego powinno przebiegać, z punktu widzenia procesu, jak przetwarzanie zwykłego pliku lokalnego – dotyczy to także procesów, które nie wiedzą o istnieniu rozproszonego systemu plików.

Uwagi: Nie wymaga się, aby operacje wykonywane na lokalnym pliku przez procesy nie wiedzące o istnieniu rozproszonego systemu plików gwarantowały spójność treści pliku z perspektywy różnych miejsc w systemie rozproszonym.

ID: *Remote-Files-Access-Transparency*

Nazwa: Przezroczystość dostępu do danych zdalnych

Pochodzenie: *Superuser, Engineer, User*

Dziedziny: Przezroczystość dostępu, Przezroczystość replikacji

Typ: Jakościowe

Priorytet: Wysoki

Treść wymagania: Przetwarzanie pliku zdalnego powinno przebiegać, z punktu widzenia procesu, jak przetwarzanie pliku lokalnego – dotyczy to także procesów, które nie wiedzą o istnieniu rozproszonego systemu plików.

Uwagi: Nie wymaga się, aby operacje wykonywane na odległym pliku przez procesy nie wiedzące o istnieniu rozproszonego systemu plików gwarantowały spójność treści pliku z perspektywy różnych miejsc w systemie rozproszonym.

5.2.4 Wymagania dotyczące bezpieczeństwa i ochrony

Katalog wymagań dotyczących bezpieczeństwa i ochrony

ID: *Acknowledgements*

Nazwa: Potwierdzanie

Pochodzenie: *Superuser, Engineer, User*

Dziedziny: Niezawodność systemu

Typ: Jakościowe

Priorytet: Wysoki

Treść wymagania: Komunikacja między węzłami wymaga potwierdzeń.

ID: *Trusted-Security*

Nazwa: Zaufanie do autoryzacji

Pochodzenie: *Superuser, Engineer, User*

Dziedziny: Bezpieczeństwo

Typ: Jakościowe

Priorytet: Bardzo wysoki

Treść wymagania: Gwarancją dla korzystania z usług w systemie rozproszonym jest pozytywna weryfikacja użytkownika przez węzeł macierzysty.

5.2.5 Wymagania związane z użytecznością

Katalog wymagań związanych z użytecznością

ID: *Distributed-Processing*

Nazwa: Przetwarzanie rozproszone

Pochodzenie: *Superuser*, *Engineer*, *User*

Dziedzina: Funkcjonalność systemu

Typ: Jakościowe

Priorytet: Bardzo wysoki

Treść wymagania: System ma stanowić środowisko umożliwiające programom działanie w sposób rozproszony.

ID: *Usage-Comfort*

Nazwa: Wygoda użytkowania

Pochodzenie: *Superuser*, *Engineer*, *User*

Dziedzina: Funkcjonalność systemu

Typ: Jakościowe

Priorytet: Wysoki

Treść wymagania: Użytkowanie systemu powinno być możliwie proste.

ID: *Easy-System-Expansion*

Nazwa: Łatwość rozszerzania systemu

Pochodzenie: *Superuser*, *Engineer*, *User*

Dziedzina: Funkcjonalność systemu

Typ: Jakościowe

Priorytet: Wysoki

Treść wymagania: Rozszerzenie systemu na dany węzeł powinno wymagać jedynie instalacji na nim oprogramowania i wskazania go jakiemuś węzłowi systemu.

6 SPECYFIKACJA LOGICZNA SYSTEMU *DPE/UNIX*

Opis logiczny systemu spełniającego podane wymagania.

Struktura specyfikacji logicznej

Kroki		**Produkty wejściowe**	**Produkty wynikowe**
Specyfikacja logiczna		Katalog wymagań	Projekt logiczny systemu
1	Zdefiniuj moduły odpowiedzialne za rozproszone przetwarzanie i obsługujące rozproszone dane.	Katalog wymagań	Specyfikacja logiczna modułów
2	Zdefiniuj schematy komunikacji związane z rozproszonym przetwarzaniem i z operowaniem na rozproszonych danych.	Katalog wymagań	Specyfikacja logiczna schematów komunikacji
3	Zdefiniuj narzędzia odpowiedzialne za lokalne i zdalne zarządzanie elementami systemu.	Katalog wymagań	Specyfikacja logiczna narzędzi
4	Zdefiniuj algorytmy konieczne do realizacji modułów, schematów komunikacji oraz narzędzi.	Katalog wymagań, Specyfikacja logiczna modułów, Specyfikacja logiczna schematów komunikacji, Specyfikacja logiczna narzędzi	Definicje algorytmów

Struktura produktów specyfikacji logicznej

Produkty

Projekt logiczny systemu
- Specyfikacja logiczna modułów
- Specyfikacja logiczna schematów komunikacji
- Specyfikacja logiczna narzędzi
- Definicja algorytmów

6.1 Moduły

Logiczna specyfikacja składni i semantyki interfejsów modułów.

6.1.1 Moduł *DPE-Info-Module* informujący o stanie węzła

Specyfikacja modułu DPE-Info-Module

ID: *DPE-Info-Module*
Nazwa: Moduł informujący o stanie węzła

Pochodzenie: *Server-Liabilities, Operating-And-Network-System-Properties, Distributed-Processing*

Funkcja: Zbieranie i udostępnianie informacji nt. aktualnego stanu zasobów węzła.

Interfejs modułu:

Polecenie *SYS* – podaj listę nazw i wartości niezmiennych parametrów systemowych.

Polecenie *RES* – wymień nazwy i wartości parametrów określających dostępne zasoby.

Uwagi:

Wymaga się, aby lista parametrów systemowych zawierała co najmniej następujące parametry: *DPE* – nazwa systemu *DPE/UNIX*, *DPEVER* – wersja systemu *DPE/UNIX*, *OS* – nazwa systemu operacyjnego, *OSVER* – wersja systemu operacyjnego, *CPU* – nazwa/typ procesora, *MEM* – wielkość pamięci w megabajtach, *DISK* – wielkość pamięci dyskowej z partycją dla głównego katalogu, *TMP* – wielkość pamięci dyskowej z partycją dla katalogu plików tymczasowych, *USR* – wielkość pamięci dyskowej dla katalogu */usr*, *SWAP* – wielkość pamięci obszaru wymiany.

Wymaga się, aby lista parametrów określających dostępne zasoby zawierała co najmniej następujące parametry: *CPUUTIL* – procentowy wskaźnik wykorzystania procesora, *FREEMEM* – wielkość wolnej pamięci w megabajtach, *FREEDISK*/*FREETMP*/*FREEUSR*/*FREESWAP* – wielkości wolnej stosownej pamięci dyskowej w megabajtach, *LOGINS* – liczba pracujących użytkowników, *PROCS* – liczba działających procesów.

6.1.2 Moduł *DPE-Services-Module* lokalnej części rozproszonej bazy usług

Specyfikacja modu³u DPE-Services-Module

ID: *DPE-Services-Module*

Nazwa: Moduł lokalnej części rozproszonej bazy usług

Pochodzenie: *Server-Liabilities*

Funkcja: Udostępnianie informacji o usługach dostępnych z danego węzła.

Interfejs modułu:

Polecenie *ARGS*(*a*) – przyjmij *a* za argumenty programu.

Polecenie *GCOND*(*c*) – przyjmij *c* za warunek globalny.

Polecenie *LCOND*(*c*) – przyjmij *c* za warunek lokalny.

Polecenie *LIST* – podaj listę wybranych usług korzystając z przyjętych wartości.

Polecenie *NICE*(*v*) – przyjmij *v* za wartość *nice*.

Polecenie *PATH*(*p*) – przyjmij *p* za ścieżkę programu.

Polecenie *PROG*(*n*) – przyjmij *n* za nazwę programu.

Polecenie *RES*(*l*) – przyjmij *l* za listę wymagań programu.

Uwagi:

Polecenie *LIST* ma podać listę opisów usług dostępnych do wykonania z danego węzła (lista ma zawierać następujące informacje: nazwa programu, warunek lokalny, warunek globalny, listę wymagań, wartość *nice*, ścieżkę programu, domyślne argumenty programu).

Wartość *nice* dla programu jest parametrem modyfikującym priorytet z jakim ten program zostanie wykonany.

6.1.3 Moduł *DPE-Select-Module* wybierający węzeł optymalny

Specyfikacja modu³u DPE-Select-Module

ID: *DPE-Select-Module*
Nazwa: Moduł wybierający węzeł optymalny
Pochodzenie: *Server-Liabilities, Execution-Placement-Transparency*
Funkcja: Odnajdywanie węzła optymalnego do wykonania danej usługi w danej chwili, przy danych wymaganiach.
Interfejs modułu:
Polecenie *DEEP*(*d*) – przyjmij *d* za głębokość poszukiwania.
Polecenie *FIND* – znajdź i podaj nazwę optymalnego węzła do wykonania usługi, korzystając z przyjętego opisu usługi (programu).
Polecenie *GCOND*(*c*) – przyjmij *c* za warunek globalny.
Polecenie *HOST*(*h*) – przyjmij *h* za nazwę węzła klienta.
Polecenie *LIST* – podaj listę nazw węzłów sąsiednich.
Polecenie PROG(n) – przyjmij n za nazwę programu.
Polecenie RES(l) – przyjmij l za listę wymagań programu.
Polecenie STAMP(s) – przyjmij s za stempel zapytania.
Polecenie TIDY – usuń stemple po zapytaniu.
Uwagi:
Odnalezienie węzła optymalnego i usunięcie stempli odbywa się przy pomocy algorytmu *Find-Best-Node-Algorithm*.

6.1.4 Moduł *DPE-Exec-Module* realizujący zdalne wykonywanie

Specyfikacja modułu DPE-Exec-Module

ID: *DPE-Exec-Module*
Nazwa: Moduł realizujący zdalne wykonywanie
Pochodzenie: *Server-Liabilities*
Funkcja: Zdalne wykonywanie programu usługowego.
Interfejs modułu:
Polecenie *ARGS*(*a*) – przyjmij *a* za argumenty programu.
Polecenie *CODE* – podaj kod zakończenie programu, jeśli jest on już dostępny.
Polecenie *ENV*(*e*) – przyjmij *e* za rozszerzenie środowiska programu do wykonania.
Polecenie *EXEC* – wykonaj program w podprocesie.
Polecenie *HOST* – przyjmij nazwę węzła klienta usługi.
Polecenie *INPUT*(*n*, *p*) – przyjmij adres *p* na węźle *s* za źródło danych wejściowych dla programu do wykonania.
Polecenie *LOG*(*n*, *p*) – przyjmij adres *p* na węźle *n* za ujście informacji o błędach dla programu do wykonania.

Polecenie *OUTPUT*(*n*, *p*) – przyjmij adres *p* na węźle *n* za ujście danych wyjściowych dla programu do wykonania.

Polecenie *KILL*(*p*) – zakończ lokalny proces *p*.

Polecenie *NICE*(*v*) – przyjmij *v* za wartość *nice*.

Polecenie *PROG*(*n*) – przyjmij nazwę programu do wykonania.

Polecenie *PID*(*p*) – przyjmij proces *p* na węźle klienta za odpowiednik procesu lokalnego.

Polecenie SIG(s) – prześlij sygnał s do procesu-usługi.

Uwagi:

Przy zdalnym wykonywaniu usługi współpracuje ze sobą para procesów: jeden na węźle klienta i jeden na węźle serwera. Zniszczenie jednego z tych procesów powoduje zniszczenie drugiego (przekazywanie sygnałów).

Wartość nice dla programu jest parametrem modyfikującym priorytet z jakim ten program zostanie wykonany.

6.1.5 Moduł *DPE-Filesys-Module* implementujący rozproszony system plików

Specyfikacja modułu DPE-Filesys-Module

ID: *DPE-Filesys-Module*

Nazwa: Moduł implementujący rozproszony system plików

Pochodzenie: *Local-Files-Access-Transparency, Remote-Files-Access-Transparency*

Funkcja: Możliwość operowania na zdalnych plikach.

Interfejs modułu:

Polecenie *ASCII* – przyjmij tekstowy tryb transmisji (domyślny).

Polecenie *BIN* – przyjmij binarny tryb transmisji.

Polecenie *OPEN*(*n*, *f*) – otwórz plik *f* z węzła *n*.

Polecenie *CLOSE*(*f*) – zamknij plik *f*.

Polecenie *GET*(*f*) – pobierz plik *f* tworząc lokalną replikę.

Polecenie PUT(f) – prześlij plik f na węzeł, z którego został pobrany.

Uwagi:

Otwarcie zdalnego pliku może oznaczać przesłanie go na węzeł, z którego jest otwierany (lokalny) i otwarcie go lokalnie. Wówczas zamknięcie pliku oznaczać może również odesłanie zmodyfikowanego pliku na węzeł zdalny.

6.2 Komunikacja

6.2.1 Schemat *DPE-Info-Requests* zapytań o stan węzła

Specyfikacja zapytań DPE-Info-Requests

ID: *DPE-Info-Request-System*

Nazwa: Zapytanie o system węzła

Pochodzenie: *Server-Liabilities, Operating-And-Network-System-Properties*

Funkcja: Uzyskanie informacji o systemie na węźle.

Definicja schematu:

1. Połączenie się z modułem *DPE-Info-Module* na węźle, którego system badamy.
2. Polecenie *SYS* – podaje listę nazw i wartości niezmiennych parametrów systemowych.
3. Zakończenie połączenia z modułem *DPE-Info-Module*.

ID: *DPE-Info-Request-Resources*

Nazwa: Zapytanie o zasoby węzła

Pochodzenie: *Server-Liabilities, Operating-And-Network-System-Properties*

Funkcja: Uzyskanie informacji o zasobach węzła.

Definicja schematu:

1. Połączenie się z modułem *DPE-Info-Module* na węźle, którego zasoby badamy
2. Polecenie *SYS* – podaje listę nazw i wartości niezmiennych parametrów systemowych.
3. Zakończenie połączenia z modułem *DPE-Info-Module*.

6.2.2 Schemat *DPE-Services-Request* zapytania o udostępniane usługi

Specyfikacja zapytania DPE-Services-Request

ID: *DPE-Services-Request*

Nazwa: Zapytanie o udostępniane usługi

Pochodzenie: *Server-Liabilities*

Funkcja: Uzyskanie informacji o dostępnych na węźle usługach (tzn. takich, które można wywołać).

