Wirtualizacja rozproszonej pamięci operacyjnej multikomputera dla systemu Linux w oparciu o koncepcję SDDS

Rozprawa doktorska

Autor:
mgr inż. Arkadiusz Chrobot

Promotor:
prof. dr hab. inż. Krzysztof Sapiecha

9 września 2010

Spis treści

1. Wstęp

Wydajność aplikacji użytkowych zależy od tego jakie zasoby zostały zgromadzone w systemie komputerowym oraz w jaki sposób są one zarządzane. Wzrost zapotrzebowania aplikacji na zasoby jest wprost proporcjonalny do rozmiaru danych, które ta aplikacja musi przetworzyć [1]. Problemy dotyczące zarządzania takimi zasobami jak czas procesora, pamięć operacyjna i pamięć masowa są dobrze znane i stanowią przedmiot wielu badań i publikacji (prace [2–4] zawierają wykaz podstawowych pozycji). Problem niedostatku zasobów może być rozwiązany poprzez połączenie pojedynczych systemów komputerowych (jedno lub wieloprocesorowych) i utworzenie w ten sposób systemów wielokomputerowych.

1.1. Systemy wielokomputerowe

Systemy wielokomputerowe są typem systemów wieloprocesorowych. Według klasyfikacji przedstawionej w [2] komputery z wieloma procesorami można podzielić na trzy kategorie:

- **wieloprocesory ze wspólną pamięcią** nazywane często po prostu **systemami wieloprocesorowymi** lub **wieloprocesorami** (ang. *multiprocessors*) - są to systemy, w których wszystkie procesory mają wspólną przestrzeń adresową, a więc modyfikacja zawartości wspólnej pamięci dokonana przez jeden z nich jest widoczna dla pozostałych,
- **wielokomputery**[1] lub **multikomputery**[1] (ang. *multicomputers*) - systemy wieloprocesorowe złożone z pojedynczych komputerów, które nie mają wspólnej przestrzeni adresowej, a wymiana informacji między nimi dokonywana jest wyłącznie za pomocą przesyłania komunikatów szybką siecią lokalną,
- **systemy rozproszone** (ang. *distributed systems*) - podobnie jak wielokomputery zbudowane są z indywidualnych systemów komputerowych, ale połączonych siecią rozległą.

Podstawową cechą pozwalającą rozróżnić systemy wieloprocesorowe ze wspólną pamięcią jest organizacja dostępu do RAM. Jeśli czasy dostępów do każdej z lokacji pamięci operacyjnej dla dowolnego procesora są porównywalne, to system wieloprocesorowy jest systemem typu UMA (ang. *Uniform Memory Access*). Komunikacja procesor-pamięć w takich komputerach może odbywać się za pomocą pojedynczej magistrali. Oznacza to, że wraz ze wzrostem liczby procesorów rośnie częstość kolizji, co skutkuje spadkiem wydajności systemu. Jest to główna wada systemów tego typu. Aby złagodzić konsekwencje tego zjawiska wyposaża się procesory w prywatną pamięć podręczną (ang. *cache*), która w razie konieczności może być uzupełniona o dodatkową prywatną pamięć RAM dla każdego procesora. Wymaga to jednak zapewnienia spójności informacji zawartych w pamięci podręcznej, a w przypadku korzystania z dodatkowej pamięci prywatnej odpowiedniego kompilatora, który umieszczałby w tej pamięci informacje tylko do odczytu

[1] Terminy „wielokomputer" i „multikomputer" będą używane wymiennie w tekście tej rozprawy, tak jak ma to miejsce w podstawowej literaturze przedmiotu.

(kod i stałe) oraz dane prywatne (stos). Możliwe jest również zastąpienie pojedynczej magistrali przełącznikiem krzyżowym lub siecią typu Omega [2, 5]. Niezależnie od przyjętego rozwiązania koszty budowy systemów typu UMA rosną wraz ze wzrostem liczby procesorów, dlatego typowe platformy UMA posiadają ich mniej niż sto. Systemy wieloprocesorowe ze wspólną pamięcią, posiadające więcej procesorów budowane są w oparciu o architekturę NUMA (ang. *Non-Uniform Memory Architecture*). W komputerach opartych na tej architekturze każdy procesor dysponuje lokalną pamięcią RAM. Przestrzeń adresowa nadal jest wspólna dla wszystkich procesorów, ale czas dostępu do pamięci lokalnej danego procesora jest krótszy niż czas dostępu do pamięci innego procesora. Tę różnicę można częściowo zniwelować wyposażając każdy procesor w pamięć podręczną. Systemy NUMA bez pamięci podręcznej określane są jako NC-NUMA (ang. *No Cache* NUMA), natomiast te, które ją posiadają nazywane są CC-NUMA (ang. *Cache-Coherent* NUMA). Te ostatnie budowane są w oparciu o katalogowe systemy wieloprocesorowe. W takich systemach istnieje centralny, sprzętowy katalog utrzymujący informacje o stanie i umiejscowieniu poszczególnych linii pamięci podręcznej. Odmianą architektury NUMA jest architektura COMA, która zakłada, że w systemie wieloprocesorowym pamięć operacyjna jest zastąpiona pamięcią podręczną [6]. W przeciwieństwie do systemów typu NUMA dane (ang. *data item*) nie posiadają procesora-właściciela, mogą migrować między pamięciami podręcznymi procesorów w zależności od tego, który procesor zgłosi na nie zapotrzebowanie. Głównym zagadnieniem, które musi być rozwiązane w projekcie takiego systemu jest efektywna lokalizacja danych. Podkategorią systemów wieloprocesorowych są komputery z procesorami wielordzeniowymi. W uproszczeniu są to systemy wieloprocesorowe zawarte w pojedynczym układzie scalonym (ang. CMP - *Chip-level MultiProcessor*), jednakże są różnice dzielące te dwa rozwiązania. Rdzenie w układzie wielordzeniowym mogą być jednakowe lub zróżnicowane (ang. *SoC - System on Chip*). W przeciwieństwie do typowego systemu wieloprocesorowego praca całego układu CMP może zostać zakłócona w wyniku awarii tylko jednego z rdzeni - układy te mają mniejszą niezawodność niż typowe systemy wieloprocesorowe. W niektórych układach CMP, np. firmy Intel, rdzenie mogą współdzielić pamięć podręczną drugiego poziomu, co nie jest możliwe w tradycyjnych systemach wieloprocesorowych. Odpowiednikiem systemów wieloprocesorowych budowanych na bazie sieci są układy NOC (ang. *Network On Chip*) [7, 8].

Systemy wielokomputerowe, nazywane również w literaturze klastrami, w przeciwieństwie do systemów wieloprocesorowych z pamięcią współdzieloną nie posiadają ograniczeń co do liczby obsługiwanych procesorów [9]. Budowane są z kompletnych lub okrojonych (np. pozbawionych większości urządzeń peryferyjnych) komputerów połączonych szybką siecią lokalną (np. Gigabit Ethernet, 10 Gigabit Ethernet, InfiniBand [10], Myrinet [11]). Każdy z tych komputerów, nazywanych węzłami, dysponuje prywatną pamięcią operacyjną. Brak jest wspólnej przestrzeni adresowej. Węzły mogą być zarówno maszynami jednoprocesorowymi, jak i wieloprocesorowymi. Komunikacja między nimi odbywa się na zasadzie przekazywania komunikatów (ang. *message passing*). Topologia sieci łączącej węzły wielokomputera jest najczęściej wybierana spośród topologii gwiazdy, pierścienia, siatki (ang. *mesh*), podwójnego torusa, sześcianu lub hipersześcianu. Stosowane są również rozwiązania mieszane. Topologie gwiazdy i pierścienia nie są odporne na awarie połączeń, gdyż każdy węzeł jest połączony z pozostałymi za pomocą jedynie jednego przewodu elektrycznego lub światłowodu. Pozostałe z wymienionych topologii pozbawione są tej negatywnej cechy, ale posiadają dodatkowe kryterium, które musi być uwzględnione przy ich wyborze. Jest to długość najdłuższej ścieżki łączącej dowolne dwa węzły multikomputera, czyli średnicy sieci. Od średnicy zależą opóźnienia w sieci, im dłuższa średnica, tym większe opóźnienia. Sieć o topologii siatki (najgorszy przypadek), która łączy n węzłów ma średnicę równą $\sqrt{n}$, natomiast sieć tworząca hipersześcian (najlepszy przypadek) o takiej samej liczbie węzłów ma średnicę równą $log_2(n)$. Ponieważ sieci o małej średnicy charakteryzują się dużą liczbą połączeń, to w praktyce wybór topologii może być podyktowany względami ekonomicznymi. Na

wydajność sieci oprócz średnicy ma wpływ również mechanizm przełączania. Zazwyczaj w systemach wielokomputerowych stosuje się dwa rodzaje takich mechanizmów. Pierwszy nosi nazwę „przełączania pakietów typu przechowaj i prześlij" (ang. *store-and-forward packet switching*), a drugi „przełączania obwodów". Mechanizm przełączania przechowaj i prześlij dzieli przesyłany komunikat na fragmenty nazywane pakietami. Wielkość pakietu zależy od pojemności buforów w jakie wyposażone są interfejsy sieciowe poszczególnych węzłów i przełączników sieciowych. Między przełącznikami sieciowymi pakiety są przesyłane szeregowo. Pakiet jest przesyłany dalej dopiero wtedy, gdy w całości znajdzie się w buforze interfejsu przełącznika sieciowego. Zaletą tej metody przełączania jest możliwość wyboru trasy dla każdego z pakietów z osobna. Niestety, powoduje to również duże opóźnienia w transmisji. W metodzie przełączania obwodów jeden z przełączników ustala najpierw ścieżkę wiodącą przez inne przełączniki do węzła docelowego, a następnie tą ścieżką są transmitowane wszystkie bity. Skutkuje to mniejszymi opóźnieniami niż w przypadku przełączania i przesyłania pakietów. Opóźnienia związane są głównie z zestawianiem ścieżki. Techniką pośrednią między przedstawionymi metodami jest trasowanie kanalikowe (ang. *wormhole routing*), w którym pakiety są dzielone na mniejsze jednostki, z których pierwsza wyznacza trasę, którą podążają następne. Oprócz efektywności komunikacji przez sieć, innymi czynnikami decydującymi o wydajności multikomputera są budowa interfejsu sieciowego węzłów oraz wydajność oprogramowania, które ten interfejs obsługuje. Większość będących w użyciu interfejsów sieciowych jest wyposażona w możliwość transmisji DMA (ang. *Direct Memory Access*) oraz specjalizowany procesor, co pozwala na uwolnienie głównego procesora węzła od konieczności nadzorowania przebiegu komunikacji przez sieć. Najnowsze rozwiązania w dziedzinie szybkich sieci komputerowych, takie jak InfiniBand oferują możliwość transmisji RDMA (ang. *Remote* DMA), która eliminuje konieczność angażowania systemu operacyjnego w odbiór i wysyłanie poprzez interfejs sieciowy komunikatów pochodzących od procesów użytkowników. Jego rola ogranicza się do odwzorowania w przestrzeni adresowej procesu użytkownika bufora interfejsu sieciowego. Ewentualne błędy w obsłudze tego interfejsu, powstałe z winy aplikacji użytkownika powodują powstanie wyjątków naruszenia ochrony pamięci. Systemy operacyjne węzłów multikomputera dostarczają procesom i wątkom użytkownika zarówno prostych, jak i zaawansowanych środków komunikacji. Do pierwszej grupy zaliczają się wywołania systemowe pozwalające nadawać (ang. *send*) lub odbierać (ang. *receive*) komunikaty. Mogą one występować zarówno w formie synchronicznej (blokującej), jak i asynchronicznej. W pierwszym przypadku wykonanie procesu jest wstrzymywane do chwili zakończenia transmisji komunikatu, w drugim może być kontynuowane. Te wywołania stanowią podstawę budowy bardziej złożonych środków komunikacji, takich jak zdalne wywołania procedur (ang. RPC - *Remote Procedure Call*) lub w przypadku oprogramowania opartego na paradygmacie obiektowym zdalne wywołanie metod (ang. RMI - *Remote Method Invocation*). Jedną z najbardziej zaawansowanych form komunikacji między węzłami w systemach wielokomputerowych jest współdzielona pamięć rozproszona (ang. DSM - *Distributed Shared Memory*) [2,5,9,12,13]. Rozwiązanie to ma na celu umożliwienie programistom aplikacji użytkownika korzystanie z modelu pamięci współdzielonej w środowisku, które jest takiej pamięci pozbawione. Podobnie jak zdalne wywołania procedur i metod rozproszona pamięć dzielona bazuje na przesyłaniu komunikatów między węzłami. DSM jest szerzej opisywana w Rozdziale 2.

Systemy rozproszone, podobnie jak multikomputery składają się z wielu komputerów połączonych siecią, jednakże w ich przypadku jest to sieć rozległa (ang. WAN - *Wide Area Network*). W wielokomputerze wszystkie węzły mają zazwyczaj jednakową architekturę, a więc jest to system homogeniczny (jednorodny). Węzły systemu rozproszonego najczęściej posiadają różną architekturę, więc jest to system różnorodny (heterogeniczny). Współpracę miedzy takimi węzłami zapewnia dodatkowa warstwa oprogramowania nazywana oprogramowaniem pośredniczącym (ang. *middleware*). Jest ona częścią oprogramowania systemowego, jednakże nie znajduje

się w jądrze systemów operacyjnych poszczególnych węzłów. Jest oprogramowaniem przestrzeni użytkownika, które dostarcza usług aplikacjom użytkowym, ukrywając przed nimi rozproszony charakter systemu. Węzły systemu rozproszonego mogą być fizycznie oddalone od siebie nawet o tysiące kilometrów. Komplikuje to administrowanie takim systemem oraz generuje znacznie większe opóźnienia w transmisji przez sieć, niż te, które pojawiają się w systemie wielokomputerowym, a mechanizmy tolerowania awarii sieci stają się ważnym elementem systemu. Systemy rozproszone posiadają trzy własności, które pozwalają je scharakteryzować: przezroczystość (ang. *transparency*), otwartość (ang. *openness*) oraz skalowalność (ang. *scalability*) [5, 12]. Przezroczystość określa sposób, w jaki użytkownicy i aplikacje postrzegają system rozproszony. Im bardziej jego zachowanie zbliżone jest do zachowania pojedynczego systemu komputerowego, tym większy jest jego stopień przezroczystości. Otwartość wyznacza zgodność systemu ze standardowymi regułami określającymi składnię i semantykę usług dostarczanych przez ten system. Im większa jest ta zgodność, tym łatwiejsza jest rozbudowa systemu. Skalowalność charakteryzuje możliwość rozszerzania systemu. Jeśli system jest dobrze skalowalny, to dodanie do niego nowych zasobów nie wpływa negatywnie na jego wydajność i sposób zarządzania. Nie zmienia się również sposób korzystania z tego systemu. Cechę skalowalności można odnieść również do pozostałych kategorii systemów wieloprocesorowych. W przypadku wieloprocesorów z pamięcią dzieloną będzie ona określała możliwość podłączenia nowych procesorów, a w systemach multikomputerowych dołączenia nowych węzłów. Pewną podgrupą systemów rozproszonych są systemy nazywane gridami (ang. *grid*). Tworzą je rozproszone węzły, które pracują nad wspólnym zadaniem, najczęściej związanym z problemami obliczeniowymi.

Między systemami komputerowymi należącymi do wymienionych kategorii zachodzi wiele różnic, ale można też znaleźć podobieństwa. Podstawową cechą odróżniającą systemy wieloprocesorowe od wielokomputerowych i rozproszonych jest sposób postrzegania pamięci operacyjnej przez poszczególne procesory. W wieloprocesorze pamięć jest wspólna dla wszystkich procesorów, w wielokomputerze i systemie rozproszonym każdy dysponuje własną. Jednakże komputery typu NUMA i COMA posiadają pamięć fizycznie rozproszoną, podobnie jak systemy należące do dwóch pozostałych kategorii, choć, tak jak w innych wieloprocesorach ta pamięć jest dostępna dla wszystkich procesorów. Zagadnienia związane z zarządzaniem pamięcią rozproszoną w multikomputerach, systemach rozproszonych oraz systemach NUMA i COMA będą więc podobne. Sposób postrzegania RAM przez procesory określa model komunikacji między procesami jaki może zastosować twórca aplikacji użytkowych. Dla systemów ze wspólna pamięcią komunikacja ta odbywa się za pomocą zmiennych współdzielonych. W systemach z pamięcią rozproszoną procesy mogą jedynie wysyłać między sobą komunikaty, o ile nie zostaną zastosowane rozwiązania typu DSM. Z punktu widzenia struktury fizycznej systemy wieloprocesorowe można klasyfikować uwzględniając sposób łączenia poszczególnych elementów systemu. W wieloprocesorze najprostszym sposobem połączenia procesorów jest użycie wspólnej magistrali. Podobne rozwiązanie może być zastosowane względem węzłów multikomputera [5]. Podobnie sieci przełączane, które są dominującym sposobem łączenia elementów wielokomputera mogą być zastosowane w systemach wieloprocesorowych ze wspólną pamięcią. W przypadku multikomputerów i systemów rozproszonych różnice są jeszcze mniejsze niż między wielokomputerami i wieloprocesorami ze wspólną pamięcią. Niektóre źródła (np. [12]) zaliczają wielokomputery do systemów rozproszonych. Inne (np. [5]) traktują wielokomputery jako bazę sprzętową do realizowania, za pomocą oprogramowania, systemów rozproszonych, które mogą być ściśle powiązane (ang. *tightly-coupled*) lub luźno powiązane (ang. *loosely-coupled*). Istnienie analogii między wymienionymi kategoriami systemów oznacza, że część zagadnień ich dotyczących będzie wspólna. Wybór typu systemu wieloprocesorowego zazwyczaj jest podyktowany kosztami ekonomicznymi. W tym względzie przewagę mają multikomputery. Są tańsze w budowie niż wieloprocesory ze wspólną pamięcią. Można je skonstruować używając typowych elementów, takich jak komputery klasy

PC oraz popularne sieci LAN jak Gigabit Ethernet. Tego typu systemy są określane mianem COW (ang. *Cluster Of Workstations*) lub NOW (ang. *Network of Workstations*). Droższe implementacje wielokomputerów, nazywane MPP (ang. *Massively Parallel Processing*) są również konstruowane na bazie komputerów klasy PC, ale łączonych specjalizowaną, szybką siecią komputerową. Inną przewagą wielokomputerów na wielprocesorami ze wspólną pamięcią jest skalowalność [14]. Systemy tego typu mogą zawierać znacznie więcej procesorów niż wieloprocesory. Wybór między multikomputerem, a systemem rozproszonym zależy głównie od zadań, do których będą one przeznaczone [2]. Wielokomputery są odpowiednie dla aplikacji wymagających intensywnych obliczeń, a systemy rozproszone dla tych, które zorientowane są na komunikację. Wyjątkiem od tej reguły są gridy. Inną cechą, która może być brana pod uwagę przy wyborze między tymi dwoma rozwiązaniami jest lokalizacja. Budowa multikomputerów bazuje na sieciach lokalnych, a więc mogą one w całości należeć do pojedynczego podmiotu (np. firma, uczelnia). Systemy rozproszone bazują na sieciach rozległych, co oznacza, że poszczególne elementy (węzły, infrastruktura sieciowa) takich systemów mogą mieć różnych właścicieli. Utrzymanie (ang. *maintenance*) systemu rozproszonego może zatem wymagać koordynacji działań wielu organizacji, a nawet indywidualnych użytkowników, podczas gdy multikomputerem może zajmować się tylko wydzielona jednostka organizacji posiadającej taki system.