Definicja schematu:

1. Połączenie się z modułem *DPE-Services-Module* na węźle, który udostępnia usługi, które chcemy poznać.
2. Opcjonalnie: polecenie *PROG*(*p*) – przyjmuje nazwę programu.
3. Opcjonalnie: polecenie *ARGS*(*a*) – przyjmuje argumenty programu.
4. Opcjonalnie: polecenie *GCOND*(*c*) – przyjmuje warunek globalny.
5. Opcjonalnie: polecenie *LCOND*(*c*) – przyjmuje warunek lokalny.
6. Opcjonalnie: polecenie *NICE*(*v*) – przyjmuje wartość *nice*.
7. Opcjonalnie: polecenie *RES*(*l*) – przyjmuje listę wymagań programu.
8. Polecenie *LIST* – podaje listę usług (jeśli użyto *PROG*, to co najwyżej jedna), które spełniają warunek zadany przez powyższe polecenia (równość odpowiednich argumentów).
9. Zakończenie połączenia z modułem *DPE-Services-Module*.

6.2.3 Schemat *DPE-Select-Request* zapytania o węzeł optymalny

Specyfikacja zapytania DPE-Select-Request

ID: *DPE-Select-Request*

Nazwa: Zapytanie o węzeł optymalny

Pochodzenie: *Server-Liabilities, Execution-Placement-Transparency*

Funkcja: Uzyskanie informacji o węźle optymalnym do wykonania danej usługi.

Definicja schematu:

1. Połączenie się z modułem *DPE-Select-Module,* z którego rozpoczynać się będzie poszukiwanie
2. Polecenie *HOST*(*h*) – przyjmij nazwę węzła klienta.
3. Opcjonalnie: polecenie *DEEP*(*d*) – przyjmij głębokość poszukiwania.
4. Polecenie *PROG*(*n*) – przyjmij nazwę programu.
5. Opcjonalnie: polecenie GCOND(c) – przyjmij warunek globalny.
6. Opcjonalnie: polecenie RES(l) – przyjmij listę wymagań programu.
7. Polecenie FIND – znajdź i podaj nazwę optymalnego węzła oraz wektor wartości wyrażeń z listy wymagań programu.
8. Zakończenie połączenia z modułem *DPE-Select-Module*.

6.2.4 Schemat *DPE-Exec-Request* zlecenia zdalnego wykonania

Specyfikacja zlecenia DPE-Exec-Request

ID: *DPE-Exec-Request*

Nazwa: Zdalne uruchomienie programu.

Pochodzenie: *Distributed-Processing, Server-Liabilities, Execution-Placement-Transparency*

Funkcja: Uzyskanie informacji o węźle optymalnym do wykonania danej usługi.

Definicja schematu:

1. Połączenie się z modułem *DPE-Exec-Module* na węźle zdalnym.
2. Polecenie *HOST*(*h*) – przyjmij nazwę węzła klienta.
3. Polecenie *PID*(*p*) – przyjmij proces *p* na węźle klienta za substytut procesu zdalnego.
4. Polecenie *PROG*(*n*) – przyjmij nazwę programu do wykonania.
5. Opcjonalnie: polecenie ARGS(a) – przyjmij argumenty programu.
6. Opcjonalnie: polecenie ENV(e) – przyjmij rozszerzenie środowiska programu.
7. Opcjonalnie: polecenie NICE(v) – przyjmij wartość nice.
8. Polecenie *INPUT*(*n*, *p*) – przyjmij adres sieciowy źródła danych wejściowych dla programu do wykonania.
9. Polecenie *OUTPUT*(*n*, *p*) – przyjmij adres sieciowy ujścia danych wyjściowych dla programu do wykonania.
10. Polecenie *LOG*(*n*, *p*) – przyjmij adres sieciowy ujścia informacji o błędach dla programu do wykonania.

11. Polecenie *EXEC* – wykonaj program.
12. Po otrzymaniu sygnału niszczącego: polecenie *SIG*(*s*) – prześlij sygnał *s* (niszczący) do procesu zdalnego.
13. Po otrzymaniu sygnału niszczącego: polecenie *CODE* – pobierz kod zakończenia programu.
14. Zakończenie połączenia z modułem *DPE-Exec-Module*.

6.2.5 Schematy *DPE-Filesys-Requests* zdalnych operacji na plikach

Specyfikacja schematów DPE-Filesys-Requests

ID: *DPE-Filesys-Request-Open*

Nazwa: Otwarcie zdalnego pliku

Pochodzenie: *Server-Liabilities, Local-Files-Access-Transparency, Remote-Files-Access-Transparency*

Funkcja: Uzyskanie możliwości operowania na treści zdalnego pliku.

Definicja schematu:

1. Połączenie się z modułem *DPE-Filesys-Module* na lokalnym węźle.
2. Polecenie *FLAGS*(*f*) – przekaż flagi dla *OPEN*.
3. Polecenie *OPEN*(*h*, *p*) – stwórz w lokalnym katalogu */tmp* kopię pliku *p* z węzła *h*; polecenie podaje ścieżkę do pliku w */tmp*.
4. Zakończenie połączenia z modułem *DPE-Filesys-Module*.

ID: *DPE-Filesys-Request-Close*

Nazwa: Zamknięcie zdalnego pliku

Pochodzenie: *Server-Liabilities, Local-Files-Access-Transparency, Remote-Files-Access-Transparency*

Funkcja: Przekazanie treści zdalnego pliku.

Definicja schematu:

1. Połączenie się z modułem *DPE-Filesys-Module* na lokalnym węźle.
2. Polecenie *CLOSE*(*h*, *p*) – usuń z lokalnego katalogu */tmp* kopię pliku *p* z węzła *h*; jeśli plik mógł być modyfikowany, to prześlij jego treść na węzeł *n*.
3. Zakończenie połączenia z modułem *DPE-Filesys-Module*.

6.3 Zarządzanie i użytkowanie systemu

Operacje lokalne

Rejestrowanie i wyrejestrowywanie może być przeprowadzane przez administratora. Z założenia, *DPE/UNIX* nie tworzy z maszyn wirtualnej ‘supermaszyny’, ale umożliwia korzystanie z jawnie dystrybuowanych usług.

Wykonywanie i niszczenie może być przeprowadzane przez każdego użytkownika.

Operacje zdalne

Wykonywanie i niszczenie może być przeprowadzane przez każdego użytkownika.

6.3.1 Narzędzie *DPE-Best-Tool* do odnajdywania węzła optymalnego

Specyfikacja narzędzia DPE-Best-Tool

ID: *DPE-Best-Tool*
Nazwa: Program do odnajdywania węzła optymalnego
Pochodzenie: *Server-Liabilities, Local-Files-Access-Transparency, Remote-Files-Access-Transparency*
Funkcja: Odnajdywanie węzła optymalnego do wykonania danej usługi.
Opis użycia narzędzia:
Użycie narzędzia *DPE-Best-Tool* z parametrem *s* – nazwa usługi.

6.3.2 Narzędzie *DPE-Run-Tool* do wykonywania usług

Specyfikacja narzędzia DPE-Run-Tool

ID: *DPE-Run-Tool*
Nazwa: Program do wykonywania usług
Pochodzenie: *Execution-Placement-Transparency, User-Comfort, Server-Liabilities, Distributed-Processing*
Funkcja: Uruchamianie podanej usługi.
Opis użycia narzędzia:
Użycie narzędzia *DPE-Run-Tool* z parametrami: *n* – *węzeł* (opcjonalnie), *s* – nazwa usługi. Niepodanie węzła oznacza zlecenie odnalezienia optymalnego węzła.
Uwagi: W przypadku, gdy nie określono węzła, warto skorzystać z narzędzia *DPE-Best-Tool*.

6.3.3 Narzędzie *DPE-Kill-Tool* do niszczenia usług

Specyfikacja narzędzia DPE-Kill-Tool

ID: *DPE-Kill-Tool*
Nazwa: Program do niszczenia usług
Pochodzenie: *User-Comfort, Server-Liabilities, Distributed-Processing*
Funkcja: Przerywanie pracy danej usługi.
Opis użycia narzędzia:
Użycie narzędzia *DPE-Kill-Tool* z parametrem *s* – identyfikator usługi.

Uwagi: Każda usługa w czasie wykonywania powinna posiadać swój identyfikator na węźle, z którego została uruchomiona.

6.4 Opis zastosowanych algorytmów

Algorytmy stosowane w systemach rozproszonych powinny dobrze działać w dowolnej skali.

6.4.1 Algorytm wyboru węzła optymalnego

Definicja algorytmu wyboru węzła optymalnego

ID: *Find-Best-Node-Algorithm*

Nazwa: Algorytm wyboru węzła optymalnego

Pochodzenie: *Execution-Placement-Transparency*

Cel: Wybór najlepszego, w danej chwili, węzła do wykonania usługi o znanych wymaganiach, przy danym warunku globalnym; przeszukiwane są węzły położone nie dalej niż dana odległość

Wejście: *h* – węzeł macierzysty (przy starcie węzeł lokalny), *p* – nazwa programu-usługi, *d* – maksymalna odległość, *g* – warunek globalny

Wyjście: *best* – nazwa węzła optymalnego

Treść algorytmu:

```
START
    stamp := unikalny stempel zapytania;
    best = find-best-node(h, p, stamp, d, g);
    tidy-after-find(h, stamp, d);
STOP.
PROCEDURE find-best-node(h, p, s, d, g) : (nazwa węzła, wektor zasobów):
    zapamiętaj stempel (s, h) w stamps-list (globalna lista stempli zapytań przetwarzanych na
    lokalnym węźle);
    IF nie został zapamiętany, bo już jest:
        RETURN (null, null);
    best-node := null;
    best-res-vec := null;
    IF p jest dostępny na węźle lokalnym:
        IF warunek lokalny AND g:
            best-res-vec := wektor zasobów węzła lokalnego;
    FOR EACH i – sąsiad węzła lokalnego:
        IF i = h OR d – (odległość do i) < 0:
            CONTINUE;
        (node, res-vec) := wykonaj find-best-node(węzeł lokalny, p, d – (odległość do i), g)
                           na wêŸle i;
```

```
                IF best-res-vec < res-vec:
                        best-node := node;
                        best-res-vec := res-vec;
        RETURN (best-node, best-res-vec);
END.
PROCEDURE tidy-after-find(h, s, d):
        usuń stempel (s, h) z listy stamps-list;
        IF nie został usunięty, bo nie było go na liście:
                RETURN;
        FOR EACH i – sąsiad węzła lokalnego:
                IF i = h OR d – (odległość do i) < 0:
                        CONTINUE;
                wykonaj tidy-after-find(węzeł lokalny, s, d) na węźle i;
END.
```

7 PROJEKT FIZYCZNY SYSTEMU *DPE/UNIX*

Projekt fizyczny systemu spełniającego podane wymagania. Uwaga: chodzi o system programowowy więc terminy nie odnoszą się zagadnień sprzętowych – i tak np. przez fizyczne składowe systemu rozumiemy moduły kodu implementującego system, a nie np. urządzenia.

Struktura projektu fizycznego

Kroki		**Produkty wejściowe**	**Produkty wynikowe**
Projekt fizyczny		Katalog wymagań, Projekt logiczny systemu	Specyfikacja platformy, Projekt fizyczny systemu,
1	Określ platformę sprzętową i programową.	Katalog wymagań	Specyfikacja platformy
2	Zdefiniuj demony jako realizacje modułów.	Specyfikacja logiczna modułów	Specyfikacja demonów,
3	Zdefiniuj protokoły jako realizacje schematów	Specyfikacja logiczna schematów komunikacji	Specyfikacja protokołów,
4	Zdefiniuj programy narzędziowe jako realizacje narzędzi.	Specyfikacja logiczna narzędzi	Specyfikacja programów narzędziowych
5	Zdefiniuj bibliotekę funkcji przetwarzania rozproszonego.	Specyfikacja platformy, Specyfikacja demonów, Specyfikacja protokołów, Specyfikacja programów narzędziowych	Specyfikacja biblioteki funkcji przetwarzania rozproszonego,
6	Zdefiniuj bibliotekę funkcji nieprzenośnych.	Specyfikacja platformy, Specyfikacja demonów, Specyfikacja protokołów, Specyfikacja programów narzędziowych	Specyfikacja biblioteki funkcji nieprzenośnych

Struktura produktów projektu fizycznego

Produkty

Projekt fizyczny systemu
- Specyfikacja platformy
- Specyfikacja demonów
- Specyfikacja protokołów
- Specyfikacja programów narzędziowych
- Specyfikacja biblioteki funkcji przetwarzania rozproszonego
- Specyfikacja biblioteki funkcji nieprzenośnych

7.1 Platforma sprzętowa i programowa

Platformą dla systemu *DPE* jest sieć *TCP/IP* komputerów *UNIX*owych.

Specyfikacja platofrmy

ID: *Unix-TCP/IP-Platform*
Nazwa: Platofrma sprzętowa i programowa
Pochodzenie: *Operating-And-Network-System-Properties*
Opis platformy: UNIX System V lub BSD
Uwagi: Nie oczekuje się *RPC*, ale gniazd; sygnały z wersji System V.

7.2 Specyfikacja demonów

7.2.1 Demon *dpeinfod* informujący o stanie węzła

Specyfikacja demona dpeinfod

ID: *dpeinfod*
Nazwa: Demon informujący o stanie węzła
Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Info-Module*
Funkcja: Zbieranie i udostępnianie informacji nt. aktualnego stanu zasobów węzła.
Implementacja demona:

```
main() {
        przeczytaj plik konfiguracyjny – wczytaj definicje parametrów,
        wylicz ich początkowe wartości;
        towait = pewien czas;
        for (;;) {
                ustaw alarm na czas towait, a obsługę na funkcję tick;
                przyjmij klienta (accept());
                towait = wyłącz alarm, weź czas, który pozostał;
                if (nie przyjęto klienta, bo alarm) continue;
                if (fork() == 0) {
                        serveclient();
                        exit(0);
                }
        }
}
serveclient() {
        for (;;) {
                pobierz polecenie od klienta;
```

```
                if (nie ma) return;
                switch (polecenie) {
                case HELP:
                        prześlij listę nazw poleceń rozdzielonych
                        spacjami; break;
                case SYS:
                        prześlij listę wierszy z nazwami parametrów
                        systemowych i ich wartościami oddzielonymi
                        spacją; break;
                case RES:
                        j.w. tylko parametry określające zasoby; break;
                case QUIT:
                        return;
                }
        }
}
tick() {
        wylicz wartości parametrów określających zasoby;
}
```

Uwagi:

Parametry systemowe i określające zasoby, które są wymagane, zostały opisane w uwagach do specyfikacji modułu *DPE-Info-Module*.

Definicja wartości parametru w pliku konfiguracyjnym ma mieć postać wyrażenia dla systemowego programu *expr* (tzn. argumentów tego programu).

Plik konfiguracyjny ma składać się z wierszy definiujących parametry.

7.2.2 Demon *dpeservicesd* lokalnej części rozproszonej bazy usług

Specyfikacja demona dpeservicesd

ID: *dpeservicesd*

Nazwa: Demon lokalnej części rozproszonej bazy usług

Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Services-Module*

Funkcja: Udostępnianie informacji o usługach dostępnych z danego węzła.

Implementacja demona:

```
main() {
        przeczytaj plik konfiguracyjny – wczytaj opisy programów;
        for (;;) {
                przyjmij klienta (accept());
                if (fork() == 0) {
                        serveclient();
```

```
                    exit(0);
                }
        }
}
serveclient() {
        for (;;) {
                pobierz polecenie od klienta;
                if (nie ma) return;
                switch (polecenie) {
                case ARGS:
                        weŸ argumenty; break;
                case GCOND:
                        weŸ warunek globalny; break;
                case HELP:
                        prześlij listę nazw poleceń rozdzielonych
                        spacjami; break;
                case LCOND:
                        weź warunek lokalny; break;
                case LIST:
                        wypisz listę wierszy z opisami programów
                        pasujących do pobranych wartości; break;
                case NICE:
                        weź wartość nice; break;
                case PATH:
                        weŸ œcie¿kê programu; break;
                case PROG:
                        weŸ nazwê programu; break;
                case QUIT:
                        return;
                case RES:
                        weź listę wymagań do zasobów; break;
                }
        }
}
```

Uwagi:

Lista ma zawierać następujące informacje, rozdzielone spacjami: nazwa programu, warunek lokalny – w postaci wyrażenia dla *expr*, warunek globalny – j.w., listę wymagań – w postaci listy wyrażeń dla *expr* oddzielonych przecinkami, wartość *nice*, ścieżkę programu, domyślne argumenty programu.

Plik konfiguracyjny ma składać się z wierszy opisujących programy.

7.2.3 Demon *dpeselectd* wybierający węzeł optymalny

Specyfikacja demona dpeselectd

ID: *dpeselectd*
Nazwa: Demon wybierający węzeł optymany
Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Select-Module*
Funkcja: Udostępnianie informacji o usługach dostępnych z danego węzła.
Implementacja demona:

```
main() {
        przeczytaj plik konfiguracyjny – wczytaj opisy węzłów sąsiednich;
        numnodes = liczba tych węzłów;
        inicjuj fdset na pusty zbiór deskryptorów;
        FD_SET(gniazdo nasłuchiwania, &fdset);
        pipestab = NULL;
        numpipes  = 0;
        stampslist = NULL;
        for (;;) {
                setjmp(loopbuf, 0);
                przepisz fdset do readfds;
                signal(SIGCHLD, onsigchild);
                ustaw alarm na czas towait, a obsługę na funkcję tick;
                select(readfds);
                signal(SIGCHLD, SIG_IGN);
                towait = wyłącz alarm, weź czas, który pozostał;
                if (select nie powiód³ siê) continue;
                if (readfds ustawiony na przyjęcie klienta)
                        przyjmij klienta (accept());
                else {
                        for (i = 0; i < numpipes; i += 2)
                                if (w readfds ustawiony [i + 1][0]-y
                                   deskryptor z tablicy pipestab) {
                                        przeczytaj operację op i stempel
                                        z tego deskryptora;
                                        switch (op) {
                                        case zachowanie stempla:
                                                zachowaj stempel
                                                w stampslist;
                                                        case usuniêcie stempla:
                                                                usuñ stempel ze
                                                                  stampslist;
```

```
                                        }
                                }
                        continue;
                }
                przydziel nowe dwie pary deskryptorów z tablicy
                pipestab, tzn. o indeksach [i][0], [i][1], [i + 1][0], [i + 1][1];
                if (powiększono tablicę pipestab)
                        numpipes += 2;
                FD_SET(pipestab[i + 1][0], &fdset);
                if (fork() == 0) {
                        serveclient();
                        exit(i);
                }
        }
}
onsigchild() {
        i = kod zakończenia potomka;
        close(pipestab[i][0]);
        close(pipestab[i][1]);
        close(pipestab[i + 1][0]);
        close(pipestab[i + 1][1]);
        FD_CLR(pipestab[i + 1][0], &fdset);
        zwolnij pozycje tablicy pipestab o indeksach
        [i][0], [i][1], [i + 1][0], [i + 1][1];
        longjmp(loopbuf, 0);
}
tick() {
        alarm(0);
        usuń z stampslist stemple sprzed określonego okresu;
        longjmp(loopbuf, 0);
}
serveclient() {
        for (;;) {
                pobierz polecenie od klienta;
                if (nie ma) return;
                switch (polecenie) {
                case DEEP:
                        weź głębokość zapytania; break;
                case FIND:
                        zwróć klientowi wynik wywołania findbestnode
                        z podanymi parametrami, w tym z nazwą
```

```
                programu; break;
            case GCOND:
                weŸ warunek globalny; break;
            case HOST:
                weź nazwę węzła pytającego; break;
            case LIST:
                wypisz listę nazw węzłów sąsiednich; break;
            case PROG:
                weŸ nazwê programu; break;
            case QUIT:
                return;
            case RES:
                weź listę wymagań do zasobów; break;
            case STAMP:
                weŸ stampel zapytania; break;
            case TIDY:
                tidyafterfind(wêze³, stempel, g³êbokoœæ); break;
            }
        }
}
findbestnode(
        nazwa węzła klienta parenthost, nazwa programu name,
        warunek globalny gcond, lista wymagań resreq, stempel stamp,
        głębokość howdeep, parametr wyjściowy bestresvec) {
        if (savestamp(parenthost, stamp) zwraca informacjê,
           że stempel już został zapamiętany na lokalnym węźle)
                return NULL;
        besthost = ¿aden;
        bestresvec = pusty;
        if (program name jest dostępny na lokalnym węźle jako program)
                if (wartość warunku lokalnego programu &&
                   wartość warunku globalnego gcond) {
                        besthost = węzeł lokalny;
                        bestresvec = wektor wartości elementów resreq;
                }
        for (i = 0; i < numnodes; i++) {
                if (i-ty wêze³ to parenthost ||
                   howdeep – (dystans to i-tego) < 0) continue;
                połącz się z dpeselectd na i-tym węźle;
                prześlij tam parametry zapytania, w tym:
```

```
                - nazwę wezła lokalnego,
                - głębokość howdeep zmniejszoną o dystans do i-tego
                  węzła);
                odbierz informację o najlepszym węźle,
                w tym wektor resvec wartości elementów resreq;
                rozłącz się;
                if (resvec lepszy niż bestresvec) {
                        besthost = i-ty węzeł;
                        bestresvec = resvec;
                }
        }
        if (besthost == ¿aden)
                return NULL;
        return besthost;
}
tidyafterfind(nazwa wêz³a klienta parenthost, stempel stamp,
            głębokość deep) {
        if (removestamp(parenthost, stamp) zwraca informację,
           że nie można usunąć, bo parenthost inny niż przy
           zapamiętywaniu) return 0;
        for (i = 0; i < numnodes; i++)
                if (i-ty wêze³ to parenthost ||
                   howdeep – (dystans to i-tego) < 0) continue;
                połącz się z dpeselectd na i-tym węźle;
                prześlij tam parametry dla polecenia TIDY, tj.:
                - nazwa węzła lokalnego,
                - głębokość howdeep zmniejszoną o dystans do i-tego
                  węzła);
                rozłącz się;
        }
        return 0 – jeśli O.K., -1 – jeśli jakiś błąd;
}
savestamp(parenthost, stamp) {
        prześlij potokiem pipestab[indeks deskryptora
                                do procesu-rodzica][1]
        polecenie zachowania parenthost i stamp;
        return kod odebrany z pipestab[j.w.][0];
}
removestamp(parenthost, stamp) {
        prześlij potokiem pipestab[indeks deskryptora
                                do procesu-rodzica][1]
```

```
        polecenie usunięcia parenthost i stamp;
        return kod odebrany z pipestab[j.w.][0];
}
```

7.2.4 Demon *dpeexecd* realizujący zdalne wykonywanie

Specyfikacja demona dpeexecd

ID: *dpeexecd*

Nazwa: Demon realizujący zdalne wykonywanie

Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Exec-Module*

Funkcja: Obsługa zdalnego wykonania programu, przekazywanie wejścia/wyjścia/błędów i sygnałów.

Implementacja demona:

```
main() {
        for (;;) {
                setjmp(loopbuf, 0);
                signal(SIGCHLD, onsigchild1);
                przyjmij klienta (accept());
                signal(SIGCHLD, SIG_IGN);
                if (fork() == 0) {
                        serveclient();
                        exit(0);
                }
        }
}
serveclient() {
        inport = outport = logport = -1;
        status = 0;
        for (;;) {
                setjmp(servebuf);
                signal(SIGCHLD, onsigchild2);
                pobierz polecenie od klienta;
                signal(SIGCHLD, SIG_IGN);
                if (sygnał przerwał pobieranie polecenia) continue;
                if (nie ma) return;
                switch (polecenie) {
                case ARGS:
                        weŸ argumenty polecenia;
                        break;
                case CODE:
```

```
            if (status > 1)
                zwróć klientowi kod exitcode;
            break;
        case ENV:
            weź środowisko dla programu do wykonania;
            break;
        case EXEC:
            tmpfile = argument polecenia CLOSE,
                tj. plik z /tmp;
            if (inport != -1)
                sockin = połącz się z węzłem inhost,
                    port inport;
            else sockin = -1;
            if (outport != -1)
                sockin = połącz się z węzłem outhost,
                    port outport;
            else sockout = -1;
            if (logport != -1)
                sockerr = połącz się z węzłem loghost,
                    port logport;
            else sockerr = -1;
            pid = runprogram(przekazane parametry,
                    a w tym nazwa programu,
                    sockin, sockout, sockerr);
            status++; break;
        case HOST:
            weź nazwę węzła klienta; break;
        case INPUT:
            weŸ nazwê wêz³a inhost, numer portu inport;
            break;
        case OUTPUT:
            weŸ nazwê wêz³a outhost, numer portu outport;
            break;
        case LOG:
            weŸ nazwê wêz³a loghost, numer portu logport;
            break;
        case KILL:
            kill(argument jako PID procesu, SIGTERM);
            break;
        case NICE:
            weź warość nice dla programu do wykonania;
```

```
                        break;
                case PROG:
                        weź nazwę programu do wykonania; break;
                case PID:
                        weŸ argument jako PID procesu klienta; break;
                case QUIT:
                        return;
                case SIG:
                        kill(pid, argument jako numer sygna³u); break;
                }
        }
}
runprogram(nazwa programu prog, argumenty args, środowisko env,
            wartoœæ nice, sockin, sockout, sockerr, wêze³ clienthost) {
        if ((pid = fork()) > 0) return pid;
        na podstawie listy argumentów args zbuduj tablicę argv;
        na podstawie listy środowiska env zbuduj tablicę envp;
        if (sockin != -1) {
                fclose(stdin); dup(sockin);
        }
        if (sockout != -1) {
                fclose(stdout); dup(sockout);
        }
        if (sockerr != -1) {
                fclose(stderr); dup(sockerr);
        }
        if (uruchom prog z argumentami argv i środowiskiem envp) {
                status++; exitcode = 0;
        }
}
onsigchild1() {
        longjmp(loopbuf, 0);
}
onsigchild2() {
        status++;
        połącz się z dpeexecd na węźle clienthost;
        wyślij tam polecenie KILL z parametrem clientpid;
        rozłącz się;
        longjmp(servebuf, 0);
}
```