1.2. Wirtualizacja pamięci wielokomputera

Systemy wielokomputerowe pozwalają łączyć zasoby. Aby wydajnie z nich korzystać potrzebne jest efektywne zarządzanie. Jedną z technik, która to umożliwia jest wirtualizacja (ang. *virtualization*), która swym znaczeniem obejmuje dwa rodzaje działań [15]:

- **podział zasobów**, który może być zastosowany zarówno w systemach jednoprocesorowych (uniprocesorowych), jak i systemach wieloprocesorowych - przykładem może być podział czasu procesora lub klasyczne techniki pamięci wirtualnej,
- **agregacja lub kombinacja zasobów**, która jest charakterystyczna dla systemów rozproszonych i multikomputerowych - przykładem mogą być rozproszone systemy plików [5, 12, 16].

Wirtualizacja może być zastosowana w wielu warstwach systemu komputerowego, przykładowo w [17] wyróżniono:

- **Wirtualizację dostępu** (ang. *access virtualization*) technika wymagająca wsparcia ze strony oprogramowania i/lub sprzętu, której celem jest zapewnienie współpracy między urządzeniami peryferyjnymi lub obiektami danych, a aplikacjami. Przykładem zastosowania tej techniki mogą być *urządzenia logiczne*, które tworzone są przez system operacyjny, aby ukryć przed aplikacją szczegóły działania urządzeń fizycznych. Wirtualizacja upraszcza ich obsługę sprowadzając ją do korzystania z interfejsu znanego aplikacji użytkownika. W systemach rozproszonych wirtualizacja dostępu umożliwia realizację przezroczystości dostępu [5, 12].
- **Wirtualizację aplikacji** (ang. *application virtualization*) - techniki programowe, które pozwalają uruchamiać tę samą aplikację na różnych platformach systemowych, pracujących pod kontrolą różnych systemów operacyjnych. Oprogramowanie realizujące ten rodzaj wirtualizacji może również dokonywać innych czynności wspierających pracę aplikacji, takich jak restart, jeżeli uległy one awarii (ang. *failure*) lub równoważenie obciążenia (ang. *load balancing*), aby zwiększyć ich skalowalność. Przykładem jest maszyna wirtualna języka Java.

- **Wirtualizacja przetwarzania** (ang. *processing virtualization*) - jest to rozwiązanie programowo-sprzętowe, które ma na celu ukrycie przed procesami użytkowymi lub systemem operacyjnym fizycznej struktury platformy sprzętowej, na której są wykonywane. Tego typu rozwiązania pozwalają aplikacjom lub systemom operacyjnym postrzegać pojedynczy komputer jako grupę systemów komputerowych (Xen [18], IBM VM/370 [2, 4]) albo przedstawiają system wielokomputerowy jako pojedynczy komputer (Kerrighed [19], inne rozwiązania typu SSI - *Single Image System*).

- **Wirtualizacja magazynu danych** (ang. *storage virtualization*) - zbiór rozwiązań sprzętowych i programowych, które ukrywają przed aplikacjami użytkowymi szczegóły związane z przechowywaniem danych, takie jak miejsce ich magazynowania. Umożliwiają równoczesny dostęp do danych wielu użytkownikom. Przykładami takich rozwiązań są rozproszone systemy plików, jak NFS [5], GFS [16] OceanStore [20] i GlusterFS [21], rozwiązania typu NAS (ang. *network attached storage*) lub SAN (ang. *storage attached network*).

- **Wirtualizacja sieci komputerowej** (ang. *network virtualization*) - techniki programowo-sprzętowe umożliwiające stworzenie aplikacjom użytkownika obrazu sieci komputerowej, który różni się od jej fizycznej struktury.

- **Zarządzanie środowiskami wirtualnymi** (ang. *management of virtualized environments*) - rozwiązania programowe pozwalające zarządzać wieloma systemami tak, jakby były one pojedynczym zasobem (SSI).

Oprogramowanie realizujące wirtualizację w systemie komputerowym może być częścią systemu operacyjnego, stanowić warstwę pośrednią (ang. *middleware*) lub być częścią aplikacji użytkownika.

Ważnym zagadnieniem w systemach multikomputerowych oraz rozproszonych jest wirtualizacja pamięci rozproszonej (ang. *memory virtualization*) [15,22–24]. Jest ona formą wirtualizacji magazynu danych, ale swoim zakresem częściowo pokrywa się z wirtualizacją dostępu i przetwarzania. Jej podstawowym zadaniem jest utworzenie wspólnego zasobu z lokalnych pamięci węzłów multikomputera lub systemu rozproszonego. Cel ten można zrealizować na wiele sposobów. Pamięć w węzłach wielokomputera może być użyta jako urządzenie wymiany (ang. *swap device*) w implementacjach pamięci wirtualnej [25] lub jako zamiennik pamięci masowej [26]. Wirtualizacja pamięci może też mieć na celu stworzenie DSM, aby w systemach o rozproszonej RAM można było stosować paradygmat programowania z użyciem pamięci współdzielonej [9,13]. Może również ona dostarczać aplikacjom użytkowym abstrakcji pamięci w postaci rozproszonej pamięci podręcznej [1,23] lub szybkiego łącza komunikacyjnego [15,23,24]. Wirtualizacja pamięci multikomputera może być przeprowadzona na poziomie sprzętowym, systemu operacyjnego, warstwy pośredniej oprogramowania lub na poziomie aplikacji użytkownika.

1.3. Skalowalne, Rozproszone Struktury Danych

Skalowalne, rozproszone struktury danych (ang. SDDS - *Scalable Distributed Data Structures*) są metodą wirtualizacji rozproszonej pamięci multikomputera na poziomie aplikacji użytkownika [27]. Część węzłów wielokomputera pełni rolę klientów SDDS, a pozostałe serwerów SDDS, które przechowują rekordy danych w swoich pamięciach operacyjnych. Każdy klient SDDS posiada pewne parametry pliku, nazywane obrazem, które są niezbędne do ustalenia adresu serwera przechowującego określony rekord. SDDS muszą spełniać trzy podstawowe założenia [14]:

1. Do adresowania danych nie może być używany żaden centralny katalog.

2. Obraz SDDS, jaki posiada klient może ulec dezaktualizacji. Jego aktualizacja dokonywana jest za pomocą komunikatów korygujących nazywanych IAM (ang. *Image Adjustment Message*).

3. Klient z niepoprawnym (przedawnionym) obrazem może wysłać zapytanie do niewłaściwego serwera SDDS, który musi w wypadku wystąpienia takiego zdarzenia przesłać klientowi komunikat IAM, a jego żądanie skierować do właściwego serwera.

Dotychczas autorzy koncepcji opracowali trzy podstawowe architektury SDDS:

- LH* - korzysta z haszowania liniowego [28] uogólnionego na środowiska z pamięcią rozproszoną [27],
- RP* - stosuje technikę podziałów zakresów, znaną z B^+ drzew, która pozwala przechowywać rekordy w sposób uporządkowany [14],
- SD-Rtree - używa adresowania wieloindeksowego do lokalizacji danych przestrzennych [29].

Dla wszystkich tych architektur istnieje pewien wspólny schemat, według którego są budowane. We wszystkich podstawową jednostką danych jest rekord, który identyfikowany jest na podstawie unikatowego klucza. Rekordy przechowywane są przez serwery SDDS w większych strukturach nazywanych wiaderkami (ang. *bucket*). Wiaderka rezydują zazwyczaj w pamięci operacyjnej węzła, na którym uruchomiony jest serwer SDDS. Wszystkie wiaderka tworzą rozproszony plik danych. Klienci SDDS mogą dodawać, aktualizować lub usuwać rekordy z wiaderek. Jeśli w trakcie wstawiania rekordów do wiaderka ulegnie ono przepełnieniu, to wykonywany jest jego podział. W wyniku przeprowadzenia tej czynności powstaje nowe wiaderko, które przejmuje około połowy rekordów zgromadzonych w przepełnionym wiaderku. W ten sposób następuje powiększenie rozproszonego pliku SDDS. Nie we wszystkich architekturach SDDS wiaderko, które ulegnie przepełnieniu podlega natychmiastowemu podziałowi. Przykładem takiej architektury jest LH*. Posiada ona dodatkowy element, nazywany koordynatorem podziałów, który wyznacza kolejność wykonywania tej czynności i który może nakazać innym wiaderkom przeprowadzenie podziału, zanim pozwoli podzielić się wiaderku przepełnionemu. Klienci posługują się swoim obrazem pliku SDDS, aby wyznaczyć w nim lokalizację rekordów. Obraz ten staje się nieaktualny na skutek podziałów wiaderek. Sposób jego korekty opisano w założeniach 1 i 3 Skalowalnych, Rozproszonych Struktur Danych. Wyjątkiem od tych reguł jest struktura RP^*_N . W jej przypadku wszelkie zapytania ze strony klienta są wykonywane na drodze komunikacji wielopunktowej (ang. *multicast*).

1.4. Cel i teza rozprawy

Spośród wymienionych w Podrozdziale 1.2 przykładów wirtualizacji na uwagę zasługuje wirtualizacja przechowywania danych dotycząca pamięci operacyjnej. Ta technika nazywana jest krótko „wirtualizacją pamięci". Należy zaznaczyć, że ten termin jest dwuznaczny. Dla systemów jedno lub wieloprocesorowych oznacza technikę zarządzania wspólną pamięcią, która polega na podziale dostępnej RAM, tak, aby każdy z procesów użytkownika mógł dysponować własną przestrzenią adresową. W przypadku systemów wielokomputerowych może również oznaczać agregację rozproszonej pamięci operacyjnej, celem uzyskania jej spójnego obrazu. Oba te zagadnienia opisane są szerzej w Rozdziale 2.

Jedną z technik wirtualizacji pamięci opisanych w tym rozdziale są Skalowalne, Rozproszone Struktury Danych (ang. *Scalable Distributed Data Structures*). SDDS jest aplikacją wirtualizującą pamięć RAM multikomputera. W SDDS rolę ramek klasycznej pamięci wirtualnej spełniają

wiaderka, natomiast rekordy są odpowiednikami stron. O ile w przypadku klasycznej pamięci wirtualnej MMU jest zarządzane z poziomu systemu operacyjnego, to SDDS jest typową aplikacją, stanowiącą warstwę pośredniczącą pomiędzy systemem operacyjnym, a klientami SDDS. Powstaje pytanie, w jaki sposób zarządzanie wiaderkami i rekordami przenieść na poziom systemu operacyjnego (analogicznie jak zarządzanie stronami i ramkami klasycznej pamięci wirtualnej) i jaka byłaby wydajność takiego rozwiązania. Zastosowanie obecnie dostępnych implementacji SDDS wymaga ingerencji w kod źródłowy aplikacji użytkownika, które mają korzystać z tego rozwiązania. Teza badawcza pracy zakłada, że można uniknąć tej niedogodności implementując część kliencką Skalowalnych, Rozproszonych Struktur Danych na poziomie systemu operacyjnego. W przypadku pewnych typów aplikacji to rozwiązanie nie powinno wpływać negatywnie na wydajność, a nawet możliwy jest jej wzrost. Główną zaletą tego rozwiązania jest skalowalność.

Praca zawiera opis architektury takiego rozwiązania i analizę jego efektywności. Badanie wydajności wskazuje zakres zastosowania, czyli określa zbiór aplikacji, dla których korzystanie z SDDS zaimplementowanych częściowo na poziomie systemu operacyjnego jest opłacalne. Uzasadnieniu podjęcia takich badań, dokładniejszemu przedstawieniu ich celu oraz postawionej tezie jest poświęcony Rozdział 3 rozprawy.

1.5. Podsumowanie

W rozdziale zostały przedstawione podstawowe pojęcia związane z tematyką pracy oraz została określona główna teza rozprawy i jej cel. Rozdział drugi zawiera informacje na temat aktualnego stanu badań dotyczących dziedziny pracy. W rozdziale trzecim sformułowano szczegółowo problem. W rozdziale czwartym przeprowadzono analizę problemu i opisano architekturę rozwiązania. Kolejne dwa rozdziały zawierają odpowiednio opis implementacji rozwiązania i jego ocenę eksperymentalną. Ostatni rozdział pracy zawiera wnioski, które sformułowano na podstawie analizy zaproponowanego rozwiązania oraz wyników eksperymentów.

2. Wirtualizacja pamięci w systemach multikomputerowych

W tym rozdziale zawarto opis wykorzystania techniki wirtualizacji w odniesieniu do zarządzania pamięcią operacyjną w systemach wielokomputerowych. Większość metod związanych z tym zagadnieniem opiera się na rozwiązaniach, które zostały opracowane na potrzeby systemów jednoprocesorowych. Podrozdział 2.1 zawiera informacje na temat tych rozwiązań. Kolejny Podrozdział (2.2) traktuje o możliwościach zastosowania wirtualizacji do obsługi rozproszonej pamięci operacyjnej multikomputerów. Cele, dla których zostały zaprojektowane przedstawione w tym podrozdziale rozwiązania mogą być różne, jednak każde z nich powinno spełniać takie kryteria jak skalowalność, wydajność i odporność na błędy, które są istotne w systemach wielokomputerowych. W podsumowaniu rozdziału umieszczono porównanie, którego celem jest wyłonienie tego rozwiązania, które w największym stopniu spełnia wymagania stawiane przed multikomputerami.

2.1. Pamięć wirtualna w systemach jednoprocesorowych

Wirtualizacja może dotyczyć zarówno podziału, jak i agregacji zasobów [15]. Pierwsze podejście zostało wykorzystane do opracowania koncepcji pamięci wirtualnej (ang. *virtual memory*) w systemach jednoprocesorowych, na której również bazuje zarządzanie pamięcią operacyjną w systemach wielokomputerowych. Idea pamięci wirtualnej zakłada, że proces może być podzielony na fragmenty i tylko te z nich (tzw. zestaw rezydentny (ang. *resident set*)), które w danej chwili są niezbędne do kontynuowania wykonania procesu są załadowane do pamięci operacyjnej [2–4]. Efektywność takiego rozwiązania wynika z badań empirycznych, które doprowadziły do sformułowania zasad lokalności (ang. *principles of locality*):

- **Zasada lokalności czasowej** (ang. *principle of temporal locality*) - jeśli podczas wykonania procesu nastąpiło w pewnej chwili odwołanie do pewnej lokacji w jego pamięci, to istnieje niezerowe prawdopodobieństwo, że w najbliższym czasie od tej chwili to odwołanie zostanie powtórzone.
- **Zasada lokalności przestrzennej** (ang. *principle of spatial locality*) - jeśli podczas wykonania procesu nastąpiło odwołanie do pewnej lokacji w jego pamięci, to istnieje niezerowe prawdopodobieństwo, że kolejne odwołania będą dotyczyły lokacji położonych w jej pobliżu.

Podsumowując - z obu zasad lokalności wynika, że odwołania do obszaru pamięci operacyjnej przydzielonego wykonującemu się procesowi mają tendencję do grupowania. Realizacja pamięci wirtualnej wymaga wsparcia ze strony systemu operacyjnego i sprzętu [3,4]. Jeśli proces odwoła się do fragmentu, którego nie ma w pamięci operacyjnej, to uaktywniana jest pułapka

(ang. *trap*), czyli przerwanie obsługujące sytuacje wyjątkowe. Procedura związana z tym przerwaniem jest częścią systemu operacyjnego. Po rozpoczęciu działania sprawdza ona poprawność adresu wygenerowanego przez proces. Jeśli ten adres jest błędny, to proces nie będzie dalej wykonywany. W przeciwnym przypadku procedura używa adresu do identyfikacji brakującego fragmentu procesu. Ten fragment jest następnie sprowadzany do pamięci operacyjnej i proces jest wznawiany. Czynność sprowadzenia odpowiedniego fragmentu procesu do RAM obejmuje wyznaczenie dla niego wolnego obszaru pamięci, jeśli taki istnieje lub wyznaczenie innego fragmentu procesu, który w danej chwili rezyduje w pamięci operacyjnej. Taki fragment jest wycofywany z RAM do urządzenia wymiany (ang. *swap device*), które jest często zrealizowane przy pomocy pamięci dyskowej, a w jego miejsce wprowadzany jest do pamięci operacyjnej brakujący fragment. Podstawową trudnością w wyborze fragmentu do wymiany jest ustalenie, który z fragmentów znajdujących się w danej chwili w RAM będzie najmniej przydatny w przyszłości, tzn. do którego z nich proces nie będzie się najdłużej odwoływał. To zadanie realizuje system operacyjny posługując się takimi algorytmami jak LRU (wymieniany jest najdłużej nieużywany fragment) lub FIFO (wymieniany jest fragment z zestawu rezydentnego, który jako pierwszy znalazł się w pamięci operacyjnej). Trafność ocen generowanych przez te algorytmy jest jednym z czynników determinujących efektywność pamięci wirtualnej. Ich niska skuteczność jest jedną z przyczyn występowania zjawiska, które określane jest mianem szamotania lub migotania (ang. *trashing*), a które polega na intensywnej wymianie fragmentów procesów, prowadzącej do zagłodzenia ich wykonania. System operacyjny musi także utrzymywać strukturę danych, w której zawarte są informacje o położeniu wszystkich fragmentów procesu zarówno w pamięci operacyjnej, jak i w urządzeniu wymiany. Ponieważ przed wykonaniem programu nie można określić, w którym miejscu pamięci znajdą się jego poszczególne fragmenty, to proces musi generować adresy logiczne nazywane wirtualnymi, które następnie są dynamicznie przeliczane na adresy fizyczne. Tego przeliczenia dokonuje jednostka zarządzania pamięcią (ang. *Memory Management Unit* - MMU) na podstawie informacji zawartych w wyżej wspomnianej strukturze danych. Do zalet pamięci wirtualnej należą:

- Zwiększenie liczby procesów, które równocześnie rezydują w pamięci operacyjnej. Wynika to z braku konieczności ładowania ich w całości do RAM.
- Możliwość przydzielenia procesowi większej ilości pamięci niż jest fizycznie dostępne w systemie komputerowym. Wynika to z oddzielenia adresów fizycznych (fizycznej przestrzeni adresowej) od adresów wirtualnych (wirtualnej przestrzeni adresowej). Dzięki temu proces może być nawet większy od całkowitej pamięci fizycznej systemu komputerowego. W teorii pamięć wirtualna procesu może być nieskończona, w praktyce ogranicza ją rozmiar stosowanych w systemie adresów, czyli szerokość magistrali adresowej.

Koncepcja ta realizowana jest najczęściej w postaci jednego z trzech rozwiązań: stronicowania na żądanie (ang. *demand paging*), segmentacji na żądanie lub stronicowanej segmentacji na żądanie.