7.2.5 Demon *dpefilesysd* rozproszonego systemu plików

Specyfikacja demona dpefilesysd

ID: *dpefilesysd*
Nazwa: Demon rozproszonego systemu plików
Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Filesys-Module*
Funkcja: Umożliwienia zdalnego dostępu do plików.
Implementacja demona:

```
main() {
        inicjuj fdset na pusty zbiór deskryptorów;
        FD_SET(gniazdo nasłuchiwania, &fdset);
        pipestab = NULL;
        numpipes  = 0;
        fileslist = NULL;
        for (;;) {
                setjmp(loopbuf, 0);
                przepisz fdset do readfds;
                signal(SIGCHLD, onsigchild);
                select(readfds);
                signal(SIGCHLD, SIG_IGN);
                if (select nie powiód³ siê) continue;
                if (readfds ustawiony na przyjęcie klienta)
                        przyjmij klienta (accept());
                else {
                        for (i = 0; i < numpipes; i += 2)
                                if (w readfds ustawiony [i + 1][0]-y
                                   deskryptor z tablicy pipestab) {
                                        przeczytaj operację op, nazwę
                                        węzła i pliku z tego deskryptora;
                                        switch (op) {
                                        case zachowanie nazwy pliku:
                                                zachowaj nazwę węzła,
                                                pliku i flagi w fileslist;
                                                        case usunięcie nazwy pliku:
                                                                usuń nazwę węzła,
                                                                pliku i flagi z fileslist;
                                        }
                                }
                        continue;
                }
```

```
                przydziel nowe dwie pary deskryptorów z tablicy
                pipestab, tzn. o indeksach [i][0], [i][1], [i + 1][0], [i + 1][1];
                if (powiększono tablicę pipestab)
                        numpipes += 2;
                FD_SET(pipestab[i + 1][0], &fdset);
                if (fork() == 0) {
                        serveclient();
                        exit(i);
                }
        }
}
onsigchild() {
        i = kod zakończenia potomka;
        close(pipestab[i][0]);
        close(pipestab[i][1]);
        close(pipestab[i + 1][0]);
        close(pipestab[i + 1][1]);
        FD_CLR(pipestab[i + 1][0], &fdset);
        zwolnij pozycje tablicy pipestab o indeksach
        [i][0], [i][1], [i + 1][0], [i + 1][1];
        longjmp(loopbuf, 0);
}
tick() {
        alarm(0);
        usuń z stampslist stemple sprzed określonego okresu;
        longjmp(loopbuf, 0);
}
serveclient() {
        for (;;) {
                pobierz polecenie od klienta;
                if (nie ma) return;
                switch (polecenie) {
                case ASCII:
                        ustaw tryb transmisji danych curmode na ASCII;
                        break;
                case BIN:
                        ustaw tryb transmisji danych curmode na
                        binarny; break;
                case CLOSE:
                        tmpfile = argument polecenia CLOSE,
                                tj. plik z /tmp;
```

```
                wywołaj freetmpfile(tmpfile, ...) pobierając nazwę
                węzła (nodename), pliku (filepath) zdalnego i
                flagi (flags);
                if (flags to O_RDWR lub O_WRONLY)
                {
                        połącz się z dpefsysd na węźle
                        nodename;
                        prześlij tam polecenie BIN;
                        prześlij tam polecenie PUT z
                        argumentem filepath;
                        prześlij tam treść pliku tmpfile;
                        roz³¹cz siê;
                }
                unlink(tmpfile);
                break;
        case FLAGS:
                pobierz flagi otwarcia pliku (curflags);
                break;
        case GET:
                filepath = argumenty polecenia GET;
                fsize = rozmiar pliku filepath;
                fd = open(filepath, O_RDONLY);
                if (fd == -1 && getfile(fd, gniazdo poleceñ,
                                        curmode, fsize) == 0) {
                        close(fd);
                        zwróć O.K. klientowi;
                }
                else {
                        if (fd != -1) close(fd);
                        zwróć błąd klientowi;
                }
                break;
        case OPEN:
                weź nodename i filepath z argumentów;
                sock2 = połącz się z dpefsysd na węźle
                         nodename;
                prześlij tam polecenie BIN;
                prześlij tam polecenie GET
                z argumentem filepath;
                i = rozmiar pliku filepath;
                tmpfile = addtmpfile(nodename, filepath,
```

```
                                        curflags);
                    prześlij tam treść pliku tmpfile;
                    fd = open(tmpfile,
                            O_CREAT|O_EXCL|O_WRONLY);
                    while (i-- > 0 && read(sock2, buf, 1) != 0)
                        if (write(fd, buf, 1) == -1) {
                            close(sock2);
                            zwróć błąd klientowi; break;
                        }
                    close(fd);
                    roz³¹cz siê (close(sock2));
                    break;
                case PUT:
                    filepath  = argument polecenia;
                    fsize = d³ugoœæ pliku filepath;
                    fd = open(filepath, O_WRONLY|O_CREAT);
                    if (fd != -1 && putfile(fd, gniazdo poleceñ,
                                curmode, fsize) == 0) {
                        close(fd);
                        zwróć O.K. klientowi;
                    }
                    else {
                        if (fd != -1) {
                            lseek(fd, 0, SEEK_END);
                            close(fd);
                        }
                        else
                            putfile(-1, gniazdo poleceñ,
                                    curmode, fsize);
                        zwróć błąd klientowi;
                    }
                    break;
                case QUIT:
                    return;
                case UID:
                    curuid = argument; setuid(curuid);
                    break;
                }
        }
}
getfile(fd, sock, mode, fsize) {
```

```
        if (mode == ASCII) {
                zapisz 2 * fsize w sock;
                while (read(fd, &b, 1) != 0) {
                        c[0...1] = szestnastkowy zapis bajtu b;
                        zapisz c w sock;
                }
        }
        else {
                zapisz fsize w sock;
                while (read(fd, &b, 1) != 0)
                        zapisz b w sock;
        }
        return 0;
}
putfile(fd, sock, mode, fsize) {
        if (mode == ASCII) {
                read(sock, &c, 1);
                zapisz 2 * fsize w sock;
                while (nie koniec)
                        b = wartość cyfry szestnastkowej c << 4;
                        read(sock, &c, 1);
                        b |= wartość cyfry szestanstkowej c;
                        if (fd != -1)
                                write(fd, &b, 1);
                        read(sock, &c, 1);
                }
        }
        else {
                while (fsize--) {
                        read(sock, &b, 1);
                        if (fd != -1) write(fd, &b, 1);
                }
        }
        return 0;
}
addtmpfile(node, file, flags) {
        prześlij potokiem pipestab[indeks deskryptora
                                do procesu-rodzica][1]
        polecenie zachowania node, file, flags;
        return kod odebrany z pipestab[j.w.][0];
}
```

freetmpfile(*tmpfile*, parametr wyjściowy *node*, parametr wyjściowy *file*,
parametr wyjściowy *flags*) {
prześlij potokiem *pipestab*[indeks deskryptora
do procesu-rodzica][1]
polecenie usunięcia nazwy *tmpfile*;
zapisz w *node*, *file*, *flags* informacje odebrane z *pipestab*[j.w.][0];
}

7.3 Specyfikacja protokołów

7.3.1 Protokół *dpeinfoproto* zapytań o stan węzła

Specyfikacja protokołu dpeinfoproto

ID: *dpeinfoproto/system*

Nazwa: Zapytanie o system węzła

Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Info-Request-System*

Funkcja: Uzyskanie informacji o systemie na węźle *n*.

Definicja protokołu:

sock = połącz się z demonem *dpeinfod* na *n*;
prześlij polecenie *SYS* do *sock*;
odbierz z *sock* listę wierszy, z których każdy zawiera nazwę
parametru i jego wartość;
close(*sock*);

Uwagi: Lista parametrów systemowych zawiera co najmniej: *DPE*, *DPEVER*, *OS*, *OSVER*, *CPU*, *MEM*, *DISK*, *TMP*, *USR*, *SWAP*.

ID: *dpeinfoproto/resources*

Nazwa: Zapytanie o zasoby węzła

Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Info-Request-Resources*

Funkcja: Uzyskanie informacji o zasobach węzła *n*.

Definicja protokołu:

sock = połącz się z demonem *dpeinfod* na *n*;
prześlij polecenie *RES* do *sock*;
odbierz z *sock* listę wierszy, z których każdy zawiera nazwę
parametru i jego aktualną wartość;
close(*sock*);

Uwagi: Lista parametrów zasobów zawiera co najmniej: *CPUUTIL*, *FREEMEM*, *FREEDISK, FREETMP, FREEUSR, FREESWAP, LOGINS, PROCS*.

7.3.2 Protokół *dpeservicesproto* zapytania o udostępniane usługi

Specyfikacja protokołu dpeservicesproto

ID: *dpeservicesproto*

Nazwa: Zapytanie o udostępniane usługi

Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Services-Request*

Funkcja: Dostarczanie informacji o serwisach na węźle *n*.

Definicja protokołu:

sock = połącz się z demonem *dpeinfod* na *n*;

prześlij polecenie *SYS* do *sock*;

odbierz z *sock* listę wierszy, z których każdy zawiera nazwę

parametru i jego wartość;

close(*sock*);

Uwagi: Lista parametrów systemowych zawiera co najmniej: *DPE*, *DPEVER*, *OS*, *OSVER*, *CPU*, *MEM*, *DISK*, *TMP*, *USR*, *SWAP*.

7.3.3 Protokół *dpeselectproto* zapytania o węzeł optymalny

Specyfikacja protokołu dpeselectproto

ID: *dpeselectproto*
Nazwa: Zapytanie o węzeł optymalny
Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Select-Request*
Funkcja: Uzyskanie informacji o węźle optymalnym do wykonania danej usługi.
Definicja protokołu:

sock = połącz się z lokalnym demonem *dpeselectd*;
opcjonalnie: prześlij polecenie *HOST h* do *sock*, gdzie *h* to węzeł klienta;
opcjonalnie: prześlij polecenie *DEEP d* do *sock*;
prześlij polecenie *PROG n* do *sock*;
opcjonalnie: prześlij polecenie *GCOND c* do *sock*;
opcjonalnie: prześlij polecenie *RES l* do *sock*;
prześlij polecenie *FIND* do *sock*;
odbierz z *sock* nazwę optymalnego węzła oraz wektor wartości wyrażeń
z listy wymagań programu;
close(*sock*);

7.3.4 Protokół *dpeexecproto* zlecenia zdalnego wykonania

Specyfikacja protokołu dpeexecproto

ID: *dpeexecproto*
Nazwa: Zdalne uruchomienie programu
Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Exec-Request*
Funkcja: Uruchomienie i kontrola programu-serwisu na zdalnym komputerze *n*.
Definicja protoko³u:

signal(*SIGTERM*, *onsigterm*);
sock = połącz się z demonem *dpeexecd* na *n*;
prześlij polecenie *HOST h* do *sock*;
prześlij polecenie *PID p* do *sock*, gdzie *p* == *PID* lokalnego procesu,
'reprezentanta' zdalnego procesu;
prześlij polecenie *PROG n* do *sock*;
prześlij polecenia: *ARGS a, ENV e, NICE v* (opcjonalne) do *sock*;
sockin = *socket*(*AF_INET*, *SOCK_STREAM*, 0);
name.sin_family = *AF_INET*; *name.sin_addr.s_addr* =
htonl(*INADDR_ANY*); *name.sin_port* = 0;

bind(*sock*, *&name*, ...); *getsockname*(*sock*, *&name*, ...);
p = *ntohs*(*name.sin_port*);
prześlij polecenie *INPUT n ip* do *sock*;
sockout = *socket*(*AF_INET*, *SOCK_STREAM*, 0);
name.sin_family = *AF_INET*; *name.sin_addr.s_addr* =
htonl(*INADDR_ANY*); *name.sin_port* = 0;
bind(*sock*, *&name*, ...); *getsockname*(*sock*, *&name*, ...);
p = *ntohs*(*name.sin_port*);
prześlij polecenie *OUTPUT n p* do *sock*;
sockerr = *socket*(*AF_INET*, *SOCK_STREAM*, 0);
name.sin_family = *AF_INET*; *name.sin_addr.s_addr* =
htonl(*INADDR_ANY*); *name.sin_port* = 0;
bind(*sock*, *&name*, ...); *getsockname*(*sock*, *&name*, ...);
p = *ntohs*(*name.sin_port*);
prześlij polecenie *LOG n p* do *sock*;
prześlij polecenie *EXEC* do *sock*;
while (są dane) {
pobieraj *stdout/stderr* z *sockout/sockerr*,
a przekazuj *stdin* do *sockin*;
}
close(*sockin*); close(*sockout*); close(*sockerr*);
onsigterm(*s*) {
prześlij polecenie *SIG s* do *sock*;
prześlij polecenie *CODE* do *sock*;
pobierz z *sock* kod zakończenia procesu-serwisu;
close(*sock*);
}

Komentarz [RM1]:

7.3.5 Protokół *dpefilesysproto* zdalnego dostępu do pliku

Specyfikacja protokołu dpefilesysproto

ID: *dpefilesysproto/open*
Nazwa: Otwarcie zdalnego pliku
Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Filesys-Request-Open*
Funkcja: Uzyskanie możliwości operowania na treści zdalnego pliku na węźle *h*.
Definicja protokołu:
sock = połącz się z lokalnym demonem *dpefilesysd*;
prześlij polecenie *FLAGS f* do *sock*;
prześlij polecenie *OPEN h p* do *sock*;

odbierz z *sock* ścieżkę do pliku w */tmp*;

close(*sock*);

ID: *dpefilesysproto/close*

Nazwa: Zamknięcie zdalnego pliku

Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Filesys-Request-Close*

Funkcja: Przekazanie treści zdalnego pliku *p* na węźle *h*.

Definicja protokołu:

sock = połącz się z lokalnym demonem *dpefilesysd*;

prześlij polecenie *CLOSE h p* do *sock*;

close(*sock*);

7.4 Specyfikacja programów narzędziowych

7.4.1 Program *dpebest* do odnajdywania węzła optymalnego

Specyfikacja programu dpebest

ID: *dpebest*

Nazwa: Program do odnajdywania węzła optymalnego

Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Best-Tool*

Funkcja: Odnajdywanie węzła optymalnego do wykonania danej usługi.

Opis użycia programu:

dpebest s d

gdzie: *s* – nazwa programu-serwisu, *d* – głębokość wyszukiwania (maks. odległość do rozpatrywanych węzłów; opcja)

Uwagi: Program podaje nazwę (*FQDN*) optymalnego węzła i wektor wartości zasobów.

7.4.2 Program *dperun* do wykonywania usług

Specyfikacja programu dperun

ID: *dperun*

Nazwa: Program do wykonywania usług

Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Run-Tool*

Funkcja: Uruchamianie podanej usługi.

Opis użycia programu:

dperun h:*s*

dperun s d

gdzie: *h* – nazwa węzła (opcjonalnie), *s* – nazwa programu-serwisu, *d* – głębokość wyszukiwania (maks. odległość do rozpatrywanych węzłów; opcja)

Uwagi: Program podaje nazwę (*FQDN*) optymalnego węzła i wektor wartości zasobów.

7.4.3 Program *dpekill* do niszczenia usług

Specyfikacja programu dpekill

ID: *dpekill*

Nazwa: Program do niszczenia usług

Pochodzenie: *Unix-TCP/IP-Platform*, *DPE-Kill-Tool*

Funkcja: Przerywanie pracy podanej usługi.

Opis użycia programu:

dpekill s

gdzie: *s* – identyfikator programu-serwisu – jest to *PID* procesu 'reprezentującego' lokalnie (tj. w miejscu uruchomienia) zdalny proces-usługę.

7.5 Specyfikacja biblioteki funkcji przetwarzania rozproszonego

Specyfikacja biblioteki *libdpe* funkcji udostępniających mechanizmy przetwarzania rozproszonego.