2.1.1. Stronicowanie na żądanie

Stronicowanie na żądanie bazuje na technice zarządzania pamięcią nazywanej stronicowaniem. W stronicowaniu rolę fragmentów procesu pełnią strony. Wszystkie strony procesów mają ten sam rozmiar wyrażony potęgą liczby dwa. Często stosowane są strony o wielkości 4 KiB i 8 KiB[1], ale spotykane są również rozwiązania, w których zastosowano strony mniejsze lub większe. Pamięć fizyczna komputera jest podzielona na obszary nazywane ramkami lub stronami fizycznymi [2–4, 30], które odpowiadają wielkością stronom. Również nośnik urządzenia

[1]Oznaczenie przedrostków jednostek pamięci według normy IEC 60027.

wymiany podzielony jest logicznie na bloki o takiej samej wielkości jak strony. Rozmieszczenie stron w ramkach jest odwzorowane w tablic stron (ang. *page table*). Elementy tej tablicy indeksowane są numerami stron i zawierają adres bazowy ramki, w której znajduje się strona o danym numerze. Adres logiczny w stronicowaniu składa się z numeru strony (starsze bity) i przesunięcia względem początku tej strony (młodsze bity). Translację adresu logicznego na fizyczny przeprowadza jednostka MMU, korzystając z tablicy stron. Czynność ta sprowadza się do odczytania z tej tablicy adresu bazowego ramki i umieszczeniu go w adresie w miejscu numeru strony. Oprócz informacji o fizycznym położeniu strony w pamięci każda pozycja tablicy stron może również zawierać informacje o tym jakie operacje są możliwe na zawartości strony (np. odczyt, zapis, wykonanie) i czy zawartość strony była ostatnio modyfikowana. Pierwszy rodzaj dodatkowych informacji pozwala na wprowadzenie ochrony zawartości stron oraz na optymalizację polegającą na umożliwieniu współdzielenia przez procesy stron, których modyfikacja jest zabroniona. Drugi rodzaj informacji również powiązany jest z optymalizacją - jeśli zawartość strony nie była modyfikowana, to jej kopia istnieje w pamięci pomocniczej i nie trzeba jej kopiować do urządzenia wymiany. W zwykłym stronicowaniu [3, 4] wszystkie strony procesu muszą zostać umieszczone w ramkach pamięci operacyjnej, ale nie jest ważne w jakiej kolejności ani nawet, czy będą przylegać do siebie (tworzyć obszar ciągły w pamięci fizycznej). Jeśli pojawi się konieczność wprowadzenia do pamięci procesu o wyższym priorytecie niż te, które znajdują się w niej obecnie, a nie będzie wystarczającej liczby wolnych ramek, by tę czynność wykonać, to system operacyjny może wycofać w całości do urządzenia wymiany proces o niskim priorytecie, a w jego miejsce umieścić proces o wysokim priorytecie.

Stronicowanie na żądanie (ang. *demand paging*), które jest implementacją pamięci wirtualnej stanowi modyfikację scenariusza zwykłego stronicowania. Każdy z procesów, który ma być wykonany otrzymuje od systemu operacyjnego pewną pulę wolnych ramek, do których ładowane są strony w momencie, kiedy proces się do nich odwoła (ang. *lazy loading*). Jeśli wyczerpią się wolne ramki, to system operacyjny może zastosować wymianę stron. Ponieważ rozmiar procesu w stronicowaniu na żądanie może być większy niż rozmiar pamięci operacyjnej, to tablica stron staje się strukturą o znaczącej wielkości. Aby zmniejszyć zużycie pamięci operacyjnej przez tę strukturę opracowano szereg jej implementacji, takich jak wielopoziomowa tablica stron (ang. *multilevel page table*) i odwrócona tablica stron (ang. *inverted page table*). W porównaniu ze zwykłym stronicowaniem każda pozycja tablicy stron może przechowywać więcej informacji. Dodatkowe informacje obejmują: bit obecności informujący o tym, czy dana strona jest załadowana do pamięci operacyjnej, czy znajduje się w urządzeniu wymiany, informacje o położeniu strony w urządzeniu wymiany lub odnośnik do innej struktury danych, która taką informację zawiera, informacje wymagane do działania algorytmów wymiany, takie jak np. czas ostatniego odwołania do strony. Do zjawisk niekorzystnych związanych ze stronicowaniem na żądanie należy zaliczyć migotanie i fragmentację wewnętrzną. Zjawisko migotania w tej implementacji pamięci wirtualnej może być spowodowane, oprócz małej skuteczności algorytmu typowania stron do wymiany, niedostateczną liczbą ramek, które procesowi przydzielił system operacyjny. Fragmentacja wewnętrzna polega na przydziale procesowi obszaru pamięci, którego on nigdy nie wykorzysta. W przypadku stronicowania jest to fragment strony, który w najgorszym przypadku może obejmować jej całość, oprócz jednego bajta (w ogólnym przypadku słowa). Z punktu widzenia programistów tworzących aplikacje wielowątkowe ważną cechą stronicowania jest możliwość współdzielenia stron przez procesy. Strony współdzielone (ang. *shared pages*) można wykorzystać nie tylko do wspomnianej wcześniej optymalizacji, ale również jako środek komunikacji między procesami, który nazywany jest pamięcią dzieloną (ang. *shared memory*).

2.1.2. Segmentacja na żądanie

W segmentacji [3, 4] proces jest podzielony na fragmenty, które mogą być różnej wielkości. Podział ten powinien być zależny od wewnętrznej struktury procesu. Typowe rodzaje segmentów obejmują np. segment kodu, segment danych i segment stosu. Podobnie jak w przypadku stronicowania rozmieszczenie segmentów w pamięci operacyjnej zapisane jest w strukturze danych nazywanej tablicą segmentów. Każda pozycja takiej tablicy zawiera informację na temat adresu bazowego segmentu w RAM oraz długości tego segmentu. Numer segmentu jest indeksem pozycji w tablicy segmentów. Adres logiczny (a w przypadku segmentacji na żądanie adres wirtualny) składa się, podobnie jak w stronicowaniu, z numeru segmentu i przemieszczenia względem jego początku. Adres fizyczny powstaje poprzez zamianę numeru segmentu na jego adres bazowy odczytany z tablicy. Jeśli pamięć wirtualna jest realizowana przy pomocy segmentacji, to każda pozycja tablicy segmentów powinna zostać wyposażona w dodatkowe elementy umożliwiające przechowywanie informacji o stanie segmentu (obecność w pamięci operacyjnej, modyfikacja zawartości) oraz jego położeniu w urządzeniu wymiany (jeśli segment tam się znajdzie). Segmentacja jest obarczona fragmentacją zewnętrzną. Ten rodzaj fragmentacji oznacza, że w pamięci operacyjnej powstają wolne obszary, które nie mogą być wykorzystane. Ponieważ segmenty mogą być przemieszczane w RAM, to fragmentację zewnętrzną można zlikwidować poprzez taką relokację segmentów, aby tworzyły one ciągły obszar w pamięci fizycznej. Segmenty mogą być współdzielone przez procesy, co pozwala na zaoszczędzenie miejsca w pamięci operacyjnej i na realizację pamięci dzielonej. Zaletą segmentacji jest to, że programista ma wpływ na podział procesu na segmenty. Dzięki temu może on np. zapewnić dodatkową ochronę wybranym strukturom danych. Jeśli ta struktura może zmieniać swój rozmiar (jak np. lista), to rozmiar segmentu, w którym jest ona umieszczona można odpowiednio do tych zmian dostosować. Jednakże, w przypadku segmentacji na żądanie ta cecha może się okazać wadą, ponieważ utrudnia zarządzanie wymianą segmentów. Systemy operacyjne, w których zastosowano segmentację na żądanie, mogą zawierać wywołania systemowe pozwalające procesom wskazywać, które segmenty nadają się do wymiany, a które powinny pozostać w RAM [31]. Przykładem takiego systemu operacyjnego jest IBM OS/2. Segmentacja na żądanie jest mniej wydajna niż stronicowanie na żądanie, dlatego jest rzadziej stosowana [4].

2.1.3. Stronicowana segmentacja na żądanie

Stronicowana segmentacja na żądanie [3] jest połączeniem technik segmentacji na żądanie i stronicowania na żądanie. Proces jest dzielony na pewną liczbę segmentów, z których każdy jest dzielony na strony. Adres wirtualny składa się z numeru segmentu i przemieszczenia wewnątrz segmentu, które z kolei jest podzielone na numer strony i przemieszczenie wewnątrz niej. Z każdym procesem jest skojarzona tablica segmentów i kilka tablic stron. Translacja adresu wirtualnego na fizyczny obejmuje trzy etapy:

1. Z tablicy segmentów odczytywany jest adres tablicy stron. Indeksem określającym pozycję w tablicy segmentów jest numer segmentu zawarty w adresie wirtualnym.

2. Z tablicy stron odczytywany jest adres bazowy ramki, w której strona się znajduje. Jako indeks elementu tablicy stron jest używany numer strony.

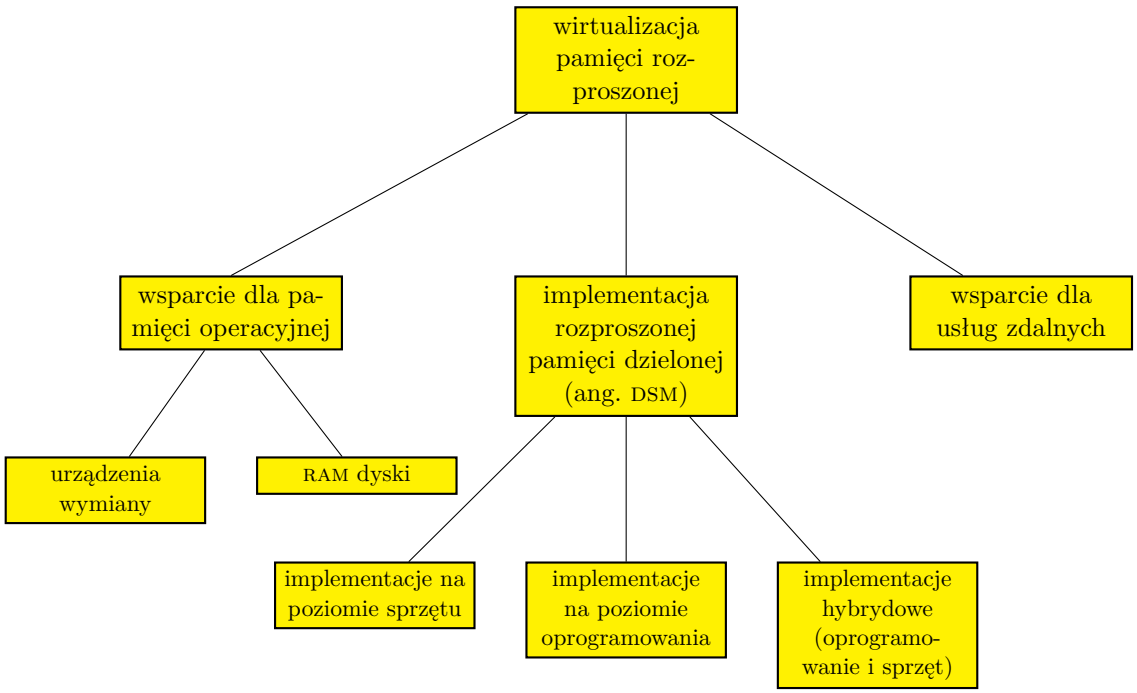
3. Numer segmentu i numer strony są zastępowane w adresie wirtualnym przez adres bazowy ramki. Otrzymany wynik to adres fizyczny.

Segmentacja stronicowana na żądanie jest efektywniejsza niż segmentacja na żądanie dzięki temu, że można do niej zastosować algorytmy wymiany stosowane w stronicowaniu na żądanie. Jest również pozbawiona fragmentacji zewnętrznej. Pozwala także lepiej dostosować ochronę

do struktury procesu, niż ma to miejsce w stronicowaniu na żądanie, a nawet zastosować nowe mechanizmy ochrony, takie jak ochrona pierścieniowa (ang. *ring protection*). Do wad segmentacji stronicowanej zalicza się fragmentację wewnętrzną („dziedziczoną" po stronicowaniu) oraz złożony mechanizm translacji adresów.

2.2. Wirtualizacja pamięci w systemach wielokomputerowych

Wirtualizacja pamięci operacyjnej w systemach wielokomputerowych polega na agregacji pamięci lokalnych poszczególnych węzłów celem uzyskania puli RAM, która byłaby dostępna dla każdego systemu komputerowego wchodzącego w skład multikomputera [15]. Rysunek 2.1 pokazuje możliwe zastosowania wirtualizacji pamięci rozproszonej [22].



Rysunek 2.1: Zastosowania wirtualizacji pamięci w systemach multikomputerowych

2.2.1. Wsparcie dla pamięci operacyjnej

Jednym z podstawowych zadań mechanizmów szeregujących systemów operacyjnych pracujących w klastrze jest dążenie do zrównoważenia obciążenia procesorów poszczególnych węzłów multikomputera. To działanie nie musi obejmować optymalizacji wykorzystania pamięci operacyjnej. Skutkiem tego część węzłów może dysponować większymi ilościami nieużywanej pamięci (ang. *idle*), niż pozostałe, gdzie może tego zasobu brakować. Wirtualizacja pozwala na agregację tej wolnej pamięci i uczynienie z niej wspólnego zasobu, z którego mogą korzystać wszystkie węzły multikomputera według potrzeb. W tym podrozdziale przedstawione są sposoby realizacji tej idei.

Urządzenia wymiany

Rozmiary zbiorów danych roboczych (ang. *data working sets*) aplikacji uruchamianych w multikomputerze mogą znacznie przekraczać ilość dostępnej RAM pojedynczego węzła. Rozwiąza-

niem tego problemu może być użycie opisanej w Podrozdziale 2.1 pamięci wirtualnej, ale opóźnienia wynikające z czasu transferu pamięć - dysk mogą być nieakceptowalne. W takim przypadku jako urządzenia wymiany można użyć wolnej pamięci operacyjnej wybranych lub wszystkich węzłów multikomputera. Według [25] istnieje kilka możliwości skonstruowania takiego urządzenia wymiany:

- jako część programu użytkownika - wymiana danych, niekoniecznie podzielonych na strony byłyby zadaniem procesu użytkownika, co pozwoliłoby dostosować algorytmy wymiany do wzorców zachowania aplikacji; to rozwiązanie wymaga dodatkowego nakładu pracy od programisty tworzącego programy (brak wirtualizacji) lub odpowiedniego wsparcia ze strony translatora,

- jako podprogramy przydziału/zwalniania pamięci zapisane w bibliotece - to rozwiązanie jest bardziej przenośne między poszczególnymi platformami sprzętowymi, wymaga mniej pracy ze strony programisty aplikacji, ale również potrzebuje wsparcia ze strony systemu operacyjnego w postaci odpowiednich wywołań systemowych pozwalających na zarządzanie tablicą stron oraz odpowiednich procedur obsługi błędu strony,

- jako mechanizm wymiany w trybie użytkownika - istnieją systemy operacyjne, takie jak Mach, które pozwalają aplikacjom użytkownika samodzielnie przemieszczać strony między RAM, a przestrzenią wymiany; to rozwiązanie nie daje się zastosować w przypadku wszystkich systemów operacyjnych,

- w postaci sterownika urządzenia - mechanizm wymiany jest implementowany jako sterownik działający w trybie jądra systemu, ale nie będący jego integralną częścią; rozwiązanie nie wymaga modyfikacji aplikacji użytkownika ani kodu jądra, jest też bardziej przenośne niż to, które zostanie zaprezentowane jako następne[2],

- jako mechanizm wymiany będący integralną częścią jądra systemu - to rozwiązanie może być bardzo wydajne, ale też trudno przenośnie między różnymi architekturami[2],

- jako rozwiązanie sprzętowe - to rozwiązanie może okazać się najmniej przenośne spośród zaprezentowanych, ale też najbardziej wydajne; polega na zastosowaniu sprzętowej jednostki MMU, która wycofywałaby strony pamięci nie na dysk, lecz do pamięci operacyjnej innego węzła multikomputera; przykładem takiego rozwiązania jest system SHRIMP.

Należy zauważyć, że implementacje na poziomie systemu operacyjnego i sprzętu są przezroczyste dla programów użytkownika, co oznacza, że nie jest wymagane przystosowanie ich do działania z nowym urządzeniem wymiany. Poniżej znajduje się przegląd wybranych implementacji takich urządzeń wymiany. Są to rozwiązania zrealizowane jako programowe sterowniki działające w trybie jądra systemu.

Model zdalnej pamięci (ang. *The Remote Memory Model*) został zaproponowany przez Douglasa Comera i Jamesa Griffionena [32]. Model ten zakłada wykorzystanie dedykowanych węzłów wielokomputera jako serwerów zdalnej pamięci (ang. *remote memory servers*). Słowo „dedykowany" oznacza w tym kontekście, że komputery pełniące rolę takich serwerów nie pełnią żadnej dodatkowej funkcji. Węzły będące serwerami zdalnej pamięci powinny być wyposażone w pamięć operacyjną i dyskową, których rozmiary znacznie przewyższają pojemność podobnych urządzeń w maszynach klienckich, choć możliwe są konfiguracje, w których serwery pamięci zdalnej są maszynami bezdyskowymi. Proponowany model nie zawiera żadnych założeń

[2]Te rozwiązania powinny w większym stopniu tolerować błędy oraz być odpornymi na nieprawidłowe dane niż implementacje działające w trybie użytkownika.

co do architektury i oprogramowania węzłów-klientów, poza tym, że muszą posiadać połączenie z serwerami pamięci zdalnej, zmodyfikowany system operacyjny pozwalający na współpracę z takimi serwerami oraz, że korzystają z pamięci wirtualnej zaimplementowanej jako stronicowanie na żądanie. Jako łącze może być zastosowana sieć LAN. Część węzłów multikomputera może dodatkowo być wykorzystana jako serwery plików. Rozwiązanie zaproponowane przez Comera i Griffionena ma szereg zalet:

- **Dodatkowa pamięć** - klienci, którzy współdzielą pamięć oferowaną przez serwery zdalnej pamięci mogą pozyskać dodatkową wolną przestrzeń dla aplikacji o dużych zbiorach roboczych danych.
- **Dowolnie duża pojemność pamięci** - serwery pamięci zdalnej mogą obsługiwać wielu klientów korzystając ze swojej pamięci operacyjnej. Jeśli jej pojemność zostanie przekroczona na skutek żądań klientów, to serwer może zastosować mechanizm pamięci wirtualnej celem obsłużenia tych żądań. Ponieważ serwery pamięci zdalnej mogą współpracować z wieloma dyskami i posiadać dużą RAM, to zasoby pamięci jakie mogą udostępniać klientom są teoretycznie nieograniczone. Dodatkowo serwer pamięci zdalnej może obsługiwać klientów heterogenicznych, bez względu na stosowany przez nich rozmiar strony, czy inne aspekty ich architektury/oprogramowania.
- **Współdzielenie danych** - serwer pamięci zdalnej stanowi scentralizowany magazyn danych, co stwarza możliwość ich współdzielenia między klientami homogenicznymi. W najprostszym przypadku współdzielone dane mogą być przeznaczone tylko do odczytu. Możliwe jest także opracowanie mechanizmów współdzielenia pozwalających na realizację komunikacji między procesami za pomocą pamięci dzielonej.
- **Odciążenie serwera plików** - jako urządzenie wymiany w systemie wielokomputerowym może zostać użyty serwer NFS [4, 5, 12]. Takie rozwiązanie powoduje jednak spadek jego wydajności jako serwera plików, ponieważ żądania dostępu do plików, które najczęściej mają charakter sekwencyjny, przeplatane są z bardzo częstymi żądaniami dostępu do przestrzeni wymiany zorganizowanej na dysku, które mają charakter swobodny (ang. *random*). Obsługa tych żądań wymaga częstego przestawiania głowicy dysku magnetycznego, wpływając tym samym negatywnie na czas realizacji żądań do „zwykłych" plików, dla których częste przestawienie głowicy jest niekorzystne. Zastosowanie osobnych serwerów pamięci zdalnej odciąża serwery plików, ponieważ ruch związany z wymianą stron nie jest kierowany do nich. Ze względu na to, że strony pamięci klientów są przechowywane głównie w RAM serwerów pamięci zdalnej, to czas dostępu do ich zawartości jest niezależny od ich położenia, chyba że konieczne jest użycie pamięci wirtualnej.
- **Semantyka pamięci zdalnej** - pamięć lokalna klientów oraz zasób pamięci dostarczany przez serwer pamięci zdalnej powinny być rozpatrywane całościowo pod względem podatności na błędy. Jeśli któryś z elementów zawiedzie, cały system może ulec awarii. W najprostszym przypadku, rozpatrywanym przez autorów modelu, serwer pamięci zdalnej nie zwielokrotnia stron klienta, nie utrzymuje trwałych ich kopii ani nie posiada innych mechanizmów, które łagodziłyby lub zapobiegały skutkom jego awarii. Możliwe jest jednakże zastosowanie tych mechanizmów, jeśli zaszłaby taka konieczność.

Wadą proponowanego rozwiązania jest brak skalowalności RAM serwera pamięci zdalnej. Jej rozmiar jest ograniczony do wielkości pamięci wirtualnej tego serwera, co przy dużej liczbie klientów obciążonych wieloma aplikacjami o dużych wymaganiach pamięciowych może okazać się gorszym rozwiązaniem od lokalnej pamięci wirtualnej. Wprawdzie w multikomputerze może

znajdować się wiele węzłów pełniących rolę serwerów pamięci zdalnej, ale pojedynczy klient jest obsługiwany tylko przez jeden z tych serwerów i podlega jego ograniczeniom pamięciowym. Jest to więc typowa architektura klient-serwer. Autorzy modelu zaproponowali konfigurację zawierając łańcuch serwerów pamięci zdalnej, która łagodzi skutki przepełnienia pamięci pojedynczego serwera, ale nie eliminuje tego problemu całkowicie. W tej konfiguracji klient kontaktuje się z serwerem pamięci zdalnej, którego rolę pełni bezdyskowa stacja robocza, która korzysta z innego serwera pamięci zdalnej wyposażonego w dysk pełniący rolę urządzenia wymiany. Wprowadzenie dwustopniowego serwera pamięci zdalnej wydłuża czas obsługi żądań pochodzących od klientów. Warto również zauważyć, że autorzy modelu nie przedstawili metody współdzielenia stron, które są przeznaczone zarówno do odczytu, jak i do zapisu przez wielu klientów. Wskazali jedynie, że taka możliwość istnieje. W prototypowej implementacji modelu pamięci zdalnej komunikacja między klientami i serwerami pamięci zdalnej odbywa się przy pomocy dwóch protokołów: XPP (ang. *Xinu Paging Protocol*) oraz NAFP (ang. *Negative Acknowledgement Fragmentation Protocol*), przy czym XPP jest protokołem wyższej warstwy bezpośrednio osadzonym na protokole NAFP, który jest z kolei osadzony na protokole UDP/IP. XPP obsługuje cztery rodzaje komunikatów: pobranie strony pamięci (ang. *page fetch request*), przechowanie strony pamięci (ang. *page store request*), tworzenie procesu (ang. *create request*), zakończenie procesu (ang. *terminate request*). Dwa ostatnie służą serwerowi pamięci zdalnej do zarządzania jego pamięcią operacyjną. Komunikaty pobierania/przechowywania mogą przenosić strony o różnych rozmiarach tym samym umożliwiając serwerowi pamięci zdalnej obsługę klientów heterogenicznych. Aby zapewnić niezawodność i prawidłowy porządek transmisji (ang. *in-order*) XPP stosuje numerowanie sekwencyjne komunikatów (każdy komunikat oprócz swego numeru sekwencyjnego zawiera również numer sekwencyjny poprzednika), potwierdzenia (ang. *acknowledgement*), pomiar upływu czasu (ang. *timeouts*) i retransmisje zagubionych pakietów. NAFP obsługuje fragmentację pakietów XPP. Dzieli je na mniejsze, nie przekraczające rozmiaru MTU (ang. *Maximal Transmission Unit*) sieci przez którą dokonywana jest transmisja danych. Podobnie jak XPP stosuje sekwencyjną numerację pakietów, ale dotyczy to tylko bieżącej grupy pakietów i zamiast potwierdzania odbioru każdego z nich przez drugą stronę transmisji stosuje negatywne potwierdzenia, tzn. potwierdzenie wysyłane jest nadawcy tylko wtedy, gdy pakiet został zagubiony i zostało to odkryte przez odbiorcę. Serwer pamięci zdalnej używa tablicy haszującej i algorytmu podwójnego haszowania [33] do lokalizacji stron klientów w jego pamięci operacyjnej. Każda aktywna pozycja tej tablicy zawiera informacje o tylko jednej stronie klienta, w skład których wchodzi identyfikator strony i lista lub zakres bloków pamięci, które przechowują tę stronę. Daną wejściową dla algorytmu podwójnego haszowania jest uporządkowana trójka danych przesyłana przez klienta, która zawiera unikatowy identyfikator klienta, identyfikator procesu oraz identyfikator strony. Alokacja bloków pamięci odbywa się w chwili otrzymania przez serwer żądania przechowania strony od klienta. Serwer oprócz wspomnianej tablicy haszującej dla danych utrzymuje również drugą tablicę haszującą dla procesów. Kiedy klient tworzy nowy proces, to powiadamia o tym serwer pamięci zdalnej wysyłając odpowiedni komunikat XPP. Odebranie tego komunikatu powoduje, że serwer dodaje do tablicy procesów nową pozycję dla nowo utworzonego procesu. Kiedy proces ten wysyła żądania przechowania stron, to czas przybycia każdej z nich jest zapamiętywany wraz z jej zawartością w tablicy danych jako znacznik czasowy (ang. *timestamp*). Kiedy serwer otrzyma informację o zakończeniu procesu, to również odnotowuje jej czas przybycia i zapamiętuje go w tablicy procesów, tym samym unieważniając (ang. *invalidate*) wszystkie strony procesu, ale nie zwalniając przydzielonych im bloków pamięci. Zwolnienie pamięci następuje w trybie leniwej dealokacji, kiedy serwer odnajdzie unieważnioną stronę w wyniku wystąpienia kolizji w haszowaniu lub w ramach odzyskiwania nieużytków (ang. *garbage collection*), które jest wykonywane przez wątek działający w tle. Oprogramowanie serwera pamięci zdalnej uruchamiane jest w przestrzeni użytkownika.

Serwery Pamięci dla Multikomputerów są implementacją modelu serwerów zdalnej pamięci opracowaną dla systemu wielokomputerowego Intel iPSC/860 [34]. Autorzy tej implementacji zaproponowali pewne modyfikacje oryginalnej koncepcji Griffionena i Comera, które jednak nie zostały w niej w całości zrealizowane. Serwery pamięci zdalnej są w tym rozwiązaniu traktowane jako warstwa mieszcząca się między RAM węzłów, a urządzeniami wymiany (dyskami). Ta warstwa pełni rolę pamięci podręcznej dla stron procesów użytkowników. Proponowane zmiany obejmują sposób wyboru węzłów pełniących rolę serwerów pamięci oraz realizację wsparcia dla pamięci dzielonej, nazywanej przez autorów wirtualną pamięcią współdzieloną. W opisywanym rozwiązaniu węzły multikomputera zostały podzielone na dwa rozłączne zbiory - węzły obliczeniowe i serwery pamięci. Podział dynamiczny mógłby być dokonywany przez system operacyjny na podstawie informacji dostarczanych przez użytkownika o zapotrzebowaniu uruchamianych aplikacji na zasoby lub o potrzebnych węzłach obliczeniowych. Dzięki takiemu podejściu serwery pamięci stałyby się skalowalne. Wsparcie dla pamięci dzielonej polegałoby na interpretacji przez klientów i serwery pamięci atrybutów nadawanych stronom dzielonym przez podsystem rozproszonej pamięci dzielonej i na przypisaniu im priorytetów, które decydowałyby o kolejności pozyskiwania stron przez klienta z serwerów pamięci. Wyróżniono cztery rodzaje takich atrybutów:

- **niedostępna** (ang. *no-access*),
- **oryginał do odczytu** (ang. *read-owner*) - strona tylko do odczytu, której właścicielem jest bieżący procesor,
- **kopia do odczytu** (ang. *read-copy*) - strona tylko do odczytu, której właścicielem nie jest bieżący procesor,
- **do zapisu** (ang. *writable*) - właścicielem strony jest bieżący procesor i jej zawartość może być zarówno odczytywana, jak i zapisywana.

Istniejąca implementacja nie obejmuje opisanych wyżej propozycji. Składa się ona z klienta zrealizowanego na poziomie systemu operacyjnego oraz serwera pracującego w trybie użytkownika. Serwer stosuje tablice haszujące do zarządzania stronami klienta znajdującymi się w jego pamięci. Do zalet proponowanego rozwiązania należy zaliczyć próby uczynienia serwerów pamięci skalowalnymi i realizacji wsparcia dla pamięci dzielonej. Przedstawiony w artykule [34] prototyp nie zawiera żadnego z tych rozwiązań. Nie posiada również mechanizmów tolerowania błędów, ale zwiększa szybkość działania pamięci wirtualnej.

Remote Memory Pager (RMP) jest wirtualnym urządzeniem wymiany, które wykorzystuje dostępną RAM węzłów multikomputera jako przestrzeń wymiany i posiada mechanizmy zapewniające tolerowanie błędów [35]. Klient RMP jest zaimplementowany jako proces systemu operacyjnego. Nie jest on jednak częścią jądra, a oprogramowaniem systemowym uruchomionym w trybie użytkownika. Każdy węzeł multikomputera może być zarówno klientem, jak i serwerem RMP. Pojedynczy węzeł-serwer może obsługiwać wielu klientów. Dla każdego z nich uruchamiana jest nowa instancja procesu serwera. Jeśli na węźle, na którym pracuje serwer zostanie uruchomiona aplikacja o dużym zbiorze roboczym (ang. *working set*) [3, 4], to proces serwera powiadamia klienta, że ilość dostępnej RAM przeznaczonej na przestrzeń wymiany uległa zmniejszeniu. Jeśli nowy rozmiar przestrzeni wymiany nie spełnia potrzeb klienta, to może on znaleźć inny serwer, o większej pojemności lub podjąć decyzję o użyciu lokalnego dysku do zorganizowania przestrzeni wymiany. RMP może realizować jeden z trzech scenariuszy tolerowania błędów:

- **odzwierciedlanie** (ang. *mirroring*) - jest najprostszą formą nadmiarowości - część serwerów RMP jest przeznaczona do przechowywania kopii stron umieszczonych w innych

serwerach; jeśli zostanie wykryta awaria serwera podstawowego, to jego rolę natychmiast może przejąć serwer zapasowy; wadą rozwiązania jest duże zużycie dostępnej pamięci rozproszonej,

- **parzystość** (ang. *parity*) - strony są dzielone na grupy; dla wszystkich stron należących do grupy wyliczana jest za pomocą operatora XOR strona zawierająca bity parzystości; jeśli nastąpi odwołanie do strony przechowywanej na serwerze, który uległ awarii, to można ją odtworzyć ponawiając operację XOR na wszystkich pozostałych stronach należących do grupy i na stronie z bitami parzystości; wadą tego rozwiązania jest konieczność dodatkowego przesyłania stron do serwera parzystości (klient → serwer → serwer parzystości); schemat ten nie zabezpiecza przed jednoczesną awarią więcej niż jednego serwera oraz błędami, które nie są wykrywane przez kontrolę parzystości,

- **rejestrowanie parzystości** (ang. *parity logging*) - strona z bitami parzystości jest wyliczana przez klienta i wysyłana do jednego z serwerów parzystości (serwer RMP, który przechowuje strony z bitami parzystości); zwykłe strony są wysyłane przez klienta do serwerów RMP wyznaczanych metodą rotacyjną (ang. *round robin policy*); w ten sposób tworzone są nowe grupy parzystości; jeśli strona, która była już używana ponownie zostanie wysłana do serwera RMP, to będzie zaliczała się do nowej grupy; jeśli wszystkie strony należące do grupy parzystości znajdą się w nowej grupie, to serwery RMP będą mogły zwolnić pamięć przeznaczoną na ich poprzednie wersje.

Remote Memory Pager zawiera wydajne mechanizmy zapewniające tolerowanie błędów, ale może również pracować w trybie, w którym nie są one używane. RMP zwiększa szybkość działania pamięci wirtualnej, ale nie oferuje żadnego rozwiązania zapewniającego skalowalność serwerów. W przypadku przekroczenia pojemności serwera klient może jedynie migrować swoje strony do innego serwera RMP lub na lokalny dysk.

Nswap jest pseudourządzeniem blokowym, które służy jako urządzenie wymiany dla systemu Linux 2.4 i jest podobne w działaniu do RMP [36]. W przeciwieństwie do RMP, nie zawiera żadnego mechanizmu gwarantującego tolerowanie błędów. Jednym z założeń Nswap jest to, że każdy węzeł multikomputera może być klientem lub serwerem, ale nie jednocześnie. Rola jaką pełni w danej chwili węzeł zależy od ilości bieżąco dostępnej wolnej pamięci operacyjnej. Klient Nswap jest zaimplementowany jako moduł jądra systemu i może zostać załadowany lub usunięty z systemu w czasie jego działania. Aby jednak jądro mogło współpracować z Nswap musi zostać wcześniej odpowiednio zmodyfikowane. Modyfikacja ta obejmuje wyeksportowanie jednego z symboli jądra dla modułów, skompilowanie kodu źródłowego i ponowne uruchomienie systemu po dokonaniu zmiany. Klient korzysta z dwóch struktur danych, tablicy „IP Table" i zdublowanego odwzorowania wymiany (ang. *shadow swap map*), aby zarządzać zdalną wymianą stron. Zdublowane odwzorowanie wymiany zawiera 16-bitowe pozycje dla każdej z wymienianych stron. Każda pozycja (ang. *swap slot*) zawiera identyfikator serwera, na którym się znajduje wymieniona strona, liczbę przeskoków (ang. *hop_count*), która ogranicza liczbę migracji jakim może podlegać strona oraz zapobiega sytuacjom hazardowym podczas migracji, znacznik czasowy (ang. *time_stamp*) pozwalający zidentyfikować strony, które nie są już używane i bit zajętości (ang. *in_use bit*), pozwalający synchronizować operacje współbieżne dotyczące zawartości pozycji odwzorowania. Tablica „IP Table" zawiera informacje na temat stanu poszczególnych serwerów Nswap. Każda pozycja tablicy „IP Table" zawiera adres serwera, ilość pamięci udostępnianej przez ten serwer oraz deskryptor gniazda sieciowego (ang. *socket*) służącego do połączenia z tym serwerem. Informacja o ilości udostępnianej pamięci jest wysyłana przez serwer klientom poprzez rozgłaszanie (ang. *broadcasting*) z użyciem protokołu UDP. Ponieważ protokół ten nie gwarantuje niezawodności dostarczenia informacji, to dane o dostępności pamięci na

poszczególnych węzłach zapisanych w „IP Table" nie muszą odzwierciedlać poprawnie stanu rzeczywistego. Serwer Nswap, podobnie jak klient, został zaimplementowany w postaci modułu jądra. Do jego zadań należy zarządzanie pulą pamięci, która jest przeznaczona na przestrzeń wymiany. Serwer monitoruje system zarządzania pamięcią węzła, na którym jest uruchomiony i jeśli wykryje zwiększenie obciążenia pamięci, to zmniejsza przestrzeń wymiany i powiadamia o tym klientów. Jeśli nowa ilość pamięci nie wystarcza na potrzeby klienta, który używa serwera, to może on podjąć decyzję o migracji swoich stron do innego serwera lub użyciu lokalnego dysku jako przestrzeni wymiany. Klient, który żąda przesłania strony wysyła do serwera metadane dotyczące tej strony, które obejmują: identyfikator klienta, numer pozycji (ang. *slot_number*), liczbę przeskoków (ang. *hop_count*) oraz znacznik czasu (ang. *time_stamp*). Podczas czynności odzyskiwania nieużytków serwer ustala na podstawie znacznika czasu, które strony mogą zostać usunięte. Te strony mogą być również porzucone (ang. *drop*) podczas migracji do innego serwera. Komunikacja między klientami i serwerami dotycząca wymiany stron przeprowadzana jest za pomocą dedykowanego protokołu, który może być osadzony na protokole TCP/IP lub UDP/IP. Protokół ten wyróżnia pięć rodzajów komunikatów. Komunikat typu PUTPAGE służy do umieszczenia strony w serwerze Nswap. Jego dopełnieniem jest komunikat GETPAGE pozwalający pobrać stronę z serwera. Komunikat typu PUNTPAGE służy do wysłania żądania migracji stron przez serwer do innego serwera. Komunikat UPDATE pozwala serwerowi powiadomić klientów o zmianie lokalizacji ich stron, która nastąpiła po przeprowadzeniu migracji. Komunikaty INVALIDATE służą klientowi do powiadomienia serwera przechowującego przed migracją jego strony, że może usunąć ich kopie. Opisane urządzenie wymiany jest szybsze od dysku, ale tylko wtedy, gdy obciążenie systemu nie jest duże i kiedy podstawowym protokołem komunikacji między klientami i serwerami jest UDP/IP. W innych warunkach lokalny dysk jest bardziej wydajny. Ponieważ serwer Nswap jest oprogramowaniem pracującym na poziomie jądra systemu operacyjnego, to na 32-bitowych platformach klasy PC może nastąpić spadek jego wydajności, jeśli na przestrzeń wymiany przydzieli więcej niż 1 GiB RAM. Wynika to ze sposobu zarządzania przez system Linux wirtualną przestrzenią adresową na tego typu platformach - strony pamięci leżące powyżej 1 GiB pamięci wirtualnej przeznaczone są na potrzeby aplikacji użytkownika i nie są automatycznie odwzorowywane w przestrzeni adresowej jądra. Nie można również odwzorować wszystkich tych stron równocześnie w przestrzeni jądra [37]. Błędy w pracy modułu serwera, których przyczyną może być np. odebranie przez moduł serwera źle skonstruowanego pakietu, mogą doprowadzić do destabilizacji pracy całego węzła. Dodatkowo takie błędy są trudne do usunięcia, gdyż mechanizmy debugowania domyślnie dostępne na poziomie jądra systemu Linux mogą okazać się niewystarczające do lokalizacji usterki powodującej wystąpienie tego błędu.