Specyfikacja funkcji biblioteki libdpe

ID: *dpeconnecttoinfod*

Nazwa: *dpeconnecttoinfod*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod, dpeseld, dpeinfoproto, dpeselproto*

Parametry: *const char *hostname*

Wynik: *int* – gniazdo łączące z demonem

Definicja funkcji:

return *sock* = połącz się z demonem *dpeinfod* na węźle *hostname*;

ID: *dpeconnecttosvcsd*

Nazwa: *dpeconnecttosvcsd*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeseld, dpeselproto*

Parametry: *const char *hostname*

Wynik: *int* – gniazdo łączące z demonem

Definicja funkcji:

return *sock* = połącz się z demonem *dpesvcsd* na węźle *hostname*;

ID: *dpeconnecttoseld*

Nazwa: *dpeconnecttoseld*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeseld, dpeselproto, dpebest, dperun*

Parametry: *const char *hostname*

Wynik: *int* – gniazdo łączące z demonem

Definicja funkcji:

return *sock* = połącz się z demonem *dpeseld* na węźle *hostname*;

ID: *dpeconnecttoexecd*

Nazwa: *dpeconnecttoexecd*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeexecd, dpeexecproto, dperun*

Parametry: *const char *hostname*

Wynik: *int* – gniazdo łączące z demonem

Definicja funkcji:

return *sock* = połącz się z demonem *dpeexecd* na węźle *hostname*;

ID: *dpeconnecttofsysd*

Nazwa: *dpeconnecttofsysd*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpefsysd, dpefsysproto*

Parametry: *const char *hostname*

Wynik: *int* – gniazdo łączące z demonem

Definicja funkcji:

return *sock* = połącz się z demonem *dpefsysd* na węźle *hostname*;

ID: *dpesendcmd*

Nazwa: *dpesendcmd*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeexecd, dpefsysd, dpeinfod, dpeseld, dpesvcsd*

Parametry: *int sock, const char *pcmd, const char *pargs*

Wynik: *int* – 0: O.K.; -1: błąd

Definicja funkcji:

strncpy(*buf*, *pcmd*, sizeof(*buf*) – 1);

if (*pargs* != *NULL*) dopisz " ", *pargs* i "\n" do *buf*;

return *write*(*sock*, *buf*);

ID: *dpereceiveline*

Nazwa: *dpereceiveline*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeexecd, dpefsysd, dpeinfod, dpeseld, dpesvcsd*

Parametry: *int sock, char *pbuf, int bufsize*

Wynik: *int* – liczba odebranych znaków; -1: błąd

Definicja funkcji:

readreturn = *read*(*sock*, *pbuf*, *bufsize*);

```
if (pbuf[strlen(pbuf) – 1] == '\n') pbuf[strlen(pbuf) – 1] = '\0';
return readreturn;
```

ID: *dpegetresourcevec*

Nazwa: *dpegetresourcevec*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeseld*

Parametry: const char *hostname, const char *resreq, char *resvec, int resvecsize

Wynik: *int* – 0: O.K.; -1: błąd

Definicja funkcji:

```
if (strcmp(resreq, "*")) == 0) return -1;
connecttoinfod(hostname);
dpesendcmd(sock, "SYS", NULL);
pobierz parametry systemowe do tablicy systemtab;
dpesendcmd(sock, "RES", NULL);
pobierz stan zasobów do tablicy resourcetab;
stwórz envtab z systemtab i resourcetab;
memset(resvec, 0, resvecsize);
dla wszystkich wyrażeń presreq z resreq
{
        val = toolevaluateexpr(presreq, (const char **) envtab);
        strncat(resvec, val, resvecsize – 2 – strlen(resvec));
        resvec[strlen(resvec)] = ',';
}
return 0;
```

ID: *dperesveccmp*

Nazwa: *dperesveccmp*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeseld*

lista identyfikatorów demonów, lista identyfikatorów protokołów, lista identyfikatorów programów>

Parametry: *const char *resvec1, const char *resvec2*

Wynik: *int* – -1: *resvec1* przed *resvec2*; 0: równe; 1: *resvec1* po *resvec2*

Definicja funkcji:

```
while (*resvec1 != '\0' && *resvec2 != 0 &&
        (d = (int) (atol(resvec1) – atol(resvec2))) == 0)
{
        resvec1 = następne wyrażenie z resvec1;
        resvec2 = następne wyrażenie z resvec2;
}
return (d < 0 ? -1 : (d > 0 ? 1 : 0));
```

ID: *dpegetprogramdesc*

Nazwa: *dpegetprogramdesc*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeseld*

Parametry: *const char *hostname, const char *programname*

Wynik: *DPEPROGRAMDESC **

Definicja funkcji:

sock = *connecttosvcsd*(*hostname*);

dpesendcmd(*sock*, “PROG”, *programname*);

dpesendcmd(*sock*, “LIST”, *programname*);

stwórz strukturę *DPEPROGRAMDESC* na podstawie wyniku “LIST”

ID: *dpegetbesthost*

Nazwa: *dpegetbesthost*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeseld*

Parametry: *const DPEPROGRAMDESC *pprogramdesc, char *bestresvec, int bestresvecsize*

Wynik: *char* * – nazwa optymalnego węzła

Definicja funkcji:

sock = *connecttoseld*(*NULL*);

weŸ do *hostname* nazwê wêz³a lokalnego;

dpesendcmd(*sock,* “HOST”, *hostname*);

dpesendcmd(*sock,* “DEEP”, pewna głębokość);

dpesendcmd(*sock,* “PROG”, pewna głębokość);

dpesendcmd(*sock,* “GCOND”, *pprogramdesc->gcond*);

dpesendcmd(*sock*, “RES”, *pprogramdesc->resreq*);

dpesendcmd(*sock*, “FIND”, *NULL*);

dpereceiveline(*sock*, *hostname*, sizeof(*hostname*));

if (*bestrecvec* != *NULL*)

dpereceiveline(*sock*, *bestrecvec*, bestrecvecsize – 1);

else

dpereceiveline(*sock*, jakiœ bufor, sizeof(jakiœ bufor));

return *hostname*;

ID: *dpeexecve*

Nazwa: *dpeexecve*

Pochodzenie: *Unix-TCP/IP-Platform*

Parametry: *const char *programname, const char *argv[],const char *envp[]*

Wynik: *int* – 0: O.K., -1: błąd

Definicja funkcji:

if ((*pprogramdesc* = *dpegetprogramdesc*(*NULL*, *programname*))

== *NULL*) return -1;

if ((*besthostname* = *dpegetbesthost*(*pprogramdesc*, *NULL*, 0))
NULL) return -1;
weŸ do *thishostname* nazwê wêz³a lokalnego;
if (*strcmp*(*besthostname, thishostname*) == 0)
return (u¿yj *pprogramdesc*, *argv*, *envp* do *execve*())
return *dperexecve*(*besthostname*, *programname*, *argv*, *envp*);

ID: *dpelexecve*
Nazwa: *dpelexecve*
Pochodzenie: *Unix-TCP/IP-Platform*
Parametry: *const char *programname, const char *argv[],const char *envp[]*
Wynik: *int* – 0: O.K., -1: błąd
Definicja funkcji:
if ((*pprogramdesc* = *dpegetprogramdesc*(*NULL*, *programname*))
NULL) return -1;
if ((*besthostname* = *dpegetbesthost*(*pprogramdesc*, *NULL*, 0))
NULL) return -1;
return (u¿yj *pprogramdesc*, *argv*, *envp* do *execve*())

ID: *dperexecve*
Nazwa: *dperexecve*
Pochodzenie: *Unix-TCP/IP-Platform*
Parametry: *const char *hostname*, *const char *programname, const char *argv[],const char *envp[]*
Wynik: *int* – 0: O.K., -1: błąd
Definicja funkcji:
argv2[0] = “dpestub”;
strcpy(*argv2*[1], *hostname*);
strcat(*argv2*[1], “:”);
strcat(*argv2*[1], *programname*);
return *execve*(“dpestub”, *argv2, envp*);

ID: *dpeopen*
Nazwa: *dpeopen*
Pochodzenie: *Unix-TCP/IP-Platform*
Parametry: *const char *pathname, int flags, mode_t mode*
Wynik: *int* – 0: O.K., -1: błąd
Definicja funkcji:
i = indeks zmiennej środowiskowej “dpeclient” z *environ*;
weź do *hostname* nazwę węzła lokalnego;
if (*environ*[*i*] == *NULL* ||

strcmp(*environ*[*i*] + *strlen*(“dpeclient”) + 1, *hostname*)
return *open*(*pathname, flags*, *mode*);
return *dperopen*(*environ*[*i*], *pathname*, *flags*, *mode*);

ID: *dperopen*
Nazwa: *dperopen*
Pochodzenie: *Unix-TCP/IP-Platform*
Parametry: *const char *hostname, const char *pathname, int flags, mode_t mode*
Wynik: *int* – 0: O.K., -1: błąd
Definicja funkcji:
if (*hostname* = *NULL*)
weź nazwę lokalnego węzła do *tmp*;
else
strcpy(*tmp*, *hostname*);
sock = *dpeconnecttofsysd*(*NULL*);
sprintf(*buf*, “%s %s”, *tmp*, *pathname*);
dpesendcmd(*sock*, “OPEN”, *buf*);
dpereceiveline(*sock*, *buf*, sizeof(*buf*));
strcpy(*tmp*, *buf*);
close(*sock*);
if ((*fd* = *open*(*tmp*, *flags*, *mode*)) != – 1)
dołącz *fd* i *tmp* (zdalna ścieżka) do listy
otwartych zdalnych plików;
return *fd*;

ID: *dpeclose*
Nazwa: *dpeclose*
Pochodzenie: *Unix-TCP/IP-Platform*
Parametry: *int fd*
Wynik: *int* – 0: O.K., -1: błąd
Definicja funkcji:
sock = *dpeconnecttofsysd*(*NULL*);
item = znajdź *fd* na liście otwartych zdalnych plików;
if (*item != NULL*)
usuń *item* z tej listy;
if (*item == NULL*)
return -1;
dpesendcmd(*sock*, “CLOSE”, zdalna œcie¿ka dla *item*);
dpereceiveline(*sock*, *buf,* sizeof(*buf*));
if (*strcmp*(*buf*, “OK.”) == 0)

```
            return 0;
        return -1;
```

7.6 Specyfikacja biblioteki funkcji nieprzenośnych

Biblioteka *libdpelink* funkcji zależnych od SO i sprzętu.

7.6.1 Funkcje określające parametry węzła

ID: *linkopsysname*
Nazwa: *linkopsysname*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *char* * – nazwa systemu operacyjnego, np. “Linux”

ID: *linkopsysver*
Nazwa: *linkopsysver*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *char* * – wersja systemu operacyjnego, np. “1.0.0”

ID: *linkcpuname*
Nazwa: *linkcpuname*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *char* * – wersja systemu operacyjnego, np. “i486”

ID: *linkmemsize*
Nazwa: *linkmemsize*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *int* – pamięć RAM w MB

ID: *linkdiskspace*
Nazwa: *linkdiskspace*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *int* – pojemność urządzenia dla systemu plików “/” w MB

ID: *linktmpspace*
Nazwa: *linktmpspace*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *int* – pojemność urządzenia dla katalogu “/tmp” w MB

ID: *linkusrspace*
Nazwa: *linkusrspace*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *int* – pojemność urządzenia dla katalogu “/usr” w MB

ID: *linkswapspace*
Nazwa: *linkswapspace*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *int* – wielkość obszaru wymiany (*swap space*) w MB

7.6.2 Funkcje badające obciążenie

Specyfikacja funkcji biblioteki libdpelink *badających obciążenie*

ID: *linkcpuutil*
Nazwa: *linkcpuutil*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *int* – wykorzystanie procesora: 0: min., 100: max.

ID: *linkfreememsize*
Nazwa: *linkfreememsize*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:
Wynik: *int* – wolna pamięć RAM w MB

ID: *linkfreediskspace*
Nazwa: *linkfreediskspace*
Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*
Parametry:

Wynik: *int* – wolna pojemność urządzenia dla systemu plików “/” w MB

ID: *linkfreetmpspace*

Nazwa: *linkfreetmpspace*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*

Parametry:

Wynik: *int* – wolna pojemność urządzenia dla katalogu “/tmp” w MB

ID: *linkfreeusrspace*

Nazwa: *linkfreeusrspace*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*

Parametry:

Wynik: *int* – wolna pojemność urządzenia dla katalogu “/usr” w MB

ID: *linkfreeswapspace*

Nazwa: *linkfreeswapspace*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*

Parametry:

Wynik: *int* – wolny obszar wymiany (*swap space*) w MB

ID: *linkloginsnum*

Nazwa: *linkloginsnum*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*

Parametry:

Wynik: *int* – liczba aktualnie zalogowanych użytkowników

ID: *linkprocnum*

Nazwa: *linkprocnum*

Pochodzenie: *Unix-TCP/IP-Platform*, *dpeinfod*

Parametry:

Wynik: *int* – aktualna liczba procesów

8 ZASTOSOWANIA SYSTEMU *DPE/UNIX*

Główne obszary zastosowań systemów przetwarzania rozproszonego w ogólności, to obliczenia naukowe oraz przetwarzanie danych w biznesie [8].

Przykładowe zastosowania w nauce:

- modelowanie klimatu, pogody, oceanów, trzęsień ziemi, przestrzeni kosmicznej;
- prognozowanie pogody;
- badanie nadprzewodnictwa, modelowanie dynamiki molekuł, przepływów płynów;
- obliczenia numeryczne, obliczenia metodą *Monte Carlo*;
- automatyczne dowodzenie;
- symulacje;
- wyszukiwanie informacji w sieciach rozległych;
- przetwarzanie obrazu i dźwięku, tworzenie i przetwarzanie animacji i fraktali.

Przykładowe zastosowania w biznesie:

- symulacje zjawisk ekonomicznych, zachowania rynku;
- trójwymiarowe modelowanie zasobów wód gruntowych, złóż bogactw naturalnych;
- praca grupowa;
- równoległe przetwarzanie danych.

8.1 Rola systemu DPE/UNIX

System *DPE/UNIX* może służyć jako najniższa warstwa złożonych aplikacji rozproszonych. Aplikacje takie mogą być projektowane od początku, albo być rozproszonymi wersjami jednostanowiskowych programów. Dobrze zaprojektowana aplikacja rozproszona powinna mieć następujące cechy:

- modularność – dzięki niej niezależne elementy będą mogły być wykonywane na różnych komputerach, efektywnie wykorzystując zasoby;
- słabe powiązanie modułów – aby nie obciążać sieci komunikacją między modułami;
- przenośność modułów – zapewni możliwość silnego rozproszenia przetwarzania w heterogenicznej sieci.

Zachowanie tych warunków jest konieczne dla skalowalności (względem wielkości grafu systemu). Nie wystarcza jednak, ponieważ taka aplikacja może wykorzystywać jedynie pewną, z góry ograniczoną, liczbę węzłów – zatem istotne jest, czy algorytm można podzielić na wiele modułów, które mogłyby się wykonywać równolegle.

W projektowaniu takich systemów można posłużyć się dwiema technikami, które autor stosował przy tworzeniu samego systemu *DPE/UNIX*: metodą abstrakcji i kapsułkowania (*encapsulation*), znanymi m.in. z programowania obiektowego.

8.2 Modele obliczeń równoległych

Dwa najbardziej popularne modele obliczeniowe w programowaniu rozproszonym to model gwiazdy i model drzewa. Często spotyka się wariacje tych modeli, np. gwiazda z procesami podrzędnymi połączonymi w pierścień, czy drzewo z liśćmi powiązanymi z korzeniem.

8.2.1 Model gwiazdy (*master-slave*)

W tym modelu zadanie jeste realizowane przez dwa rodzaje procesów:

- nadrzędny, ‘pan’ (ang. *master*), który tworzy, usuwa i nadzoruje pracę innych procesów, którym przekazuje dane i od których odbiera wyniki;
- procesy podrzędne, ‘niewolnicy’ (ang. *slaves*) wykonujący zadania obliczeniowe.

Zadanie zostaje więc podzielone na wiele mniejszych podzadań w taki sposób, że podzadania nie wymagają komunikacji, tj. wymiany informacji (danych) – wraz z procesem nadrzędnym tworzą strukturę gwiazdy.

8.2.2 Model drzewa

W modelu drzewa zadanie jest realizowane przez wiele procesów, które wymieniają informację drogami tworzącymi logicznie strukturę hierarchiczną, tj. drzewo. Patrząc inaczej, każdy ze składowych procesów, prócz liści drzewa, zleca pewnej liczbie procesów wykonanie części swojego zadania, a wyniki przekazuje do procesu-rodzica.

W tym modelu łatwo jest zaimplementować w sposób równoległy algorytmy oparte na strategii “dziel i rządź”, np. metodę podziału i ograniczeń (*branch-and-bound*), czy kompilację dużego systemu przy pomocy programu *make*.

8.3 Przykłady zastosowań

Przedstawiono cztery przykłady zastosowań systemu *DPE*:

- rozproszona metoda podziału i ograniczeń;
- równoległe dowodzenie twierdzeń;
- wyszukiwanie rozproszonych informacji w dużych sieciach;
- rozproszone bazy danych.

8.3.1 Metoda podziału i ograniczeń

Definicja problemu

Metoda podziału i ograniczeń jest techniką rozwiązywania problemów optymalizacji kombinatorycznej. Z problemami tego typu mamy do czynienia najczęściej w fizyce i inżynierii. Problem taki można zdefiniować następująco: znaleźć minimum/maksimum funkcji $f : Z^n \rightarrow R$ (wartość i argument), której dziedzinę ograniczono zestawem nierówności (dla $j = 1,...,m$):

$$\sum_{i = 1,...,n} a_{i,j} x_i \leq b_j$$

Metoda podziału i ograniczeń rozwiązuje takie zadania stosując strategię “dziel i rządź”. Początkowa przestrzeń x jest rekurencyjnie dzielona na podprzestrzenie, aż do osiągnięcia pojedynczych rozwiązań. Rekurencyjny podział można przedstawić w postaci drzewa, w którego liściach znajdują się rozwiązania. Nie całe drzewo jest przeszukiwane, ale tylko te gałęzie, w których można znaleźć rozwiązanie lepsze od dotychczas znalezionego, tzn. ‘obcina się’ te gałęzie, dla których można udowodnić, że nie zawierają rozwiązania poprawiającego.

Przykładem z klasy problemów opisanych wyżej jest tzw. problem plecakowy: zmaksymalizować funkcję

$$\sum_{i = 1,...,n} c_i x_i \qquad x_i = 0,1$$

przy ograniczeniach (dla $j = 1,...,m$):

$$\sum_{i=1,...,n} a_{i,j} x_i \leq b_j \quad a_{i,j} > 0, b_j > 0$$

Realizacja rozwiązania w systemie DPE

Rozwiązanie oparte jest na kilku zasadach. Każdy proces ma pewną wiedzę (lokalną): o najlepszym lokalnym rozwiązaniu i o lokalnej podprzestrzeni rozwiązań. Procesy mogą wymieniać tą wiedzę i aktualizować swoją. Najlepsze znalezione dotąd rozwiązanie jest przekazywane wszystkim procesom. Drzewo podziału przestrzeni przeszukiwane jest wgłąb. Komunikacja między procesami jest asynchroniczna.

Przyjmijmy model obliczeń drzewa. Algorytm przedstawia się wówczas w skrócie następująco:

1. Proces nadrzędny (koordynator) tworzy *p* procesów podrzędnych (podział przestrzeni na *p* podprzestrzeni).
2. Każdy proces podrzędny przeszukuje swoje poddrzewo wgłąb, uaktualniając lokalne najlepsze rozwiązanie i przesyłając je do koordynatora.
3. Gdy koordynator otrzymuje rozwiązanie, porównuje je z globalnie najlepszym rozwiązaniem i przesyła je do wszystkich procesów podrzędnych. Jeśli proces potrafi ocenić, że w przydzielonej mu podprzestrzeni nie ma lepszych rozwiązań, kończy działanie.

Implementacja tego rozwiązania w systemie *DPE* wymaga zakodowania koordynatora i procesów podrzędnych, które powinny być zarejestrowane jako rozproszone usługi. Niech *bbroot.c* będzie kodem koordynatora, a *bbnode.c* kodem węzłów. Usługę *bbnode* należy zarejestrować na wybranych węzłach, a na węźle użytkownika uruchamiającego program należy umieścić kod *bbroot*. Ten sam węzeł powinien przechowywać plik *bbglobal* z informacją o najlepszym rozwiązaniu.

Wymiana wiedzy odbywać się powinna przy pomocy rozproszonych plików i sygnałów. Asynchroniczną komunikację można zrealizować z pomocą neutralnego sygnału *SIGALRM* (po wprowadzeniu zmian do pliku należy przesłać ten sygnał). A zatem:

1. Proces *bbnode* znajdujący lepsze rozwiązanie zapisuje je w pliku *bbglobal* (przekazanie rozwiązania do koordynatora) i przesyła sygnał do koordynatora.
2. Proces-koordynator przekazuje sygnał do potomków, które przekazują go wgłąb drzewa.
3. Wszystkie procesy pobierają rozwiązanie z pliku *bbglobal*.

8.3.2 Automatyczne dowodzenie twierdzeń

Definicja problemu

Celem automatycznego dowodzenia twierdzeń jest pokazanie, że dane zdanie *S* jest konsekwencją zbioru wskazanych aksjomatów, przy zastosowaniu podanych reguł. Przykładowym zastosowaniem jest formalna weryfikacja poprawności programów.

Podstawowym problemem w dowodzeniu twierdzeń jest duża złożoność czasowa wynikająca w wykładniczego wzrostu przestrzeni zdań. Dlatego narzuca się wykorzystanie przetwarzania równoległego. Przedstawione zostanie na przykładzie wnioskowania ‘wstecz’ (ang. *backward reasoning*). W metodzie tej mamy do czynienia z tezą *T*, której prawdziwości nie znamy. Należy pokazać, czy – i w jaki sposób – teza ta sprowadza się do zdań prawdziwych (podanych), sprowadzając *T(T1,...,Tn)* do tez *T1,...,Tn* i rekurencyjnie tezy *Ti* do kolejnych, aż do uzyskania zdań, których prawdziwość potrafimy ocenić.

Kontrast dla metody wnioskowania ‘wstecz’ stanowi wnioskowanie ‘w przód’, w którym wychodząc ze zdań prawdziwych, próbujemy dojść do wskazanej tezy.

Ogólne rozwiązanie

Przyjmijmy dla uproszczenia, że zdania składają się z formuł łączonych operatorami *OR* i *AND*. Wystąpienie węzeła *N* z operatorem *OR* w drzewie dowodowym wskazuje, że wystarczy skonstruować dowód dla jednej gałęzi, *AND* – że dla wszystkich, jeśli bowiem nie uda się tego zrobić, to dowód zdania odpowiadającego *N* nie istnieje i poddrzewo o korzeniu *N* można odciąć z drzewa dowodowego.

Przeszukiwanie drzewa w celu znalezienia dowodu można wykonywać równolegle. Mamy wówczas gwarancję, że część procesów podąża w 'dobrym' kierunku. Jeżeli konieczne jest obcięcie, to być może część procesów trzeba zatrzymać zwalniając zasoby.

Strategia przydziału procesów do zadań przeszukiwania drzewa może być różna i należy ją dostosować do semantyki operatorów (relacji).

8.3.3 Wyszukiwanie rozproszonych informacji

Definicja problemu

Pojemność informacyjna dużej sieci jest ogromna, a wraz z jej wzrostem mogą powiększać się trudności w dostępie do informacji, o ile nie powstanie stosowny mechanizm umożliwiający odnajdywanie potrzebnych danych (np. mechanizm zapytań). Dane te mają z reguły postać tekstową lub graficzną.

Przykładem systemu wyszukującego informację w systemie rozproszonym jest *WAIS*. Jest to system klient-serwer, któremu użytkownik podaje listę serwerów, słowa kluczowe i odniesienia do innych obiektów.

Realizacja rozwiązania w systemie DPE

Przeszukiwanie całej sieci jest zwykle niewydajne, a stworzenie jednego katalogu wszystkich danych nieużyteczne praktycznie. Można więc skorzystać z metainformacji, czyli stworzyć indeksy. Indeksy zawierają słowa kluczowe i wszystkie inne informacje potrzebne algorytmom implementującym mechanizm zapytań (tj. częstości wystąpień, itp.) – same dane nie są przeglądane. Indeksy umożliwiają szybsze wyszukiwanie, ale traci się informację o obiekcie. Tworzenie indeksów możemy zlecić ludziom (tzw. etykiety) lub możemy je generować automatycznie.

Odnajdywane obiekty mogą mniej lub bardziej odpowiadać warunkom zapytania (wyszukiwanie rozmyte). Określa się wówczas funkcję podobienstwa (dla danych tekstowych i graficznych). Użytkownicy wolą zadawać pytania w sposób niejednoznaczny, najchętniej w języku naturalnym. Wtedy potrzebne są słowniki i tezaurusy – te ostatnie przechowują informację o synonimach i 'stopniu zgodności' i budowane są często automatycznie przy pomocy metod statystycznych analizy lingwistycznej. Natomiast słowniki informują o znaczeniu.

Dobrze jest podzielić sieć na odrębne, spójne segmenty – czyli zastosować strategię "dziel i rządź". Każdy węzeł (który może obsługiwać segment) powinien udostępniać stosowną usługę (niech usługa nazywa się *infosearch*), która potrafiłaby skorzystać z informacji indeksowej węzłów w segmencie. Podział na segmenty można wykonać następująco: należy wprowadzić na każdym węźle nowy parametr (jako zasób) *INFOSEG* i przypisać mu wartość numeru segmentu. W ten sposób, jeżeli system *DPE* będzie przeszukiwał sieć, może on uwzględniać tylko węzły wskazanego segmentu *s*, jeśli w warunku globalnym umieścimy wyrażenie *INFOSEG* = *s*, które przyjmuje wartość *0* tylko wtedy, gdy węzeł nie należy do segmentu *s*. Ponieważ *s* może przyjmować tyle wartości, ile jest segmentów, należy na węźle z którego kierujemy zapytanie zarejestrować usługę *infosearch* tyle razy, z taką liczbą nazw i z taką liczbą warunków globalnych, ile jest segmentów.

Wybór węzła przeszukującego segment warto dostosować do obciążenia sieci i rozmieszczenia danych. Dlatego należy odpowiednio dobrać wyrażenie określające wymagania usługi *infosearch* wobec zasobów węzła (maksymalizowane), np.:

*-x * CPUUTIL + y * INDEXED*

gdzie *x* to współczynnik określający znaczenie obciążenia procesora, a *y* określa znaczenie wielkości danych (*INDEXED*, np. w megabajtach). Węzły o większej wielkości danych i mniejszym obciążeniu procesora będą faworyzowane.

Przekazywanie danych najlepiej zrealizować przy pomocy standardowego wejścia i wyjścia. Na wejściu usługa otrzymywałaby zapytanie, na wyjściu zwracałaby wyniki (ścieżki dostępu do danych spełniających warunki zapytania).

8.3.4 Rozproszone bazy danych

Definicja problemu

Bazy danych zawsze były ważnymi aplikacjami w każdym środowisku komputerowym.

W systemie rozproszonym możemy mieć do czynienia z bazami danych w następujących postaciach:

- wielodostępna baza danych;
- baza z danymi rozproszonymi w sieci;
- wiele baz danych funkcjonujących jako całość;
- rozproszona równoległa baza danych;
- repozytorium rozproszonych, migrujących obiektów.

W związku z tym mogą pojawić się następujące problemy implementacyjne:

- uogólnione zapytania dotyczące rozproszonej informacji (definicja języka zapytań, optymalizacja rozproszonych zapytań);
- optymalizacja rozkładu obciążenia w sieci (obciążenie przetwarzaniem i danymi).

Rozproszone dane spełniać powinny następujące warunki:

- lokalna autonomia – dzięki niej dane mogą być lokalnie chronione i zarządzane;
- decentralizacja, czyli brak centralnych zbiorów danych;
- niezależność mechanizmów dostępu od lokalizacji danych;
- istnienie mechanizmów replikacji danych;
- mechanizm rozproszonych transakcji, gwarantujący spójność rozproszonych danych;
- niezależność reprezentacji od platformy.

Obecnie największe nadzieje wiąże się z rozproszonymi równoległymi bazami danych i repozytoriami obiektów. System *DPE* można wykorzystać do implementacji obu środowisk. Zajmiemy się równoległymi bazami danych, ponieważ rozproszone obiekty mają liczne implementacje i bogatszą literaturę.

Rozwiązanie w systemie DPE

Wydaje się, że należy zdecydować się na relacyjny model bazy danych:

- jest to model najbardziej powszechny;
- opracowano standard języka zapytań: *SQL*;
- operacje relacyjne (projekcja, selekcja, złączenie, unia) można zrównoleglić.

Równoległa baza danych powinna być oparta na wielu serwerach relacyjnych baz danych, pracujących na wielu węzłach. Serwery powinny używać tego samego języka zapytań (najlepiej ten sam dialekt *SQL*).

Przyjmując model obliczeń gwiazdy z węzłami powiązanymi w pierścień możemy stworzyć system działający w następujący sposób:

1. Węzeł nadrzędny będzie przyjmował i interpretował zapytania od użytkownika (program *pardbsql*).

2. Przetworzone zapytanie (lub zapytania) należy przesłać do wszyskich węzłów/serwerów (poprzez zdalne wywołanie usługi *pardbserver*).
3. Zapytania zostaną odebrane przez procesy podrzędne, które przekażą je (być może po modyfikacjach) serwerom baz danych (te je przetworzą i zwrócą wyniki), a procesy współpracując ze sobą (przekazując dane przez rozproszone pliki) stworzą zbiór wynikowy.
4. Zbiór wynikowy należy przekazać węzłowi nadrzędnemu.

Ponieważ serwery baz danych znajdują się na określonych węzłach, nie warto wykorzystywać mechanizmu równoważenia obciążenia (*DPE*), ponieważ zbiory wynikowe od serwerów mogą być duże co spowodowałoby obciążenie sieci.