Distributed Anemone (DA) [38] jest wirtualnym urządzeniem blokowym zoptymalizowanym na potrzeby przestrzeni wymiany stron i przeznaczonym dla systemu Linux. Według autorów może ono być również wykorzystane jako RAM dysk, choć ta możliwość nie została przez nich sprawdzona. Komponenty DA, podobnie jak w przypadku Nswap, zostały zaimplementowane jako oprogramowanie działające w trybie jądra systemu operacyjnego. Jedyny z elementów odróżniających to rozwiązanie od Nswap jest użycie dedykowanego protokołu do komunikacji między serwerami i klientami DA o nazwie RMAP (ang. *Remote Memory Access Protocol*). Jest to niskopoziomowy protokół, który korzysta wyłącznie z ramek sieci Ethernet i adresów fizycznych węzłów komputera (MAC). Ogranicza to zastosowanie DA do multikomputerów składających się z węzłów należących do jednej sieci, ale zwiększa szybkość działania Distributed Anemone. RMAP posiada mechanizmy zapewniające niezawodne dostarczanie pakietów (ang. *Reliable Packet Delivery*), kontrolę przepływu (ang. *Flow-Control*), fragmentację i składanie pakietów (ang. *Fragmentation and Reassembly*). Oprócz komunikatów dotyczących bezpośrednio wymiany stron RMAP wykorzystywany jest również do przesyłania informacji związanych z zarządzeniem strukturą DA. Wraz z komunikatami zawierającymi żądania odnośnie stron przesyłane są

również (ang. *piggyback*) dane o dostępnej przestrzeni wymiany, którą dysponują serwery. Dodatkowo zarówno klienci, jak i serwery co 10 sekund wysyłają komunikaty o swojej dostępności. To pozwala klientom na odkrycie nowych serwerów DA w sieci oraz wykluczenie określonego serwera lub grupy serwerów. Serwery wykorzystują tę informację do usuwania danych należących do klientów, którzy zakończyli swe działanie. Podobnie jak w Nswap klienci i serwery są zaimplementowani jako moduły jądra. W przypadku DA nie są wymagane żadne modyfikacje jądra Linuksa, aby możliwa była współpraca z tymi modułami. Każdy węzeł multikomputera może dynamicznie zmieniać swoją rolę i w zależności od zapotrzebowania być klientem lub serwerem DA. Moduł klienta jest widoczny dla reszty systemu jako kolejne urządzenie wymiany, które jest obsługiwane za pomocą tych samych wywołań systemowych, co inne urządzenia blokowe [39–41]. W przeciwieństwie do fizycznych urządzeń dyskowych DA nie wymaga szeregowania żądań, co dodatkowo skraca czas ich przetwarzania. Klient DA utrzymuje również listę serwerów DA, które przechowują jego strony. Jeśli któryś z używanych przez niego serwerów zgłosi brak wolnego miejsca w pamięci na nowe strony klient może, bazując na informacjach, które otrzymał wcześniej, wybrać inny serwer. W ten sposób strony klienta mogą być przechowywane na kilku serwerach, co zapewnia skalowalność rozwiązania. Informacje o serwerach ze stronami przechowywane są przez klienta w tablicy haszującej stosującej metodę łańcuchową do rozwiązywania kolizji [33]. Ten sposób adresowania serwerów powoduje dodatkowe zużycie pamięci po stronie klienta i jego efektywność spada wraz ze zwiększającą się liczbą używanych serwerów. Mechanizm adresowania został zmieniony w nowej wersji DA nazwanej MemX [42]. Nowa wersja wykorzystuje drzewa pozycyjne (ang. *radix tree*). MemX zawiera również modyfikacje, które pozwalają mu na współpracę z maszyną wirtualną Xen. Prawidłowe wyłączenie serwera DA nie powoduje załamania systemów obsługiwanych węzłów klienckich, ale DA nie posiada mechanizmów, które zapobiegają skutkom awarii węzłów lub sieci. Implementacja serwera jako oprogramowania pracującego w trybie jądra systemu może powodować te same problemy, które były wymienione w opisie Nswap.

Innym urządzeniem wymiany dostępnym dla systemu Linux jest HPBD (ang. *High Performance Block Device*) [43]. Serwery pamięci w tym rozwiązaniu pracują w przestrzeni użytkownika, natomiast oprogramowanie klienckie jest modułem jądra tworzącym wirtualne urządzenie blokowe. HPBD zostało zaprojektowane w celu zbadania wydajności wymiany stron przy użyciu sieci InfiniBand [10]. Implementując je wykorzystano mechanizm bezpośredniego dostępu do zdalnej pamięci RAM oferowany przez tę sieć. Zgodnie z oczekiwaniami twórców urządzenie jest szybsze niż rozwiązania korzystające z sieci Gigabit Ethernet, a nawet szybsze od urządzeń komunikujących się przez sieć InfiniBand, ale z użyciem stosu TCP/IP.

RAM dyski

Dyski RAM (ang. RAM *disks*) określane również w literaturze polskiej RAM dyskami (zapożyczenie z języka angielskiego) są wirtualnymi urządzeniami blokowymi, które dostępne są aplikacjom użytkownika w ten sam sposób, co urządzenia fizyczne, na których został osadzony system plików. RAM dysk jest rozwiązaniem stosowanym w systemach jednoprocesorowych, najczęściej celem przyspieszenia operacji wykonywanych na plikach. Podobną rolę pełni ono w wielokomputerach, gdzie może być zbudowane nie tylko w oparciu o dostępną pamięć pojedynczego węzła, ale także w oparciu o wolną pamięć rozproszoną całego systemu.

Przykładem takiego wykorzystania nieużywanej, rozproszonej pamięci jest The Network RamDisk (NRD) przeznaczony dla systemów operacyjnych Digital Unix i Linux [26]. Rozwiązanie to składa się z pojedynczego węzła, który jest klientem i kilku serwerów, które go obsługują. Klient jest modułem jądra systemu operacyjnego, który widoczny jest dla aplikacji użytkownika jako wirtualne urządzenie blokowe. Utrzymuje on tablicę, która odwzorowuje adres bloku na

serwer, w którym ten blok jest przechowywany. Serwer jest oprogramowaniem działającym na poziomie użytkownika. Węzły, na których pracują serwery i klient, są równoprawne. Oznacza to, że mogą one wykonywać inne aplikacje użytkownika oprócz oprogramowania NRD. W takim przypadku serwery NRD muszą rywalizować o pamięć operacyjną z innymi procesami i w wyniku tej rywalizacji mogą być zmuszone do zwolnienia części jej obszaru przeznaczonego na bloki dyskowe. Jeśli dojdzie do takiej okoliczności, to serwer może powiadomić klienta o zaistniałej sytuacji, a ten może podjąć decyzję o migracji bloków dyskowych na inny serwer. W NRD zastosowano również schematy tolerowania błędów znane z RMP. Okazało się, że rejestrowanie parzystości, które sprawdzało się w przypadku urządzeń wymiany, powoduje spadek wydajności sieciowego RAM dysku poniżej akceptowalnego poziomu. Spowodowane jest to mniejszą granulacją danych, które w przypadku urządzenia blokowego są przesyłane w grupach (blokach) po maksymalnie osiem sektorów. Pojedynczy sektor w badanych systemach ma rozmiar 512 lub 1024 bajtów. Taki podział powoduje dodatkowy ruch w sieci przy stosowaniu rejestracji parzystości. Autorzy zaproponowali wiec inny schemat, który nazwali podręczną pamięcią parzystości (ang. *parity cache*). Polega on na tym, że klient utrzymuje po jednym buforze dla każdego z serwerów, z którymi współpracuje (serwer zapamiętujący bloki parzystości również jest uwzględniony). Każdy z tych buforów może zapamiętać określoną liczbę bloków. Dopiero w momencie, kiedy wszystkie zostaną zapełnione klient tworzy z nich grupy parzystości i dla każdej grupy liczby bloki parzystości. Następnie wszystkie bloki (danych i parzystości) są wysyłane do przydzielonych im serwerów. W tym schemacie pojawia się trudność aktualizacji bloków parzystości w przypadku, kiedy zostanie zmieniona zawartość któregoś z bloków danych, które wchodzą w skład określonych grup parzystości. Rozwiązaniem jest wprowadzenie adaptatywnej polityki podręcznej pamięci parzystości (ang. *adaptive parity caching policy*). Jej podstawą jest obserwacja, że zgodnie z zasadą lokalności przestrzennej aplikacje użytkownika korzystają z bloków danych, które są do siebie przyległe, tzn. posiadają kolejne adresy. Taki ciąg sekwencyjnych odwołań (ang. *references*) do bloków nazywany jest strumieniem danych (ang. *data stream*). Bloki należące do określonego strumienia najczęściej należą także do tego samego pliku i jest na nich wykonywana ta sama operacja (odczyt lub zapis). Grupy parzystości mogą więc być tworzone na bazie strumieni danych związanych z modyfikacją bloków. W systemach wielozadaniowych każda z uruchomionych aplikacji może tworzyć własny strumień danych, który może przeplatać się ze strumieniami innych wykonywanych programów. Klient NRD utrzymuje kilka pamięci podręcznych, w których umieszcza bloki na podstawie ich przynależności od określonego strumienia. Jeśli nie pasują one do żadnego z zapamiętanych częściowo strumieni, to trafiają do pamięci podręcznej oznaczonej jako „V" (od angielskiego słowa VICTIM). Jeżeli dojdzie do zapełnienia przynajmniej jednej pamięci podręcznej, to umieszczone w niej bloki są traktowane jako grupa i jest dla nich liczony blok parzystości, a następnie każdy z bloków jest przesyłany do odpowiedniego serwera. To postępowanie może jednak prowadzić do sytuacji, w której część pamięci podręcznych będzie zawierała mniej bloków niż jest potrzebne do ich wypełnienia, a nowe bloki należące do związanych z nimi strumieni nie będą napływać. Aby jej uniknąć, każda pamięć podręczna, która nie zostanie wypełniona w czasie potrzebnym do dwukrotnego opróżnienia pamięci „V", uzupełniana jest blokami zawierającymi zera. Pamięć „V" może być zapełniana wolno, ale ten proces zawsze się kończy. Adaptatywna polityka podręcznej pamięci parzystości zmniejsza obciążenie sieci, a jednocześnie jest równie efektywna jak rozwiązania z RMP, które zastąpiła. NRD jest wirtualnym urządzeniem magazynującym dane, które może współpracować z wieloma systemami plików. Jest rozwiązaniem bardziej ogólnym niż urządzenia wymiany i bardziej wydajnym niż dyski magnetyczne. Pozwala również korzystać z kilku mechanizmów tolerowania błędów. Nie jest ono także uzależnione od konkretnego typu sieci komputerowej, na bazie której zbudowany jest multikomputer. Współpracuje z heterogenicznymi multikomputerami. Wadą tego rozwiązania jest brak skalowalności serwera.

2.2.2. Rozproszona pamięć dzielona

Jednym z podstawowych modeli komunikacji między procesami w systemach uniprocesorowych i wieloprocesorowych ze wspólną pamięcią jest komunikacja z użyciem pamięci dzielonej. Ten model jest wygodny dla programisty tworzącego aplikacje użytkownika. W systemach wielokomputerowych nie istnieje pamięć wspólna, a podstawowym modelem komunikacji między procesami lub wątkami działającymi na osobnych maszynach jest przekazywanie komunikatów (ang. *message passing*). Możliwe jest jednak stworzenie abstrakcji, która pozwoli na użycie w wielokomputerze komunikacji przez pamięć dzieloną. To rozwiązanie nazywa się rozproszoną pamięcią współdzieloną (ang. *Distributed Shared Memory* - DSM) [5, 9, 12, 13]. Rysunek 2.1 wymienia trzy sposoby realizacji takiego rozwiązania: sprzętowo, w postaci oprogramowania i hybrydowo, czyli łącznie za pomocą sprzętu i oprogramowania. Tę klasyfikację można bardziej uszczegółowić [13]:

- sprzęt,
- oprogramowanie,
  - system operacyjny,
    - w jądrze systemu,
    - poza jądrem systemu,
  - biblioteka podprogramów,
  - wsparcie kompilatora,
- hybryda sprzętu i oprogramowania.

Granice między poszczególnymi kategoriami nie są wyraźnie zaznaczone, więc niektóre implementacje mogą należeć równocześnie do więcej niż jednej z nich. Od poziomu, na którym zostanie zaimplementowana DSM, zależą takie jej parametry, jak konfiguracja architektoniczna systemu i organizacja danych współdzielonych. Architektura multikomputera, w którym jest implementowana DSM wpływa na jej wydajność i skalowalność. Decydująca dla tych czynników jest budowa węzłów multikomputera (liczba procesorów, organizacja ich pamięci podręcznej) oraz rodzaj i topologia sieci łączącej te węzły. Budowa węzłów jest ważna w różnym stopniu dla poszczególnych implementacji DSM, generalnie jednak ma duże znaczenie w przypadku rozwiązań sprzętowych, ale niewielkie lub żadne w przypadku implementacji programowych (ang. *software implementations*). Większość DSM zrealizowanych programowo została zaimplementowana w środowisku wielokomputerowym opartym o sieć Ethernet. Niektóre z nich są jednak niezależne od rodzaju sieci i mogą zyskać na wydajności pracując w środowiskach wyposażonych w szybsze wersje połączeń sieciowych. Zaawansowane topologie sieciowe są domeną sprzętowych implementacji DSM. W obu przypadkach topologia sieci ma duże znaczenie dla algorytmu współdzielenia danych stosowanego w DSM, ponieważ determinuje koszt i możliwości transmisji rozgłoszeniowej (ang. *broadcast*) oraz wielopunktowej (ang. *multicast*). Organizacja danych współdzielonych obejmuje dwa parametry: ich strukturę i ziarnistość jednostki spójności (ang. *granularity of coherence unit*). Pod względem struktury dane mogą być nieuporządkowane lub być zmiennymi specjalnych typów (np. obiektami). Wśród jednostek spójności można wyróżnić słowa, linie pamięci podręcznej, strony i złożone struktury danych. Implementacje sprzętowe DSM używają zazwyczaj nieuporządkowanych obiektów danych, takich jak np. linie pamięci podręcznej. Część implementacji DSM na poziomie oprogramowania stosuje zmienne specjalnych typów. Dzięki temu możliwe jest dostosowanie wielkości jednostek współdzielenia do lokalności odwołań generowanych przez aplikacje użytkownika. Implementacje DSM, których podstawą są mechanizmy pamięci wirtualnej stosują większe jednostki współdzielenia, takie jak

strony i segmenty. Innym kryterium klasyfikacji systemów rozproszonej pamięci współdzielonej jest algorytm współdzielenia. Podział w tym przypadku zależny jest od istnienia lub nie wielu kopii danych i praw dostępu jakie dla nich obowiązują. Uwzględniając te czynniki można wyróżnić cztery klasy algorytmów współdzielenia:

- SRSW - pojedynczy czytelnik/pojedynczy pisarz (ang. *Single Reader/Single Writer*),
  - bez migracji,
  - z migracją,
- MRSW - wielu czytelników/jeden pisarz (ang. *Multiple Reader/Single Writer*),
- MRMW - wielu czytelników/wielu pisarzy (ang. *Multiple Reader/Multiple Writer*).

Od klasy algorytmu współdzielenia zależą takie parametry DSM jak typ zarządzania rozproszoną pamięcią współdzieloną oraz model i polityka spójności. Zarządzanie DSM ma na celu utrzymanie spójności danych w systemie. Może ono być centralne lub rozproszone. Scentralizowane zarządzanie DSM jest łatwiejsze w implementacji, ale podatne na awarie - węzeł zarządzający stanowi pojedynczy punkt załamania systemu (ang. *single point of failure*). Rozproszone zarządzanie DSM może być statyczne lub dynamiczne. W statycznym zarządzaniu rolę zarządców pełnią tylko określone węzły multikomputera i ta rola nie może być im odebrana. W zarządzaniu dynamicznym każdemu z węzłów multikomputera może zostać, w zależności od potrzeb, przypisana rola zarządcy. Rozproszone zarządzanie DSM eliminuje „wąskie gardła" (ang. *bottlenecks*) w systemie i zwiększa jego skalowalność. Rozproszenie zarządzania DSM jest bezpośrednio związane z rozproszeniem katalogu utrzymującego informacje o współdzielonych danych. Takie katalogi mogą być zorganizowane wówczas w listy lub drzewa. Model spójności (ang. *consistency model*) określa prawidłowy porządek, w jakim widziane są odwołania pojedynczego procesu do pamięci, przez pozostałe procesy w systemie wielokomputerowym. Modele silnej spójności (ang. *strong consistency*) są wygodne w użyciu dla programisty aplikacji użytkowych, ale powodują obciążenia łącza komunikacyjnego. Modele złagodzonej spójności (ang. *relaxed consistency*) zapewniają lepszą wydajność, ale wymagają większego nakładu pracy od programistów. Do modeli spójności zalicza się [5]:

- spójność ścisłą,
- spójność sekwencyjną,
- liniowość,
- spójność przyczynową,
- spójność FIFO,
- spójność słabą,
- spójność zwalniania,
- spójność wejścia.

Spójność ścisła (ang. *strict consistency*) zdefiniowana jest następująco: „Każde czytanie zmiennej współdzielonej zwraca wartość odpowiadającą wynikowi ostatnio wykonanej na tej zmiennej operacji pisania." Jeśli jeden z procesów dokona zmiany wartości zmiennej współdzielonej, to ta zmiana widoczna jest natychmiast dla wszystkich pozostałych procesów. Jest to najbardziej intuicyjny dla programistów model spójności.

Spójność sekwencyjna (ang. *sequential consistency*) jest słabszym modelem spójności niż spójność ścisła. Zdefiniowana jest następująco: „Wynik dowolnego wykonania jest taki sam, jak gdyby operacje (czytania i pisania) wszystkich procesów na pamięci danych były wykonane w pewnym porządku jedna po drugiej, przy czym operacje każdego poszczególnego procesu wystąpiły w tym samym ciągu w kolejności określonej przez program." Termin „pamięć danych" oznacza w tym przypadku obszar rozproszonej pamięci współdzielonej przez procesy. Spójność sekwencyjna dopuszcza przeplot operacji pochodzących od różnych procesów, ale ten przeplot musi być taki sam dla wszystkich procesów korzystających z tych samych zmiennych. Z definicji spójności sekwencyjnej wynika, że nie wymaga ona określenia globalnego czasu.

Liniowość (ang. *linearizability*) zakłada, że każdej operacji przypisywany jest znacznik czasowy generowany na podstawie globalnego zegara, który pracuje z ograniczoną dokładnością. Zdefiniowana jest ona następująco: „Wynik dowolnego wykonania jest taki sam, jak gdyby operacje (czytania i pisania) wszystkich procesów na pamięci danych były wykonywane w pewnym porządku jedna po drugiej, przy czym operacje każdego poszczególnego procesu wystąpiły w tym ciągu w kolejności określonej przez jego program. Ponadto, jeśli $zc_{OP1}(x) < zc_{OP2}(x)$, to operacja $OP1(x)$ powinna w tym ciągu poprzedzić $OP2(x)$." Symbol zc używany jest na oznaczenie znacznika czasu, OP na oznaczenie operacji, a x na oznaczenie zmiennej współdzielonej. Liniowość jest słabsza od ścisłej spójności, ale silniejsza od spójności sekwencyjnej.

Spójność przyczynowa (ang. *casual consistency*) zdefiniowana jest następująco: „Zapisy potencjalnie powiązane przyczynowo muszą być widziane przez wszystkie procesy w takim samym porządku. Zapisy współbieżne mogą być na różnych maszynach oglądane w różnej kolejności."

Spójność FIFO (ang. FIFO *consistency*) określa, że „Zapisy wykonane przez jeden proces są oglądane przez wszystkie inne procesy w porządku, w którym powstały, lecz zapisy pochodzące od różnych procesów mogą być przez różne procesy oglądane w różnym porządku". Spójność FIFO nazywana jest również spójnością procesora (ang. *processor consistency*).

Spójność słaba (ang. *weak consistency*) wymaga użycia zmiennych synchronizacji do zapewnienia, że wyniki końcowe sekcji krytycznych procesów dokonywanych na zmiennych współdzielonych zostaną rozesłane do wszystkich kopii tej zmiennej w pamięci danych. Spójność ta posiada trzy cechy:

1. Dostępy do zmiennych synchronizacji, skojarzonych z pamięcią danych, są spójne sekwencyjnie.

2. Działanie na zmiennej synchronizacji jest zabronione do czasu, aż wszystkie poprzednie zapisy zostaną wszędzie ukończone.

3. Na jednostkach danych zabrania się wykonywania operacji czytania i pisania dopóty, dopóki nie zostaną wykonane wszystkie poprzednie operacje na zmiennych synchronizacji.

Spójność zwalniania (ang. *release consistency*) stanowi udoskonalenie spójności słabej polegające na zastosowaniu dwóch osobnych operacji synchronizacji, aby możliwe było rozpoznanie, czy proces wchodzi, czy opuszcza sekcję krytyczną. Dzięki temu rozróżnieniu możliwe jest wyeliminowanie niektórych zbędnych działań, które występują w przypadku spójności słabej. Operacja nabycia (ang. *acquire*) informuje, że proces wchodzi do sekcji krytycznej, a operacja zwolnienia (ang. *release*), że ją opuszcza. Spójność zwalniania jest zachowana, jeśli:

1. Przed wykonaniem operacji czytania lub zapisania danych współdzielonych muszą się pomyślnie zakończyć wszystkie poprzednie nabycia dokonane przez proces.

2. Zanim będzie wolno wykonać zwolnienie, w procesie należy zakończyć wszystkie poprzednie operacje czytania i pisania.

3. Dostępy do zmiennych synchronizacji wykazują spójność FIFO (nie jest wymagana spójność sekwencyjna).

Spójność wejścia (ang. *entry consistency*) również posługuje się operacjami nabycia i zwolnienia, ale wymaga, aby z każdą zwykłą zmienną współdzieloną skojarzyć odpowiadającą jej zmienną synchronizacji. Spójność wejścia jest zachowana, jeśli:

1. Nabycie zmiennej synchronizacji nie może w procesie nastąpić dopóty, dopóki nie zostaną wykonane wszystkie aktualizacje strzeżonych danych współdzielonych dotyczących tego procesu.

2. Przed udzieleniem procesowi zezwolenia na dostęp do zmiennej synchronizacji w trybie wykluczającym, należy zagwarantować, że żaden inny proces nie będzie utrzymywał tej zmiennej - nawet w trybie niewykluczającym.

3. Po wykonaniu wykluczającego dostępu do zmiennej synchronizacji żaden inny proces nie może wykonać do niej niewykluczającego dostępu bez uzgodnienia tego z właścicielem zmiennej.

Polityka spójności (ang. *coherence policy*) definiuje w jaki sposób zarządzać kopiami danych, których oryginały uległy modyfikacji. Stosowane są zazwyczaj dwa scenariusze: aktualizacja kopii lub ich unieważnienie (ang. *invalidate*). Pierwsze rozwiązanie stosuje się w przypadku, kiedy współdzielone jednostki danych są drobnoziarniste (ang. *fine grain*), ponieważ koszt ich uaktualnienia jest porównywalny z kosztem unieważnienia. Unieważnienie stosowane jest w przypadku gruboziarnistych (ang. *coarse-grain*) współdzielonych jednostek danych.

Sprzętowe rozproszone pamięci dzielone

Działanie sprzętowych rozproszonych pamięci dzielonych (ang. *hardware* DSM) zazwyczaj bazuje na odpowiednio zmodyfikowanych protokołach spójności podręcznych pamięci procesorów w systemach wieloprocesorowych typu UMA. W takich komputerach średni czas dostępu do każdej z lokacji we wspólnej pamięci operacyjnej jest taki sam dla każdego z procesorów. Tą cechą nie dysponują systemy wieloprocesorowe typu NUMA, których konstrukcja ma wiele podobieństw z budową systemów wielokomputerowych (wspólna przestrzeń adresowa, ale oparta o indywidualne pamięci operacyjne procesorów połączonych siecią wewnętrzną). Istniejące sprzętowe implementacje DSM są stosowane w obu typach systemów. Można podzielić je na co najmniej trzy grupy [13]:

1. CC-NUMA - (ang. *Cache Coherent Non-Uniform Memory Architecture*) - architektura pamięci o niejednolitym czasie dostępu i spójnych pamięciach podręcznych,

2. COMA (ang. *Cache-Only Memory Architecture*) - architektura pamięci bazująca wyłącznie na pamięci podręcznej,

3. RMS - (ang. *Reflective Memory System Architecture*) - architektura pamięci bazująca na systemie pamięci odzwierciedlającej.

Współdzielona wirtualna przestrzeń adresowa w sprzętowych implementacjach DSM o architekturze CC-NUMA jest statycznie rozdysponowana między poszczególne węzły systemu. Czasy dostępów są różne w zależności od tego, czy dostęp do określonego obszaru pamięci jest wykonywany przez procesor lokalny, czy należący do innego węzła. Używając tego typu DSM należy zadbać, aby odwołania do pamięci lokalnej były częstsze niż odwołania do pamięci zdalnej.

Katalogi DSM mogą mieć różną budowę, od pełnych odwzorowań do list lub drzew. Najczęstszą polityką spójności w tego rodzaju DSM jest unieważnianie. Stosowane są modele spójności oferujące złagodzoną spójność.

Sprzętowe implementacje DSM oparte o architekturę COMA dostarczają współdzielonej pamięci rozproszonej zorganizowanej w postaci pamięci podręcznej drugiego poziomu. Żadna jednostka danych współdzielonych nie ma przypisanej lokacji domowej (ang. *home memory location*), a jej kopie mogą istnieć w pamięciach wielu węzłów systemu jednocześnie. Rozmieszczenie danych w pamięci współdzielonej jest dynamicznie dostosowywane do zachowania aplikacji użytkownika, więc nie występuje problem braku lokalnych odwołań znany z architektury CC-NUMA, ale za to pojawia się nadmiarowość danych. Cechą charakterystyczną architektury COMA są hierarchiczne sieci komunikacyjne, na których bazują jej implementacje. Te sieci upraszczają wyszukiwanie danych i wymianę (zastępowanie) bloków pamięci podręcznej.

Jeśli DSM jest zrealizowana w oparciu o architekturę RMS, to każdy z węzłów systemu może wyznaczyć fragment swojej pamięci lokalnej, który zostanie włączony do wspólnej dla wszystkich węzłów wirtualnej przestrzeni adresowej. Aktualizacja danych wykonywana jest sprzętowo, a dane te odznaczają się zazwyczaj małą ziarnistością. Spójność współdzielonych regionów pamięci węzłów jest zapewniana przez pełną replikację. Polega ona na rozpropagowaniu każdej zmiany danych za pomocą transmisji rozgłoszeniowej lub wielopunktowej, dlatego koszt operacji zapisu w architekturach RMS jest wysoki. Architektury te powinny więc być stosowane dla aplikacji, które nie generują dużej liczby zapisów. Pamięć współdzielona oparta o architekturę RMS nazywana jest też pamięcią zwierciadlaną (ang. *mirror memory*).

W porównaniu z implementacjami programowymi, sprzętowe implementacje DSM wykazują mniejsze opóźnienia w komunikacji, minimalizują efekt fałszywego współdzielenia oraz migotania.

Zjawisko fałszywego współdzielenia (ang. *false sharing*) pojawia się, gdy jednostka współdzielenia (np. strona pamięci) zawiera dwie lub większą liczbę niepowiązanych ze sobą zmiennych, które są używane przez różne procesy mogące wykonywać się na różnych węzłach systemu [9]. Mimo, że poszczególne zmienne mogą nie być przez te procesy współdzielone, to taka jednostka uznawana jest za współdzieloną. Problem fałszywego współdzielenia jest powiązany z zagadnieniem doboru ziarnistości jednostek współdzielenia.

Zjawisko migotania zostało opisane w Podrozdziale 2.1, ale w DSM ma inną przyczynę niż w przypadku systemów jednoprocesorowych. Są nią częste zapisy do tych samych zmiennych współdzielonych przez dwa lub większą liczbę węzłów systemu. W przypadku niektórych implementacji DSM (nie tylko sprzętowych) może to prowadzić do konieczności bardzo częstego przesyłania jednostki współdzielenia, która zawiera te zmienne, między dzielącymi ją węzłami [9].

Powstało szereg sprzętowych implementacji DSM. W przeciwieństwie do rozwiązań bazujących na oprogramowaniu są one częściej wdrażane w przemyśle, ale mniej popularne w środowiskach akademickich ze względu na ograniczoną możliwość modyfikacji.

Przedstawicielem rodziny sprzętowych DSM jest Memnet (*Memory as Network Abstraction*) [9, 13, 44]. Urządzenie będące podstawą tego systemu pełni dwie funkcje. Lokalnie jest sterownikiem dwuportowej pamięci, a w odniesieniu do multikomputera interfejsem pomiędzy pojedynczym węzłem, a siecią typu token ring, którą połączone są wszystkie węzły. Cała przestrzeń adresowa pamięci współdzielonej jest odwzorowana we wszystkich węzłach i składa się z 32 bajtowych fragmentów. Kiedy żądanie dostępu do pamięci, które zgłasza proces działający na jednym z węzłów nie może być zaspokojone lokalnie, to interfejs Memnet wysyła odpowiedni komunikat i blokuje procesor, od którego pochodziło żądanie. Komunikat jest przekazywany między kolejnymi węzłami multikomputera. Jeden z nich jest adresatem tej wiadomości. Jeśli dotrze ona do niego, to węzeł ten odpowiada modyfikując oryginalną zawartość komunikatu.

Komunikat jest usuwany z pierścienia, kiedy dotrze do węzła, który go wysłał. Po tym zdarzeniu następuje również zwolnienie zablokowanego procesora. Dzięki zastosowaniu sieci typu token ring, czas, po którym węzeł powinien otrzymać odpowiedź ma bezwzględną górną granicę, a przepustowość sieci nie maleje wraz z dołączaniem do niej nowych węzłów. Specyfika sieci pozwala również na zastosowanie ścisłej spójności. Komunikaty pojawiają się w poszczególnych węzłach zawsze w tej samej kolejności. Protokół spójności bazuje na unieważnieniach i podobny jest do protokołów podglądania magistrali (ang. *bus snooping*) jakie są stosowane do zapewnienia spójności lokalnych pamięci podręcznych procesora. Przykładem takiego protokołu jest MESI stosowany np. w procesorach Pentium firmy Intel [45]. Żądanie odczytu w Memnet jest zaspokajane przez pierwszy napotkany węzeł, który posiada ważną kopię danych, do których następuje odwołanie. W przypadku modyfikacji danych, które są skopiowane w innych węzłach multikomputera (ang. *non-exclusive copy*) po żądaniu zapisu wysyłane jest żądanie unieważnienia. Memnet rezerwuje część pamięci każdego węzła multikomputera na potrzeby przechowywania fragmentów pamięci dzielonej. Jeśli któryś z fragmentów zostanie usunięty z lokalnej pamięci węzła (ang. *eviction*), to gwarantowanym jest, że zostanie przechowany w innym węźle [46]. Zaletami Memnet są przewidywalny czas odpowiedzi na żądania węzłów, dzięki zastosowaniu sieci token ring oraz ścisła spójność. Do wad należy ograniczona skalowalność, spowodowana tym, że ilość pamięci przeznaczonej na katalog DSM opisujący stan każdego ze współdzielonych fragmentów pamięci rośnie proporcjonalnie do kwadratu liczby węzłów multikomputera.

System DASH (Directory Architecture for Shared Memory) jest sprzętową implementacją DSM opartą na architekturze CC-NUMA [9, 13, 47]. Katalog DSM jest zrealizowany w DASH sprzętowo. Każdy węzeł DASH (nazywany też klastrem) jest właścicielem takiej samej, pod względem wielkości, części pamięci jak pozostałe węzły i jest z nimi połączony siecią o topologii kratownicy (ang. *mesh*), w której stosowane jest trasowanie kanalikowe. Każde z połączeń w tej sieci jest dwukierunkowe (żądanie/odpowiedź). Węzły w DASH są wieloprocesorowe. Hierarchia pamięci w tym systemie ma cztery poziomy:

1. pamięć podręczna procesora,
2. pamięci podręczne pozostałych procesorów węzła (klastra),
3. węzeł (klaster) domowy - przechowuje określony blok pamięci, wraz z jego pozycją katalogu DSM,
4. zdalny węzeł (klaster) - węzeł, który oznaczony jest jako ten, który przechowuje kopię bloku.

Każdy z bloków może znaleźć się w jednym z trzech stanów: UNCACHED - żadna kopia bloku nie istnieje, CACHED - istniej jedna lub więcej niezmodyfikowanych kopii bloku, DIRTY - kopia bloku została zmodyfikowana w zdalnym węźle. Katalog DASH jest katalogiem DSM rozproszonym z pełnym odwzorowaniem (ang. *full-map*). Sprzętowy sterownik katalogu DASH stanowi interfejs pomiędzy węzłem, a siecią połączeń oraz odpowiedzialny jest za zachowanie spójności współdzielonych danych. Jeśli odwołanie do bloku nie może zostać spełnione lokalnie, to wysyłany jest komunikat z odpowiednim żądaniem do klastra domowego tego bloku. W zależności od stanu bloku, który jest zapisany w jego pozycji katalogu DSM, węzeł domowy podejmuje odpowiednie czynności, aby spełnić to żądanie. Żądanie odczytu dla bloku, który jest w stanie UNCACHED lub CAHED jest spełniane przez węzeł domowy. Jeśli blok znajduje się w stanie DIRTY, to takie żądanie zostanie przekazane węzłowi zdalnemu, który posiada aktualną kopię bloku. Ten węzeł odpowiada na żądanie oraz przesyła zawartość swojej kopii do klastra domowego. Jeżeli żądanie dotyczy zapisu bloku, który jest współdzielony, to węzeł domowy przed jego spełnieniem wysyła komunikaty typu punkt-punkt (ang. *point-to-point*) celem unieważnienia

wszystkich kopii zdalnych bloku. Z powyższego opisu wynika, że DASH stosuje model złagodzonej spójności - spójność wyjścia. System DASH jest w dużym stopniu skalowalny, jednakże ma złożoną strukturę i jest trudny do weryfikacji.

Innym przedstawicielem sprzętowej DSM jest SCI - Scalable Coherent Interface [13, 48–50]. SCI nie jest pełną implementacją DSM, ale standardem szybkiej sieci komputerowej, który posiada sprzętowe wsparcie dla realizacji rozproszonej pamięci dzielonej. Podstawowym rodzajem topologii sieci SCI jest pierścień (ang. *ring*), ale możliwa jest też budowa bardziej złożonych struktur za pomocą specjalizowanych przełączników sieciowych. Każda para węzłów SCI jest połączona dwoma jednokierunkowymi łączami. SCI pozwala stworzyć z pamięci operacyjnej wszystkich węzłów wspólną, adresowaną fizycznie pamięć dzieloną. Adres pojedynczej lokacji w takiej DSM jest 64-bitowy, przy czym 16 starszych bitów stanowi identyfikator węzła, a młodszych 48 bitów właściwy adres fizyczny komórki pamięci operacyjnej tego węzła. Ze względu na stosowaną topologię nie jest możliwa transmisja rozgłoszeniowa, co wyklucza możliwość zastosowania protokołów spójności pamięci podręcznej, które oparte są na podglądaniu magistrali. W zamian SCI (opcjonalnie) zapewnia rozwiązanie oparte na rozproszonym katalogu DSM zrealizowanym w postaci listy dwukierunkowej. Z każdym współdzielonym blokiem pamięci związana jest lista procesorów przechowujących jego kopię w swoich pamięciach podręcznych. Każdy procesor, który odwołuje się do określonego współdzielonego bloku pamięci jest dodawany na początek tej listy (ang. *head*). W przypadku żądania odczytu dane dla tego procesora nie są przesyłane bezpośrednio z pamięci operacyjnej, którą zajmuje współdzielony blok, ale z pamięci podręcznej procesora, który poprzedza go na liście. Procesor może zostać usunięty z listy, jeśli usuwa kopię współdzielonego bloku ze swojej pamięci podręcznej. Aby uniknąć uszkodzenia listy wprowadzono priorytety kolejności usuwania na wypadek, gdyby kilka procesorów próbowało przeprowadzić tę operację współbieżnie. Jako pierwszy usuwany jest z listy ten procesor, który znajduje się bliżej jej końca (ang. *tail*). W przypadku operacji zapisu procesor, który zażąda jej wykonania zostaje dodany na początku listy, a do pozostałych procesorów na tej liście wysyłany jest komunikat o unieważnieniu, który powoduje usunięcie ich z tej listy. SCI jest rozwiązaniem o dużym współczynniku skalowalności, ale posiada również pewne wady. Jedną z nich jest konieczność wykonywania operacji przeglądania listy dwukierunkowej, której koszt rośnie wraz z liczbą jej elementów. Komitet IEEE, który opracował standard SCI zaproponował jako rozwiązanie tego problemu zastosowanie innych niż lista dwukierunkowa struktur danych (np. drzewo) oraz łączenie żądań.

KSR1 to komercyjny, sprzętowy system DSM, który jest oparty na architekturze COMA [13]. Składa się on z hierarchicznie połączonych pierścieni węzłów. Mechanizm (ang. *engine*) pamięci KSR1 o nazwie ALLCACHE korzysta z pamięci podręcznych poszczególnych węzłów, omijając pamięć główną. Konsekwencją tego rozwiązania jest to, że żaden adres nie ma na stałe powiązanej z nim lokacji fizycznej. Jednostką przydziału pamięci jest strona o rozmiarze 16 KiB, ale jednostką współdzielenia i transferu między lokalnymi pamięciami podręcznymi jest 128 bajtowa podstrona. Mechanizm ALLCACHE odpowiedzialny za odnajdywanie i kopiowanie podstron przechowywanych w pamięciach podręcznych węzłów jest zrealizowany sprzętowo. Stosowanych jest kilka mechanizmów ALLCACHE, które zorganizowane są hierarchicznie. Najniższy poziom zajmują grupy oznaczonej jako ALLCACHE Group:0. Każda z tych grup składa się z mechanizmu oznaczonego jako ALLCACHE Engine:0 i przyporządkowanej mu lokalnej pamięci podręcznej. Każdy mechanizm ALLCACHE tego poziomu zawiera katalog DSM, który odwzorowuje adresy na lokacje w pamięci podręcznej procesorów należących do jego grupy. Grupy wyższych poziomów skonstruowane są z grup niższych poziomów, a ich katalogi DSM odwzorowują adresy na zbiory takich grup. Wszystkie grupy tworzą drzewo. KSR1 zapewnia sekwencyjną spójność danych za pomocą protokołu bazującego na unieważnieniach. Każda podstrona znajduje się w jednym z czterech stanów:

- wyłącznym (ang. EXCLUSIVE) - nie istnieje żadna kopia podstrony,
- niewyłącznym (ang. NON-EXCLUSIVE) - pamięć jest właścicielem podstrony, która może mieć jedną lub więcej kopii w systemie,
- skopiowanym (ang. COPY) - podstrona jest ważną (ang. *valid*) kopią podstrony z innej pamięci podręcznej,
- unieważnionym (ang. INVALID) - podstrona jest przydzielona, ale jej zawartość jest unieważniona.