Wyjaśnienia wymaga punkt 3. Rozpatrzmy typowe zapytanie:

SELECT X.A,Y.B FROM X,Y WHERE X.K=Y.K. (Q)

Jeżeli tabela *X* jest przechowywana na węźle *N1*, a tabela *Y* na węźle *N2*, to po otrzymaniu zapytania procesy *pardbserver* przekażą serwerom baz danych zapytanie z żądaniem dokonania projekcji:

- wêze³ *N1*: *SELECT X.A,X.K FROM X*; (Q1)
- wêze³ *N2*: *SELECT Y.B,Y.K FROM Y*. (Q2)

Złączenie (*WHERE X.K=Y.K*) musi być przeprowadzone przez wykorzystanie tych wyników – np. przez węzeł *N1*, na którym zakłada się tymczasową tabelę *Y* z danymi z *N2* i wykonuje zapytanie (Q). Jeśli jednak zbiór wynikowy (Q1) byłby mniejszy od (Q2), to lepiej założyć tablicę tymczasową na *N2.*

Niebanalnym problemem implementacyjnym jest więc przetwarzanie zapytań *SQL* na postać wielu zapytań do równoległego wykonania.

9 PODSUMOWANIE

9.1 Wnioski

System przetwarzania rozproszonego może być zaawansowany technologicznie, wiele problemów mogą stwarzać kwestie techniczne, trudniejsze jest uzyskanie przenośności. Bardzo duże trudności sprawia testowanie i wszelkie inne próby oceny poprawności kodu.

Metodyka *DPE* w przypadku systemu *DPE/UNIX* pozwalała przede wszystkim na systematyzację pracy. Podejście metodyczne zawsze wprowadza pewien narzut, który w tym przypadku był proporcjonalnie duży. Praca autora miała charakter badawczy. W przypadku dużego rozproszonego systemu czasu rzeczywistego, np. koordynującego działanie sygnalizacji świetlnej w aglomeracji, użycie metodologii autor uważa za niezbędne.

9.2 Rozwój systemu – wskazówki

9.2.1 Oprogramowanie narzędziowe

Śledzenie i alarmy

Do słabszych punktów systemu *DPE* należą mechanizmy bezpieczeństwa i ochrony danych. Jednym z elementów systemu bezpieczeństwa są systemy śledzenia (ang. *auditing system*), które szczegółowo rejestrują czynności użytkowników, pozwalając na wykrycie szkodliwej działalności – czy to celowej, czy niezamierzonej. Ważnym elementem są również alarmy ostrzegające administratorów o wystąpieniu zdarzeń lub zajściu ustalonych warunków – do takich zdarzeń należeć mogłyby w systemie *DPE*: wykonanie usługi, zatrzymanie demona, przesłanie sygnału do demona, otwarcie zdalnego pliku.

Warto rozważyć zaimplementowanie tych elementów w postaci oprogramowania narzędziowego.

Monitorowanie

Monitorowanie systemu mogłoby obejmować obserwację następujących działań i pomiar parametrów:

- obciążenia procesorów, wykorzystania pamięci, czy ogólniej, zasobów węzłów – np. w celu strojenia wymagań (warunki w opisie usług), rozmieszczenia usług lub optymalizacji kodu usług;
- dostępu do zdalnych plików – w celu lepszego rozmieszczenia danych w sieci;
- aktywności użytkowników.

Dystrybucja usług

Aktualnie instalacja usługi na wielu węzłach może być rzeczą kłopotliwą, ponieważ uprawniony jest do tego jedynie administrator systemu (ang. *root*), ponadto należy przenieść i skompilować kod źródłowy. Kompromisem pomiędzy łatwością implementacji a wygodą użytkowania byłoby narzędzie automatyzujące te czynności.

9.2.2 Równoważenie obciążenia

Równoważenie obciążenia w systemie *DPE* odbywa się metodą zachłanną – przez wybór najlepszego węzła dla kolejnego zadania. Gdyby żądania grupować, byłaby możliwa optymalizacja rozmieszczania całych grup wywołań.

10 DODATKI

10.1 Definicja metodyki

10.1.1 Struktura zadań

Motodyka przygotowana do zaprojektowania i realizacji systemu *DPE/UNIX* przedstawia się w skrócie następująco:

Etapy i kroki		Produkty wejściowe	Produkty wynikowe
1	Analiza wymagań		Katalog wymagań
1.1	Określ użytkowników systemu.		Katalog użytkowników
1.2	Zdefiniuj wymagania jakościowe dotyczące niezależności od platformy, własności elementów przetwarzających, przezroczystości, bezpieczeństwa i ochrony, sytuacji awaryjnych oraz związane z użytecznością.	Katalog użytkowników	Katalog wymagań jakościowych
1.3	Zdefiniuj wymagania ilościowe dotyczące wydajności oraz skalowalności.	Katalog użytkowników	Katalog wymagań ilościowych
2	Specyfikacja logiczna	Katalog wymagań	Projekt logiczny systemu
2.1	Zdefiniuj moduły odpowiedzialne za rozproszone przetwarzanie i obsługujące rozproszone dane.	Katalog wymagań	Specyfikacja logiczna modułów
2.2	Zdefiniuj schematy komunikacji związane z rozproszonym przetwarzaniem i z operowaniem na rozproszonych danych.	Katalog wymagań	Specyfikacja logiczna schematów komunikacji
2.3	Zdefiniuj narzędzia odpowiedzialne za lokalne i zdalne zarządzanie elementami systemu.	Katalog wymagań	Specyfikacja logiczna narzędzi
2.4	Zdefiniuj algorytmy konieczne do realizacji modułów, schematów komunikacji oraz narzędzi.	Katalog wymagań, Specyfikacja logiczna modułów, Specyfikacja logiczna schematów komunikacji, Specyfikacja logiczna narzędzi	Definicje algorytmów
3	Projekt fizyczny	Katalog wymagań, Projekt logiczny systemu	Projekt fizyczny systemu, Specyfikacja testów
3.1	Określ platformę sprzętową i programową.	Katalog wymagań	Specyfikacja platformy
3.2	Zdefiniuj demony (i ich testy) jako realizacje modułów.	Specyfikacja logiczna modułów	Specyfikacja demonów, Specyfikacja testów demonów
3.3	Zdefiniuj protokoły (i ich testy) jako realizacje schematów komunikacji.	Specyfikacja logiczna schematów komunikacji	Specyfikacja protokołów, Specyfikacja testów protokołów
3.4	Zdefiniuj programy narzędziowe	Specyfikacja logiczna	Specyfikacja programów

	(i ich testy) jako realizacje narzędzi.	narzędzi	narzędziowych, Specyfikacja testów programów narzędziowych
3.5	Zdefiniuj bibliotekę funkcji przetwarzania rozproszonego (i testy tych funkcji).	Specyfikacja platformy, Specyfikacja demonów, Specyfikacja protokołów, Specyfikacja programów narzędziowych	Specyfikacja biblioteki funkcji przetwarzania rozproszonego, Specyfikacja testów biblioteki funkcji przetwarzania rozproszonego
3.6	Zdefiniuj bibliotekę funkcji nieprzenośnych (i testy tych funkcji).	Specyfikacja platformy, Specyfikacja demonów, Specyfikacja protokołów, Specyfikacja programów narzędziowych	Specyfikacja biblioteki funkcji nieprzenośnych, Specyfikacja testów biblioteki funkcji nieprzenośnych
4	Implementacja	Specyfikacja platformy, Projekt fizyczny systemu, Definicje algorytmów, Specyfikacja biblioteki funkcji przetwarzania rozproszonego, Specyfikacja biblioteki funkcji nieprzenośnych	Kod systemu, Opis realizacji systemu, Dokumentacja systemu
4.1	Zaimplementuj i opisz realizację demonów odpowiedzialnych za rozproszone przetwarzanie i obsługujących rozproszone dane.	Specyfikacja demonów Definicje algorytmów	Kod demonów, Opis realizacji demonów, Dokumentacja demonów
4.2	Zaimplementuj i opisz realizację protokołów związanych z rozproszonym przetwarzaniem i z operowaniem na rozproszonych danych.	Specyfikacja protokołów, Definicje algorytmów	Kod protokołów, Opis realizacji protokołów, Dokumentacja protokołów
4.3	Zaimplementuj i opisz realizację programów narzędziowych odpowiedzialnych za lokalne i zdalne zarządzanie elementami systemu.	Specyfikacja programów narzędziowych, Definicje algorytmów	Kod programów narzędziowych, Opis realizacji programów narzędziowych, Dokumentacja programów narzędziowych
4.4	Zaimplementuj i opisz realizację biblioteki funkcji przetwarzania rozproszonego	Specyfikacja biblioteki funkcji przetwarzania rozproszonego	Kod biblioteki funkcji przetwarzania rozproszonego, Opis realizacji biblioteki funkcji przetwarzania rozproszonego
4.5	Zaimplementuj i opisz realizację biblioteki funkcji nieprzenośnych	Specyfikacja biblioteki funkcji nieprzenośnych	Kod biblioteki funkcji nieprzenośnych, Opis realizacji biblioteki funkcji nieprzenośnych
5	Testy	Specyfikacja platformy, Projekt testów Kod systemu, Specyfikacja testów	Przetestowany kod systemu, Katalog błędów
5.1	Zdefiniuj testy integralności systemu.	Kod systemu, Dokumentacja systemu, Specyfikacja testów	Specyfikacja testów integralności

5.2	Przetestuj system.	Kod systemu, Dokumentacja systemu, Specyfikacja testów, Specyfikacja testów integralności	Przetestowany kod systemu, Katalog błędów
6	Korekta systemu	Przetestowany kod systemu, Katalog błędów	Poprawne produkty i kod systemu
6.1	Przejrzyj produkty analizy wymagań, projektu logicznego, projektu fizycznego i implementacji, napraw znalezione błędów usterki.	Przetestowany kod systemu, Katalog	Poprawne produkty i kod systemu

10.1.2 Struktura produktów

Z poprzedniego punktu wynika następująca hierarchia dokumentacji projektowej *PBS* (*Product Breakdown Structure*):

Produkty

Katalog użytkowników
Katalog wymagań
- Katalog wymagań jakościowych
  - Katalog wymagań dotyczących niezależności od platformy
  - Katalog wymagań dotyczących własności elementów przetwarzających
  - Katalog wymagań dotyczących przezroczystości
  - Katalog wymagań dotyczących bezpieczeństwa i ochrony
  - Katalog wymagań dotyczących sytuacji awaryjnych
  - Katalog wymagań związanych z użytecznością
- Katalog wymagań ilościowych
  - Katalog wymagań dotyczących wydajności
  - Katalog wymagań dotyczących skalowalności

Projekt logiczny systemu
- Specyfikacja logiczna modułów
- Specyfikacja logiczna schematów komunikacji
- Specyfikacja logiczna narzędzi
- Definicja algorytmów

Projekt fizyczny systemu
- Specyfikacja platformy
- Specyfikacja demonów
- Specyfikacja protokołów
- Specyfikacja programów narzędziowych
- Specyfikacja biblioteki funkcji przetwarzania rozproszonego
- Specyfikacja biblioteki funkcji nieprzenośnych
- Specyfikacja testów
  - Specyfikacja testów demonów
  - Specyfikacja testów protokołów
  - Specyfikacja testów programów narzędziowych
  - Specyfikacja testów biblioteki funkcji przetwarzania rozproszonego
  - Specyfikacja testów biblioteki funkcji nieprzenośnych

Opis realizacji systemu
- Opis realizacji demonów
- Opis realizacji protokołów
- Opis realizacji programów narzędziowych
- Opis realizacji biblioteki funkcji przetwarzania rozproszonego
- Opis realizacji biblioteki funkcji nieprzenośnych

Dokumentacja systemu
- Dokumentacja demonów
- Dokumentacja protokołów
- Dokumentacja programów narzędziowych
- Dokumentacja biblioteki funkcji przetwarzania rozproszonego
- Dokumentacja biblioteki funkcji nieprzenośnych

Specyfikacja testów integralności

Katalog błędów

10.1.3 Definicje produktów

Opiszemy teraz dokładniej zawartość katalogów produktów projektowych. Każdy element każdego katalogu ma określoną strukturę – opis zawiera co najmniej identyfikator (*ID*) i nazwę elementu, opcjonalne uwagi i pole **Pochodzenie** (z wyjątkiem opisów użytkowników), którego zawartość będąca zawsze listą identyfikatorów-odnośników określa z jakich specyfikacji projektowych korzysta się przy tworzeniu bieżącej specyfikacji.