Cechą, która wyróżnia KSR1 wśród innych implementacji DSM opartych na architekturze COMA jest wykorzystanie lokalności równoległości - zjawiska polegającego na istnieniu pewnych wspólnych wzorców odwołań do pamięci generowanych przez spokrewnione wątki. Wadą KSR1 jest jej hierarchiczna budowa wprowadzająca opóźnienia w działaniu oraz złożona struktura zaimplementowana całkowicie w sprzęcie.

Na architekturze COMA oparty jest również system DDM (ang. *Data Diffusion Machine*) zbudowany z procesorów wyposażonych w pamięć lokalną, które połączone są hierarchiczną magistralą zamiast sieci [6, 13, 51]. Dane przechowywane w pamięciach procesorów mają unikatowy w obrębie całej struktury adres wirtualny, ale nie są przypisane na stałe do żadnej fizycznej lokacji. Zamiast tego są przemieszczane do pamięci lokalnej procesora, który zgłosi na nie zapotrzebowanie. Migracja danych jest przezroczysta dla oprogramowania użytkowego i nadzorowana przez sprzętowe sterowniki uporządkowane hierarchicznie. Na najniższym szczeblu tej hierarchii znajdują się sterowniki pamięci lokalnej, która zawiera fragment obrazu globalnej wirtualnej przestrzeni adresowej. Sterownik pamięci, pamięć lokalna i związana z nią pamięć podręczna oraz procesor są ze sobą połączone za pomocą magistrali lokalnej. Ta z kolei jest połączona za pośrednictwem sterownika katalogu z magistralą wyższego poziomu. Sterowniki katalogów połączone są ze sobą poprzez sterowniki katalogów kolejnego, wyższego poziomu. Katalogi zawierają informacje o bieżącym stanie wszystkich danych zgromadzonych w podległych im pamięciach lokalnych. Rolą sterowników jest pośredniczenie między elementami systemu, które znajdują się niżej i wyżej od nich w hierarchii. Jeśli żądanie procesora może zostać spełnione lokalnie, to jest ono natychmiast realizowane, ale w przypadku konieczności dostępu do zdalnych pamięci sterowniki odpowiedzialne są za przesłanie tego żądania w górę lub w dół hierarchii magistral. Lokalnie mogą być zrealizowane tylko żądania dotyczące odczytu lub zapisu danych znajdujących się w pamięci lokalnej, pod warunkiem, że modyfikowane dane nie są współdzielone. Współdzielenie w DDM oznacza, że istnieje kilka kopii danej, które mają ten sam co oryginalna dana adres wirtualny, ale różne adresy fizyczne. Jeśli dokonywany jest zapis danej współdzielonej, to najpierw kasowane są wszystkie jej kopie. Dopiero po wykonaniu tej czynności następuje właściwy zapis i zmodyfikowana informacja oznaczana jest jako niewspółdzielona. W przypadku odczytu danych nielokalnych ich kopia jest sprowadzana do pamięci lokalnej procesora, który zgłosił na nie zapotrzebowanie, a następnie obie instancje danej (oryginał i kopia) oznaczane są jako współdzielone. Możliwość istnienia nielokalnych kopii danych powoduje konieczność wprowadzenia protokołu gwarantującego zachowanie ich spójności. W DDM stosowane są dwa takie protokoły - jeden dla danych w pamięci, drugi dla danych w katalogu. Oba są modyfikacją protokołu MESI. Protokół dla danych w pamięci określa pięć stanów, w których mogą się one znajdować: **I** - dane są niepoprawne (nie istnieją), **E** - dane nie posiadają kopii, istnieją tylko lokalnie, **S** - istnieją nielokalne kopie danych, **W** - dane będą zapisywane lokalnie, zewnętrzne kopie danych są unieważniane, **R** - dane będą odczytywane lokalnie, oczekiwana jest odpowiedź z zewnątrz. Ostatnie dwa stany są stanami przejściowymi. Pamięci lokalne przechowują dane i informacje o ich stanie (tak jak klasyczne pamięci podręczne). Katalogi przechowują wyłącznie informacje o stanie danych w podległych im pamięciach.

Protokół związany z katalogami definiuje siedem stanów, w których mogą znaleźć się pozycje katalogowe. Pięć z nich jest takich samych, jak w protokole dla danych w pamięci. Dwa dodatkowe to: **ER** - dane nie są współdzielone i odpowiadają żądaniu odczytu pochodzącemu od innego procesora niż lokalny, **SR** - dane są współdzielone i odpowiadają żądaniu odczytu od procesora zewnętrznego. Oba protokoły gwarantują sekwencyjny model spójności, ale istnieje możliwość realizacji również słabszych form spójności. Na magistrali lokalnej mogą pojawić się transakcje dotyczące odczytu i zapisu danych z pamięci lokalnej. Na magistralach wyższego poziomu pojawiają się transakcje dotyczące odczytu danych, kasowania, powiadomienia o skasowaniu oraz powiadamiające o wartości odczytywanej danej. Ponieważ pojemność zarówno katalogów, jaki i pamięci lokalnych jest ograniczona, to stosowany jest algorytm zastępowania, który usuwa dane współdzielone. W pierwszej kolejności usuwane są te, które najdawniej nie były używane. Jeśli nie ma takich danych, to wybierane są dowolne dane współdzielone. Konstrukcja systemu DDM jest silnie oparta o zasady lokalności dostępu do danych. Większość odwołań do danych ma charakter lokalny. Informacje tworzone przez procesor są zapisywane w jego pamięci lokalnej, a dane znajdujące się w innych węzłach są do niej sprowadzane, tylko jeśli wystąpi taka konieczność. Transmisje z pamięci zdalnych, które są kosztowne, występują rzadko. Spójność danych przy zapisie jest zapewniana poprzez likwidację kopii modyfikowanej informacji, co jest mniej kosztowne niż ich aktualizacja.

Inne podejście do sprzętowej realizacji współdzielonej pamięci rozproszonej reprezentują systemy z pamięcią odzwierciedlającą (ang. RMS - *Reflective Memory Systems*). Przykładem takiego rozwiązania są produkty firmy Encore Computer Corporation [13, 52]. Węzły systemu z pamięcią odzwierciedlającą połączone są specjalnymi łączami, nazwanymi szyną RM. Pewne obszary lokalnych pamięci węzłów multikomputera są oznaczone jako współdzielone. Kontrolery pamięci lokalnych wykrywają zapis do takich obszarów i wysyłają poprzez szynę RM informację aktualizującą, zawierającą zmodyfikowane dane do wszystkich pozostałych węzłów. Ponieważ systemy RMS nie korzystają z pamięci podręcznych, a informacje o modyfikacjach są rozsyłane natychmiast, wymagana jest jedynie synchronizacja dostępu do danych współdzielonych. Nie ma konieczności korzystania z protokołów spójności.

Do systemów RMS zaliczany jest również Merlin [13, 53]. Interfejsy sprzętowe tego systemu śledzą zapisy do lokalnych pamięci węzłów multikomputera i wykonują kopie zapisywanych danych, wraz z ich adresem, jeśli są one współdzielone. Te kopie są natychmiast przesyłane przez sieć. Wszystkie odczyty dokonywane są na lokalnej pamięci węzła, która zawiera powielone obszary współdzielone. Podobnie jak w przypadku systemów firmy Encore konieczna jest jedynie synchronizacja, aby zachować spójność danych. Intencją autorów tego rozwiązania było stworzenie systemu rozproszonej pamięci dzielonej dla systemów heterogenicznych, ale Merlin nie zapewnia konwersji formatów danych. Współdzielone obszary pamięci lokalnych węzłów multikomputera określane są statycznie, nie ma możliwości określenia ich w sposób dynamiczny.

Rozproszone pamięci dzielone zrealizowane na poziomie oprogramowania

W porównaniu z rozwiązaniami sprzętowymi, implementacje DSM na poziomie oprogramowania są mniej wydajne, ale łatwiejsze w opracowaniu i modyfikacji. Realizacje programowej DSM można sklasyfikować według warstwy oprogramowania, na której w znaczącym stopniu są oparte. Do tych warstw zaliczają się [13]:

1. jądro systemu operacyjnego,
2. oprogramowanie systemowe działające poza jądrem,
3. mechanizmy wbudowane w języki programowania,

4. biblioteki podprogramów włączane do programu użytkownika na etapie konsolidacji lub podczas wykonania.

W większość istniejących implementacji jedna z tych warstw jest podstawową dla realizacji DSM, natomiast pozostałe mogą pełnić rolę pomocniczą. Przykładowo, DSM oparta na mechanizmach wbudowanych w język programowania może być wspierana przez mechanizm stronicowania systemu operacyjnego.

Mirage [9, 12, 13, 46, 54] jest przykładem DSM zrealizowanej na poziomie jądra systemu operacyjnego o nazwie Locus. Rozproszona pamięć dzielona w tym systemie wykorzystywania jest nie tylko jako środek komunikacji, ale umożliwia również migrację procesów i plików. Jest ona zbudowana na modelu MRSW z silną spójnością. Mirage bazuje na stronicowanej segmentacji. Segment pamięci współdzielonej dołączany jest do pamięci procesu jako obszar z prawem wyłącznie do odczytu lub do odczytu i zapisu. Każdy segment ma swój węzeł-właściciela, który jest jego twórcą i zarządza nim oraz węzeł-użytkownika, który przez określony czas jest w posiadaniu aktualnej kopii segmentu. Węzły będące właścicielami określane są mianem bibliotek (ang. *library site*), a węzły-użytkownicy nazywani są użytkownikami czasowymi (ang. *clock site*). Wszystkie żądania odczytu i zapisu danych znajdujących się na stronach segmentu są gromadzone w bibliotece, która zarządza wykonaniem tych operacji. Zapisy są wykonywane sekwencyjnie (w kolejności w jakiej nadeszły), a odczyty gromadzone są we wsady (ang. *batch*). Przed dokonaniem modyfikacji strony wszystkie jej kopie w systemie są unieważniane. Aby ograniczyć zjawisko migotania każdy z węzłów może wykonywać operacje na stronach współdzielonego segmentu przez okres czasu, którego długość określona jest wartością parametru Δ. Ta wartość może być dobierana statycznie lub dynamicznie. Od niej zależy skuteczność ochrony przed migotaniem. Ten mechanizm jest zapożyczony z klasycznej pamięci wirtualnej [4], gdzie parametr Δ służy do wyznaczania zbioru roboczego (ang. *working set*) procesu, czyli stron, które muszą w tym samym czasie razem przebywać w pamięci operacyjnej, aby nie wystąpiło zjawisko migotania. W Mirage został zmodyfikowany tak, jak to zostało opisane wyżej oraz uzupełniony o wywołanie systemowe o nazwie „yield", które pozwala zwolnić strony współdzielonego segmentu przed upływem czasu określonego przez Δ. Dzięki temu węzeł, który czeka na wykonanie swoich operacji na współdzielony stronach może rozpocząć je wcześniej. Inną cechą Mirage, która redukuje zjawisko migotania oraz zmniejsza liczbę wykonywanych transmisji jest możliwość podwyższania (ang. *upgrade*) i obniżania (ang. *downgrade*) uprawnień dotyczących przeprowadzania operacji na stronach. Jeśli węzeł posiada prawa do odczytu strony, to może podwyższyć swoje uprawnienia i wykonać zapis na jej lokalnej kopii, która ze względu na sposób zarządzania zapisami w Mirage (MRSW) jest aktualna. Podobnie, jeśli węzeł obniży swoje uprawnienia i zrezygnuje z zapisu, to do przeprowadzenia operacji odczytu wystarczy mu lokalna, aktualna kopia strony. Zaletą DSM Mirage jest przezroczystość (ang. *transparency*) dla aplikacji użytkownika oraz zastosowanie mechanizmów redukujących zjawisko migotania. Niestety obie te cech nie są w pełni zrealizowane - aplikacje użytkownika muszą używać wywołania „yield", co zmniejsza przezroczystość rozwiązania i wymaga wsparcia ze strony kompilatora, a skuteczność mechanizmu zmniejszania częstości występowania migotania może być osłabiona przez zły dobór wartości parametru Δ.

Mechanizm rozproszonej pamięci dzielonej w systemie Clouds jest przykładem implementacji DSM bazującej na elementach systemu operacyjnego umieszczonych poza jądrem systemu [9, 12, 13, 44, 46, 55, 56]. Clouds jest rozproszonym systemem operacyjnym, którego budowa jest oparta na modelu obiektowym. Obiekty w tym systemie są pasywne i służą do przechowywania względnie dużej ilości danych. Korzystają z nich wątki, których zadaniem jest przetwarzanie informacji zgromadzonych w obiektach. Obie abstrakcje tworzone są z użyciem elementów dostarczanych przez jądro systemu nazwane Ra. Obiekty są współdzielone na zasadzie migracji. Jeśli węzeł multikomputera potrzebuje informacji zawartych w obiekcie znajdującym się w in-

nym węźle, to może go sprowadzić do siebie korzystając z metod (usług) dostarczanych przez kontrolera DSM. Kontroler ten (DSMC) jest zaimplementowany jako oprogramowanie systemowe działające poza jądrem. Spójność danych w obiektach jest zapewniana przez protokół umożliwiający cztery tryby dostępu do segmentów wchodzących w skład obiektu: tryb odczytu, tryb słabego odczytu (ang. *weak read*), tryb odczytu i zapisu oraz tryb pusty (ang. *none*). Tryb odczytu nie zapewnia wyłącznego dostępu do danych, ale gwarantuje, że nie ulegną one zmianie na skutek wystąpienia zapisu podczas przeprowadzania operacji odczytywania. Tryb słabego odczytu nie daje takiej gwarancji. W trybie pustym segment może migrować do innego węzła. Jedynie tryb zapisu i odczytu gwarantuje wyłączny dostęp do danych. Zamknięcie danych wewnątrz obiektu i zastosowanie opisanych trybów dostępu umożliwia wydajną pracę systemu.

Linda [5, 9, 13, 57] i Orca [5, 13, 57] są językami programowania, które realizują mechanizmy DSM. W przypadku Lindy współdzielonymi jednostkami danych są krotki (ang. *tuple*) adresowane zawartością. Tworzą one logicznie współdzieloną pamięć asocjacyjną dostępną dla wszystkich węzłów. Zawartość krotki jest niezmienna. Aby zmodyfikować dane w krotce należy ją usunąć, dokonać modyfikacji i ponownie ją wprowadzić do przestrzeni współdzielonych krotek. Ten mechanizm zmian wartości krotek zapobiega naruszeniu spójności danych, ale w przypadku aplikacji wykonujących intensywnie zapisy może prowadzić do spadku wydajności. Orca jest środowiskiem programowania przeznaczonym dla systemów rozproszonych i multikomputerowych, w skład którego wchodzi język programowania oparty na paradygmacie obiektowym, kompilator tego języka i środowisko wykonania. Obiekty w języku Orca są pasywne. Ich metody, nazwane operacjami, są aktywowane przez sekwencyjne procesy. Każda operacja obiektu jest wykonywana w sposób niepodzielny i spójny sekwencyjnie. Orca umożliwia również synchronizację według warunku. Warunek jest realizowany za pomocą zmiennej logicznej (ang. *boolean*) nazwanej dozorem (ang. *guard*). Replikacja obiektów uzależniona jest od częstości wykonywania operacji zapisu oraz odczytu i dokonywana jest automatycznie przez środowisko wykonawcze. Spójność sekwencyjna obiektów zwielokrotnionych jest zapewniana przez mechanizm aktualizacji, którego działanie uzależnione jest od środków komunikacji udostępnianych przez system operacyjny (komunikacja typu punkt-punkt, wielopunktowa lub rozgłoszeniowa).

Munin [13, 46, 58], IVY [9, 12, 13, 44, 46, 59], Midway [13, 46, 60], Agora [13, 61], Mermaid [9, 13, 46] oraz TreadMarks [62] są systemami DSM opartymi o system bibliotek łączonych dynamicznie lub statycznie, które tworzą środowisko wykonawcze dla programów. Munin udostępnia programiście aplikacji szereg typów, które służą do tworzenia zmiennych współdzielonych. Sposób zapewnienia spójności danych dzielonych zależny jest od użytego typu zmiennej. Na poziomie języka programowania typy zmiennych współdzielonych opisane są przez adnotacje, które następnie przekształcane są przez preprocesor do postaci rozpoznawanej przez środowisko wykonawcze. W procesie tworzenia mechanizmu DSM udział bierze również zmodyfikowany program konsolidujący, który tworzy wydzielony segment pamięci dla zmiennych współdzielonych. Munin w wersji prototypowej nie jest wspierany przez kompilator, który zapewniałby większą kontrolę typów niż preprocesor oraz pozwalałby na większą automatyzację tworzenia zmiennych współdzielonych. Autorzy [13] proponują również dodanie do oprogramowania Munin możliwości dynamicznego wykrywania przez środowisko wykonawcze typów zmiennych na podstawie wzorców ich użycia. To zwiększyłoby elastyczność systemu. Innym usprawnieniem proponowanym przez autorów wymienionego artykułu jest wsparcie sprzętowe, którego celem byłaby poprawa wydajności. W systemie IVY jednostką współdzielenia są strony pamięci o rozmiarze 1 KiB. Tworzą one współdzielony fragment w obszarze pamięci procesu. Dzięki zastosowanym w IVY mechanizmom odwzorowującym wszystkie takie fragmenty zorganizowane są w pojedynczą przestrzeń adresową. Mechanizmy odwzorowujące stanowią częścią zarządcy odwzorowania. Do jego zadań należy zapewnienie spójności danych współdzielonych. IVY zapewnia silną spójność dzięki zastosowaniu algorytmów MRSW opartych na unieważnianiu przy zapisie. Strony,