Katalog użytkowników

Katalog użytkowników składa się z opisów użytkowników. Każdy opis ma następującą strukturę:

ID: <identyfikator użytkownika>
Nazwa: <nazwa użytkownika>
Uprawnienia: <lista uprawnień, np. *Zdalne wykonywanie usług*>
Rola użytkownika: <funkcja pełniona przez użytkownika w systemie>
Uwagi: <uwagi dotyczące użytkownika>

Katalog wymagań

Katalog wymagań składa się z definicji wymagań o następującej strukturze:

ID: <identyfikator wymagania>
Nazwa: <nazwa wymagania>
Pochodzenie: <lista identyfikatorów użytkowników>
Dziedziny: <lista nazw dziedzin, np. *Przezroczystość replikacji*>
Typ: Jakościowe | Ilościowe
Priorytet: Zerowy | Niski | Średni | Wysoki | Bardzo wysoki
Treść wymagania: <dokładny opis wymagania>
Uwagi: <uwagi dotyczące wymagania>

Specyfikacja logiczna modułów

Specyfikacja logiczna modułów składa się z definicji modułów o następującej strukturze:

ID: <identyfikator modułu>
Nazwa: <nazwa modułu>
Pochodzenie: <lista identyfikatorów wymagań>
Funkcja: <przeznaczenie modułu>
Interfejs modułu: <definicja interfejsu modułu – składnia i semantyka>
Uwagi: <uwagi dotyczące modułu>

Specyfikacja logiczna schematów komunikacji

Specyfikacja logiczna schematów komunikacji jest złożona z definicji tych schematów na poziomie logicznym. Każda definicja ma następującą strukturę:

ID: <identyfikator schematu komunikacji>
Nazwa: <nazwa schematu>
Pochodzenie: <lista identyfikatorów wymagań>
Funkcja: <przeznaczenie schematu>
Definicja schematu: <opis definiujący schemat komunikacji>
Uwagi: <uwagi dotyczące schematu>

Specyfikacja logiczna narzędzi

Specyfikacja logiczna narzędzi składa się z definicji tych narzędzi. Definicje posiadają następującą strukturę:

ID: <identyfikator narzędzia>
Nazwa: <nazwa narzędzia>
Pochodzenie: <lista identyfikatorów wymagań>
Funkcja: <przeznaczenie narzędzia>
Opis użycia narzędzia: <opis korzystania z narzędzia>
Uwagi: <uwagi dotyczące narzędzia>

Definicja algorytmów

Definicja algorytmów składa się z opisów treści tych algorytmów. Każdy opis ma następującą strukturę:

ID: <identyfikator algorytmu>
Nazwa: <nazwa algorytmu>
Pochodzenie: <lista identyfikatorów wymagań, lista identyfikatorów modułów, lista identyfikatorów schematów komunikacji, lista identyfikatorów narzędzi>
Cel: <cel przetwarzania realizowanego przez algorytm>
Wejście: <dane wejściowe wykorzystywane przez algorytm>
Wyjście: <dane wynikowe produkowane przez algorytm>
Treść algorytmu: <opis przetwarzania realizowanego przez algorytm>
Uwagi: <uwagi dotyczące algorytmu>

Specyfikacja platformy

Specyfikacja platformy ma następującą strukturę:

ID: <identyfikator platformy>
Nazwa: <nazwa platformy>
Pochodzenie: <lista identyfikatorów wymagań>
Opis platformy: <opis platformy>
Uwagi: <uwagi dotyczące platoformy>

Specyfikacja demonów

Specyfikacja demonów składa się z definicji *interface*'ów tych demonów. Każdy opis ma następującą strukturę:

ID: <identyfikator demona>
Nazwa: <nazwa demona>
Pochodzenie: <identyfikator platformy, identyfikator modułu>
Funkcja: <przeznaczenie demona>
Interfejs demona: <definicja interfejsu demona – składnia i semantyka>
Uwagi: <uwagi dotyczące demona>

Specyfikacja testów demonów

Specyfikacja testów demonów składa się z opisów scenariuszy testów. Opis scenariusza ma następującą strukturę:

ID: <identyfikator demona>
Nazwa: <nazwa demona>
Pochodzenie: <identyfikator platformy, identyfikator demona>
Scenariusz testu demona: <opis scenariusza testu demona>
Uwagi: <uwagi dotyczące tesu>

Specyfikacja protokołów

Specyfikacja protokołów składa się z definicji tych protokołów. Każda ma następującą strukturę:

ID: <identyfikator protokołu>
Nazwa: <nazwa protokołu>
Pochodzenie: <identyfikator platformy, identyfikator schematu komunikacji>
Funkcja: <przeznaczenie protokołu>
Definicja protokołu: <opis definiujący protokół>
Uwagi: <uwagi dotyczące protokołu>

Specyfikacja testów protokołów

Specyfikacja testów protokołów składa się z opisów scenariuszy testów. Opis scenariusza ma następującą strukturę:

ID: <identyfikator protokołu>
Nazwa: <nazwa protokołu>
Pochodzenie: <identyfikator platformy, identyfikator protokołu>
Scenariusz testu protokołu: <opis scenariusza testu protokołu>
Uwagi: <uwagi dotyczące testu>

Specyfikacja programów narzędziowych

Specyfikacja programów narzędziowych składa się z definicji tych programów. Każda definicja programu narzędziowego posiada następującą strukturę:

ID: <identyfikator programu>
Nazwa: <nazwa programu>
Pochodzenie: <identyfikator platformy, identyfikator narzędzia>
Funkcja: <przeznaczenie programu>
Opis użycia programu: <opis sposobu korzystania z programu>
Uwagi: <uwagi dotyczące programu>

Specyfikacja testów programów narzędziowych

Specyfikacja testów programów narzędziowych składa się z opisów scenariuszy testów. Opis scenariusza ma następującą strukturę:

ID: <identyfikator programu>
Nazwa: <nazwa programu>
Pochodzenie: <identyfikator platformy, identyfikator programu>
Scenariusz testu programu: <opis scenariusza testu programu>
Uwagi: <uwagi dotyczące testu>

Specyfikacja biblioteki funkcji przetwarzania rozproszonego

Specyfikacja biblioteki funkcji przetwarzania rozproszonego składa się z definicji funkcji tej biblioteki. Definicja funkcji ma następującą strukturę:

ID: <identyfikator funkcji>
Nazwa: <nazwa funkcji>
Pochodzenie: <identyfikator platformy, lista identyfikatorów demonów, lista identyfikatorów protokołów, lista identyfikatorów programów>
Parametry: <opis parametrów>
Wynik: <opis wartości wynikowej>
Definicja funkcji: <definicja funkcji>
Uwagi: <uwagi dotyczące funkcji>

Specyfikacja testów biblioteki funkcji przetwarzania rozproszonego

Specyfikacja testów biblioteki funkcji przetwarzania rozproszonego składa się z opisów scenariuszy testów – mają one następującą strukturę:

ID: <identyfikator funkcji>
Nazwa: <nazwa funkcji>
Pochodzenie: <identyfikator platformy, identyfikator funkcji przetwarzania rozproszonego>
Scenariusz testu funkcji: <opis scenariusza testu funkcji>
Uwagi: <uwagi dotyczące testu>

Specyfikacja biblioteki funkcji nieprzenośnych

Specyfikacja biblioteki funkcji nieprzenośnych składa się z definicji funkcji tej biblioteki. Definicja funkcji ma następującą strukturę:

ID: <identyfikator funkcji>
Nazwa: <nazwa funkcji>
Pochodzenie: <identyfikator platformy, lista identyfikatorów demonów, lista identyfikatorów protokołów, lista identyfikatorów programów, lista identyfikatorów funkcji przetwarzania rozproszonego>
Parametry: <opis parametrów>
Wynik: <opis wartości wynikowej>
Definicja funkcji: <definicja funkcji>
Uwagi: <uwagi dotyczące funkcji>

Specyfikacja testów biblioteki funkcji nieprzenośnych

Specyfikacja testów biblioteki funkcji nieprzenośnych składa się z opisów scenariuszy testów – mają one następującą strukturę:

ID: <identyfikator funkcji>
Nazwa: <nazwa funkcji>
Pochodzenie: <identyfikator platformy, identyfikator funkcji nieprzenośnej>
Scenariusz testu funkcji: <opis scenariusza testu funkcji>
Uwagi: <uwagi dotyczące testu>

Opis realizacji demonów

Opis realizacji demonów składa się z elementów o następującej strukturze:

ID: <identyfikator demona>
Nazwa: <nazwa demona>
Pochodzenie: <identyfikator demona, lista identyfikatorów algorytmów>
Opis implementacji demona: <opis implementacji demona>
Uwagi: <uwagi dotyczące modułu>

Opis realizacji protokołów

Opis realizacji protokołów składa się z elementów o następującej strukturze:

ID: <identyfikator protokołu>
Nazwa: <nazwa protokołu>
Pochodzenie: <identyfikator protokołu, lista identyfikatorów algorytmów>
Opis implementacji protokołu: <opis implementacji protokołu>
Uwagi: <uwagi dotyczące protokołu>

Opis realizacji programów narzędziowych

Opis realizacji programów narzędziowych składa się z elementów o następującej strukturze:

ID:<identyfikator programu>
Nazwa:<nazwa programu>
Pochodzenie: <identyfikator programu, lista identyfikatorów algorytmów>
Opis implementacji programu: <opis implementacji programu>
Uwagi: <uwagi dotyczące programu>

Opis realizacji biblioteki funkcji przetwarzania rozproszonego

Opis realizacji biblioteki funkcji przetwarzania rozproszonego składa się z opisów funkcji tej biblioteki. Opis funkcji ma następującą strukturę:

ID: <identyfikator funkcji>
Nazwa: <nazwa funkcji>
Pochodzenie: <identyfikator funkcji przetwarzania rozproszonego, lista identyfikatorów algorytmów>
Opis implementacji funkcji: <opis implementacji funkcji>
Uwagi: <uwagi dotyczące funkcji>

Opis realizacji biblioteki funkcji nieprzenośnych

Opis realizacji biblioteki funkcji nieprzenośnych składa się z opisów funkcji tej biblioteki. Opis funkcji nieprzenośnej ma następującą strukturę:

ID: <identyfikator funkcji>
Nazwa: <nazwa funkcji>
Pochodzenie: <identyfikator funkcji nieprzenośnej>
Opis implementacji funkcji: <opis implementacji funkcji>
Uwagi: <uwagi dotyczące funkcji>

Katalog błędów

Raport o błędach jest złożony z opisów błędów dostrzeżonych podczas testowania systemu. Opis błędu posiada następującą strukturę:

ID:<identyfikator błędu>
Nazwa:<nazwa błędu>

Pochodzenie: <lista identyfikatorów testów>
Opis błędu: <opis sytuacji w której pojawia się błąd oraz objawów błędu>
Uwagi: <uwagi dotyczące błędu>

10.2 Podręcznik administratora

10.2.1 Instalacja systemu

Instalacja systemu na każdym węźle musi być przeprowadzona przez administratora, tj. użytkownika z uprawnieniami *root*.

Instalacja składa się z następujących etapów:

1. Należy stworzyć katalog przeznaczony na kod źródłowy systemu *DPE/UNIX*, np. */usr/dpe* i skopiować do niego plik *dpe.tar*: *mkdir /usr/dpe;cp dpe.tar /usr/dpe;cd /usr/dpe*.
2. Należy rozpakować plik *dpe.tar* przy pomocy programu *tar*: *tar xf dpe.tar;rm dpe.tar*.
3. Otrzymany kod źródłowy należy skompilować używając pliku *makefile*: *make;make clean*.
4. Programy binarne i skrypty *sh* należy przenieść do katalogu */usr/bin*: *mv dpestart dpestop dperun dpebest dpestub dpetest dpeinfod dpeexecd dpefsysd dpeseld dpesvcsd /usr/bin*.
5. Pliki konfiguracyjne demonów należy uzupełnić, a następnie przenieść do katalogu */etc*: *mv *.conf /etc*.

10.2.2 Uruchamianie systemu

System rozpoczyna pracę przez uruchomienie jego demonów. Może robić to każdorazowo administrator, ale wygodniej jest umieścić odpowiednią deklarację w pliku */etc/services* [4].

10.2.3 Pielęgnacja systemu

Wprowadzanie nowych usług polega na dodaniu informacji o usłudze w pliku konfiguracyjnym demona bazy usług, tj. */etc/dpesvcsd.conf*.

Zmiana struktury sieci może wymagać zmiany w pliku */etc/dpeseld.conf*, ponieważ pewne węzły mogą przestać być dostępne. Pojawienie się nowego węzła może zostać w tym pliku zarejestrowane.

Informacje o nowych zasobach węzła, np. dodana pamięć, większy dysk, należy zanotować w pliku */etc/dpeinfod.conf*.

10.3 Podręcznik użytkownika

10.3.1 Korzystanie z systemu

Uruchomienie usługi odbywa się przy pomocy skryptu *dperun*. Należy podać nazwę usługi.

Pośrednie zadanie, mianowicie znalezienie najlepszego węzła, wykonuje skrypt *dpebest*.

Program *dpeclient* umożliwia połączenie się z każdym demonem *DPE* na dowolnym węźle i zlecanie jemu komend do wykonania.

10.3.2 Programowanie

Programista tworzy rozproszone programy przy pomocy trzech funkcji:

- funkcja *dpeexecve*: przypomina funkcję *execve*, jednak bieżący proces jest zastępowany procesem *dpestub*, który współpracuje z procesem na odległym węźle (program tego procesu powinien być zarejestrowany jako usługa na lokalnym węźle);
- funkcja *dpeopen*: przypomina funkcję *open*, ale otwierany plik może znajdować się na odległym węźle (musi się zgadzać identyfikator liczbowy użytkownika); jeżeli tak jest, w katalogu */tmp* tworzona jest replika tego pliku i na niej operuje program;
- funkcja *dpeclose*: zamyka plik otwarty przez *dpeopen*; replika zostaje odesłana.

11 BIBLIOGRAFIA

[1] "*A Distributed Load-balancing Policy for a Multicomputer"*, Amnon Barak, Amnon Shiloh, *Software – Practice and Experience*, September 1985;
[2] "*Disconnected Operation in a Distributed File System*", James Jay Kistler, *Springer*, 1996;
[3] "*Distributed Operating Systems – The Logical Design*", Andrzej Goscinski, *Addison-Wesley Publishing Company*, 1991;
[4] "*Internet System Handbook*", Daniel C. Lynch, Marshall T. Rose, *Addison-Wesley Publishing Company*, 1993;
[5] "*Intoduction to SSADM Version 4*", Caroline Ashworth, *McGRAW-HILL Book Company Europe*, 1993;
[6] *LBMS Systems Engineer* – dokumentacja;
[7] "*Load Distributing for Locally Distributed Systems*", Niranjan G. Shivaratri, Philip Krueger, Mukesh Singhal, *COMPUTER*, December 1992;
[8] *LSF* – dokumentacja, m.in.: "*LSF: Load Sharing in Large Heterogeneous Distributed Systems*", Songnian Zhou, *Platform Computing Corporation*, 1995;
[9] "*Network and Distributed Systems Management*", Morris Slowman, *Addison-Wesley Publishing Company*, 1994;
[10] "*Podstawy systemów operacyjnych*", Abraham Silberschatz, James L. Peterson, Peter B. Galvin, *WNT*, 1993;
[11] "*Programowamie w systemie Unix dla zaawansowanych*", Marc J. Rochkind, *WNT*, 1993;
[12] "*Programowanie współbieżne i rozproszone – w przykładach i zadaniach*", Zbigniew Weiss, Tadeusz *Gruźlewski*, *WNT*, 1993;
[13] "*Programowamie zastosowań sieciowych w systemie Unix*", W. Richard Stevens, *WNT*, 1995;
[14] "*PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*", Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Suneram, *The MIT Press*, 1994;
[15] "*Strategie klient-serwer – Systemy informatyczne w procesie przekształcania przedsiębiorstwa*", David Vaskevitch, *IDG Poland S.A.*, 1995;
[16] "*Systemy rozproszone – wykład*", Zbigniew Weiss, *WMIiM UW*, 1993;
[17] "*The MOSIX Distributed Operating System – Load balancing for UNIX*", Amonon Barak, Shai Guday, Richard G. Wheeler, *Springer-Verlag*, 1993;
[18] "*The UNIX operating system*", Kaare Christian, Susan Richter, *John Wiley & Sons Inc.*, 1994
[19] "*Wielka encyklopedia sieci*", Tom Sheldon, *Wydawnictwo Robomatic*, 1995.
[20] "*Zrównoleglanie obliczeń*", Andrzej Góralski, *UNIXforum*, 4'97