które przeznaczone są do odczytu mogą być powielone, natomiast strony, które są modyfikowalne nie mogą mieć kopii i rezydują w pamięci węzła, który jest ich właścicielem. Utrzymywanie informacji o właścicielach stron jest kolejnym zadaniem wykonywanym przez zarządcę odwzorowania. Każdy właściciel strony ma pole blokady (ang. *lock field*) używane do synchronizacji żądań dostępu do strony oraz listę innych węzłów, które są w posiadaniu kopii tej strony. Zarządca odwzorowania może być scentralizowany lub rozproszony. W ostatnim przypadku rozproszenie może być statyczne lub dokonywane dynamicznie, w trakcie pracy systemu. Mermaid zawiera rozszerzenia koncepcji IVY na multikomputerowe środowiska heterogeniczne. Zaimplementowany jest w postaci biblioteki dołączanej do programów korzystających z DSM. Składa się z trzech komponentów: modułu zarządzającego wątkami, modułu zarządzającego pamięcią dzieloną oraz modułu odpowiedzialnego za zdalne operacje. Spójność danych współdzielonych zapewniana jest poprzez użycie protokołu unieważniania przy zapisie w połączeniu z komunikacją rozgłoszeniową. Mermaid umożliwia korzystanie z DSM w multikomputerach heterogenicznych, ale jego wersja prototypowa nakłada szereg ograniczeń co do typów współdzielonych danych. Agora jest również implementacją DSM przeznaczoną dla systemów heterogenicznych. Współdzielenie obejmuje definiowane przez programistę struktury danych, wraz z ich funkcjami dostępowymi (ang. *access-functions*). Definicje te opisane są w języku wysokiego poziomu, o prostej składni, którego kompilator generuje kod dla środowiska wykonawczego Agory i dla wspieranych języków programowania. W czasie wykonania dane są adresowane za pomocą łańcuchów znaków (ang. *string*) lub liczb całkowitych (ang. *integer*). Zapis danych dzielonych jest przeprowadzany podobnie jak modyfikacja krotek w języku Linda. Różnica polega na zastosowaniu mechanizmu odśmiecania (ang. *garbage collector*) do automatycznego usuwania struktur. Dane mogą być powielane. W przypadku modyfikacji danych powielonych żądanie zapisu jest wysyłane do oryginału. Wszystkie operacje sieciowe są przeprowadzane za pośrednictwem specjalnego procesu nazwanego AgoraServer. Agora dostarcza również mechanizmów automatycznej konwersji formatów danych. Zjawisko fałszywego współdzielenia jest zredukowane poprzez użycie typów danych definiowanych przez programistę. Replikacja danych może wpłynąć niekorzystnie na wydajność aplikacji używających tej implementacji DSM. Midway dostarcza wielu mechanizmów zapewniania spójności danych. W tym systemie DSM historycznie po raz pierwszy została wprowadzona spójność wejścia danych współdzielonych. Dane współdzielone kojarzone są ze zmiennymi synchronizacji na poziomie języka programowania. W Midway istnieje wiele typów takich zmiennych, łącznie z blokadami (ang. *locks*) i barierami (ang. *barriers*). Dzięki etykietowaniu zmiennych dzielonych, które również jest dostępne w tym systemie, możliwa jest realizacja pozostałych mechanizmów zachowywania spójności. Midway składa się z trzech komponentów: zbioru słów kluczowych i funkcji używanych do opisu programów współbieżnych, kompilatora generującego kod oznaczający dane po wystąpieniu zapisu jako zmodyfikowane (ang. *dirty*) oraz środowiska wykonawczego dostarczającego mechanizmów spójności. Do zarządzania zmiennymi synchronizującymi używane są znaczniki czasowe (ang. *timestamp*). Zaletą Midway jest różnorodność dostępnych modeli zapewniania spójności. Wadą jest trudny w użyciu mechanizm spójności wejścia. Projektanci systemu TreadMarks za cel postawili sobie stworzenie systemu DSM, który mógłby być zastosowany w multikomputerach zbudowanych ze stacji roboczych połączonych siecią lokalną typu Ethernet. Ponieważ takie sieci cechują się często dużymi opóźnieniami transmisji, to ważnym zadaniem projektowym była redukcja komunikacji koniecznej do zachowania spójności dzielonych danych. W TreadMarks użyto więc wariantu spójności zwalniania, nazywanego *leniwą spójnością zwalniania*, w którym zmiany propagowane są dopiero wtedy, gdy na jednym z węzłów zostanie wykonana operacja nabycia. Oprócz mechanizmu zachowania spójności innym źródłem nadmiarowej komunikacji jest zjawisko fałszywego współdzielenia. Aby zredukować częstość jego występowania TreadMarks używa protokołu wielu pisarzy (ang. *multiple-writer*) opartego na leniwym tworzeniu rekordów opisujących modyfikacje

dokonane na zawartości strony (ang. *diffs*).

Hybrydowe rozproszone pamięci dzielone (programowo-sprzętowe)

Mechanizm rozproszonej pamięci dzielonej można zaimplementować łącząc rozwiązania sprzętowe i programowe [13]. Elementy programowe wprowadzane są w sprzętowych DSM bazujących na katalogach z pełnym odwzorowaniem, aby zwiększyć ich skalowalność. Po przekroczeniu pojemności katalogu nadmiarowe pozycje katalogowe są przechowywane i zarządzane przez oprogramowanie. Głównym celem dodawania elementów sprzętowych do programowej rozproszonej pamięci dzielonej jest zwiększenie jej wydajności. Wspomaganie sprzętowe, w postaci specjalizowanych procesorów może przejąć od oprogramowania część operacji związanych np. z realizacją protokołów spójności.

Hybrydową DSM jest PLUS [9, 13, 63]. Odwzorowanie pamięci, zarządzanie spójnością oraz operacje niepodzielne w tej DSM realizowane są sprzętowo, natomiast system operacyjny odpowiedzialny jest za powielanie i przemieszczanie współdzielonych stron. Spójność jest zapewniana poprzez wykonywanie wszystkich modyfikacji w tej samej kolejności oraz przez użycie zapisu skrośnego (ang. *write-through*) do aktualizacji wszystkich kopii zmodyfikowanej strony. Zapisy są operacjami nieblokującymi. Operacja odczytu jest blokująca, jeśli dotyczy strony, której zawartość jest w danej chwili modyfikowana. Inna implementacja DSM o nazwie Lynx/Galactica Net wykorzystuje niezmodyfikowaną MMU jako wsparcie sprzętowe [13]. W tym rozwiązaniu RAM poszczególnych węzłów służy jako pamięć podręczna dla globalnej pamięci dzielonej. Wspomniana jednostka zarządzania pamięcią jest obsługiwana przez oprogramowanie pamięci wirtualnej zaimplementowane w systemie operacyjnym Mach. Strona pamięci jest jednostką współdzielenia i może znajdować się w jednym z trzech stanów: tylko do odczytu, prywatnym i aktualizacji. Rola systemu operacyjnego w zapewnieniu spójności polega na oznaczeniu stron jako aktualizowanych i zainicjowaniu tablic przechowujących informacje o węzłach posiadających kopie tych stron. Każdy zapis do strony aktualizowanej jest wykrywany przez sprzęt i propagowany do węzłów, które mają jej kopię. Jeśli zapisy nie są częste, to stosowane jest unieważnianie. W przeciwnym wypadku wykonywana jest aktualizacja. Alewife [13, 64] realizuje mechanizm DSM oparty na sprzętowym katalogu o ograniczonej pojemności, w którym dzięki oprogramowaniu sterowanemu przerwaniami emulowany jest katalog z pełnym odwzorowaniem. Kooperacja sprzętu i oprogramowania zapewnia również w Alewife ścisłą spójność. Ta implementacja DSM została użyta do realizacji systemu MGS, który integruje pamięci węzłów w DSM stosując jednostki współdzielenia o różnej ziarnistości [65]. FLASH stosuje oprogramowanie wspomagane przez sprzęt do realizacji DSM, w której RAM każdego węzła jest zarządzana jak pamięć podręczna [13, 66]. Protokół spójności tej pamięci jest podobny do protokołu zastosowanego w systemie DASH, ale posiada kilka dodatkowych elementów pozwalających uniknąć zakleszczeń (ang. *deadlock*). Jest on realizowany przez oprogramowanie korzystające ze specjalnego 64-bitowego procesora superskalarnego o nazwie MAGIC (ang. *Memory and General Interconnection Controller*), który integruje w sobie działanie kontrolera pamięci, sterownika wejścia-wyjścia oraz interfejsu sieciowego. Jego zadaniem jest zminimalizowanie konieczności angażowania procesora głównego w obsługę DSM. MAGIC zawiera i obsługuje sprzętowy katalog pamięci podręcznej oraz umożliwia realizację wielu protokołów spójności. Może również nadzorować wymianę komunikatów między węzłami multikomputera. PARADIGM jest architekturą łączącą w sobie elementy oprogramowania systemowego, sprzętu i komponentów zaliczanych do oprogramowania typu *firmware*, której jednym z zastosowań jest realizacja DSM [13, 67]. Podstawą działania rozproszonej pamięci dzielonej związanej z tą architekturą jest mechanizm rozproszonego pliku pamięci podręcznych, który pozwala na realizację efektywnych operacji wejścia-wyjścia i dostarcza mechanizmów tolerowania błędów. System operacyjny w architek-

turze ParaDiGM udostępnia każdemu procesowi użytkownika osobną przestrzeń adresową, do której odwzorowywane są pliki związane z tym procesem. To odwzorowanie jest implementowane za pomocą lokalnego zbioru pamięci podręcznych i modułów serwerowych, które nadzorują operacje zapisu i odczytu plików oraz zarządzają spójnością danych. Plik pamięci podręcznych jest odpowiednikiem zbioru stron pamięci wirtualnej w konwencjonalnych systemach operacyjnych. Operacje na tym pliku wykonywane są za pomocą mechanizmu RPC. Konstrukcja pamięci podręcznej w ParaDiGM pozwala użyć do zachowania jej spójności tych samych protokołów, które są używane w rozproszonych systemach plików. Jednostką spójności jest blok pamięci podręcznej, co w połączeniu ze sprzętem zapewniającym spójność pamięci podręcznej w obrębie węzła, pozwala osiągnąć większą wydajność, niż w systemach, w których jednostką spójności jest strona pamięci wirtualnej. Cashmere [68] jest rodziną implementacji DSM, których działanie uzależnione jest od funkcji dostarczanych przez sieci komputerowe dedykowane do budowania systemów wielokomputerowych, takie jak Memory Channel. Pierwsza wersja Cashmere oznaczona jako CSM-1L traktuje każdy procesor, nawet w wieloprocesorowych węzłach, jako oddzielny węzeł sieciowy. Programy, które korzystają z CSM-1L poddawane są dodatkowemu etapowi przetwarzania podczas kompilacji, którego celem jest zapewnienie wydajnej realizacji zapisu skrośnego. Taki zapis pozwala zwiększyć efektywną pojemność jaką oferuje ta pamięć. Druga wersja Cashmere, nazwana CSM-2L używa sprzętowego wsparcia spójności i stosuje dwukierunkowy mechanizm różnicujący (ang. *two-way diffing mechanism*), aby aktualizować kopie współdzielonych danych podczas zapisu. Cashmere-VLM [68, 69] pozwala dodatkowo zarządzać polityką stronicowania tak, aby uniknąć kosztownego wysyłania współdzielonych stron do dyskowych urządzeń wymiany. Zamiast tych urządzeń używana jest dostępna RAM wszystkich węzłów multikomputera. Aby algorytm wymiany uwzględniał zbiór stron współdzielonych, to wszystkie strony pamięci są dzielone na cztery kategorie: strony wyłączne (ang. *exclusive pages*), strony domowe (ang. *home pages*), strony modyfikowalne (ang. *read-write pages*) i strony tylko do odczytu oraz unieważnione (ang. *read-only and invalid pages*). Przynależność strony do jednej z wymienionych kategorii wpływa na prawdopodobieństwo jej wymiany. ViSMI to implementacja DSM [70], która przez jej twórców określana jest jako programowa, ale może być uruchomiona wyłącznie w systemie multikomputerowym opartym o sieć InfiniBand [10]. Można ją więc uznać za rozwiązanie hybrydowe. Jednostką współdzielenia w ViSMI jest strona. Stosowanym protokołem spójności jest leniwa spójność zwalniania. Komunikacja w tej implementacji DSM jest oparta na mechanizmie RDMA dostarczanym przez InfiniBand. Innym mechanizmem tej sieci stosowanym w ViSMI jest sprzętowa komunikacja rozgłoszeniowa, używana przy realizowaniu protokołu spójności do przesyłania zapisów różnic zawartości stron oryginalnych (ang. *clean page*) i zmodyfikowanych (ang. *dirty page*). ViSMI dostarcza również makr dla programistów, które ułatwiają zrównoleglenie programów sekwencyjnych.

Jako podsumowanie podrozdziału poświęconego DSM można przytoczyć opinie autorów Quarks [71]. Uważają oni, że większość projektantów DSM bierze pod uwagę w swoich pracach tylko wydajność, zaniedbując pozostałe kwestie, do których zaliczają:

- narzędzia do diagnostyki DSM,
- dostępność oznaczającą w tym przypadku możliwość pozyskania oprogramowania DSM,
- integrację z istniejącymi elementami środowiska (ang. *software environment*),
- stosowalność - większość implementacji DSM jest dedykowana dla określonego problemu lub grupy problemów.

2.2.3. Wsparcie dla usług zdalnych

Wirtualizacja pamięci rozproszonej, której celem jest wsparcie dla RAM jest szczególnie przydatna dla aplikacji pracujących z dużymi zbiorami roboczymi danych. Należy do nich oprogramowanie świadczące usługi wielu klientom, takie jak transakcyjne bazy danych, serwery treści multimedialnej, serwery aplikacji. Jeśli system operacyjny nie dostarcza aplikacjom takiego udogodnienia, to może ono zostać zaimplementowane w warstwie pośredniej oprogramowania (ang. *middleware*) lub wprost na poziomie aplikacji. Przykładami takich rozwiązań są Oracle Coherence Data Grid [1,72], produkty firmy RNA Networks (RNAcache i RNAmessenger) [22] oraz rozproszone struktury danych [14, 27, 73].

Oracle Coherence Data Grid jest warstwą pośrednią oprogramowania, której głównym zadaniem jest utworzenie z rozproszonej RAM pamięci podręcznej, która mogłaby pomieścić zbiór roboczy danych aplikacji, tym samym pozwalając na jego efektywne użytkowanie. Podstawowym elementem tego rozwiązania jest protokół sieci typu peer-to-peer (P2P). Zastosowanie takiego protokołu gwarantuje duży stopień odporności na błędy. Komputery mogą być dynamicznie dodawane lub usuwane ze środowiska tworzonego przez Oracle Coherence, ale te zmiany nie powodują utraty danych lub efektywności ich przetwarzania. Aby zwiększyć wykorzystanie dostępnej RAM ograniczono stosowanie replikacji danych jedynie do tworzenia kopii zapasowych, na wypadek awarii komputera, który przechowuje dane oryginalne. Zastosowano również mechanizm równomiernego rozmieszczania danych między poszczególnymi węzłami tworzącymi pamięć podręczną (ang. *load balancing*). W Oracle Coherence istnieje tylko jeden właściciel danych. Wszelkie zmiany informacji są wykonywane lokalnie, co jest osiągane przez zastosowanie migracji danych i przetwarzania (ang. *state and behaviour migration*). W tej kwestii zastosowano więc rozwiązania znane z niektórych implementacji DSM. Spójność pamięci podręcznej tworzonej przez Oracle Coherence i zawartości źródła danych (np. bazy danych) jest zapewniana przez strategie spójności pamięci podręcznej, których głównym założeniem jest idempotentność operacji związanych z aktualizacją źródła danych. Idempotentność w tym kontekście jest własnością operacji, która umożliwia jej wielokrotne powtarzanie bez skutków ubocznych, które byłyby niebezpieczne dla spójności danych.

Produkty firmy RNA Networks, podobnie jak Oracle Coherence, służą do wirtualizacji rozproszonej pamięci na poziomie warstwy pośredniej oprogramowania [15,22,23]. Niestety, szczegóły ich działania nie zostały publicznie ujawnione. RNAcache łączy pamięci operacyjne poszczególnych węzłów multikomputera w pojedynczy zasób, który nie jest związany (ang. *decoupled*) z żadnym konkretnym procesorem. Każdy z procesorów jest w równym stopniu uprawniony do korzystania z niego. Ta wspólna pamięci pozwala na wyeliminowanie operacji wymiany stron (ang. *swaping*), co skutkuje lepszym wykorzystaniem procesora, który nie musi czekać na ich zakończenie. To z kolei podnosi interaktywność i stopień wieloprogramowości systemu [4]. Dodatkową zaletą tego rozwiązania jest przezroczystość - żadne zmiany w aplikacji użytkowników nie są konieczne. RNAmessenger tworzy łącze komunikacyjne bazujące na pliku umieszczonym w puli pamięci stworzonej przez RNAcache. Wysłanie komunikatu jest równoważne zapisowi do pliku, a odebranie informacji odpowiada odczytowi z pliku. Eliminuje to konieczność używania przez aplikacje gniazd (ang. *socket*) do komunikacji w środowisku sieciowym.

Wirtualizację rozproszonej pamięci multikomputera na poziomie systemu operacyjnego wykorzystują rozwiązania typu SSI. Przykładowo, w skład oprogramowania Kerrighed wchodzi moduł KDDM (ang. *Kernel Distributed Data Management*) tworzący warstwę współdzielonej pamięci rozproszonej umożliwiającą migrację obiektów danych należących do jądra systemu. Dzięki niej usługi jądra zainicjowane w dowolnym węźle klastra mają dostęp do struktur danych jądra wszystkich pozostałych węzłów. Moduł ten może być również wykorzystany niezależnie od reszty oprogramowania Kerrighed w celu rozproszenia usług wykonywanych przez aplikacje

przestrzeni użytkownika.

Rozwiązaniem, które umożliwia wirtualizację rozproszonej pamięci operacyjnej na poziomie aplikacji użytkownika są opisane w Podrozdziale 1.3 Skalowalne, Rozproszone Struktury Danych (SDDS). W większości implementacji serwery SDDS są oprogramowaniem wykonywanym jako aplikacja użytkowa, natomiast klienci są realizowani na poziomie oprogramowania użytkowego lub hybrydowo - częściowo jako warstwa pośrednia (ang. *middleware*), a częściowo jako podsystem aplikacji [27, 74]. Pierwotnie SDDS zostały zaprojektowane na potrzeby systemów multikomputerowych, ale można je uogólnić na systemy rozproszone, czego przykładem jest SDDS LH*$_{\text{RS}}$ P2P [75, 76]. W tej implementacji SDDS każdy węzeł, nazywany partnerem (ang. *peer*) może pełnić równocześnie rolę klienta i serwera. Partnerzy tworzą sieć nakładkową (ang. *overlay network*) typu P2P (ang. *peer-to-peer*). Tą siecią zarządza nadzorca (ang. *super-peer*), który został nazwany koordynatorem partnerów (ang. *peer coordinator* - CP). Nadzoruje on podziały, a więc jest odpowiednikiem koordynatora podziałów znanego z klasycznej wersji SDDS LH*$_{\text{RS}}$. Do jego zadań należy również obsługa nowych węzłów, które chcą dołączyć się do sieci nakładkowej. Takie węzły początkowo pełnią wyłącznie rolę klientów SDDS i nazywane są węzłami kandydującymi lub uczniami (ang. *pupil*). Koordynator przypisuje im nauczycieli (ang. *tutor*). Nauczycielem może zostać węzeł, który posiada wiaderko danych, a więc pełni rolę nie tylko klienta, ale również serwera SDDS. Zadaniem nauczyciela jest przekazanie swoim uczniom posiadanego obrazu pliku SDDS i poinformowanie ich o położeniu poszczególnych partnerów w sieci nakładkowej. Jeśli nastąpi podział, to nauczyciel wysyła im komunikaty aktualizujące obraz pliku. Uczeń pozostaje pod nadzorem nauczyciela do momentu, aż otrzyma własne wiaderko danych. Jeśli zamiast wiaderka danych otrzyma wiaderko z informacjami nadmiarowymi, które mogą posłużyć do odtwarzania uszkodzonych wiaderek danych, to nadal pozostaje uczniem. Działanie mechanizmu tolerowania błędów w LH*$_{\text{RS}}$ P2P opiera się na parzystości wyliczanej za pomocą kodu detekcyjno-korekcyjnego Reeda Solomona [77]. Cześć partnerów przechowuje właściwe dane aplikacji, a pozostali magazynują informacje nadmiarowe, będące słowami kodowymi wyliczonymi na podstawie tych danych. Niektóre z węzłów mogą przechowywać wyłącznie te informacje, nie będąc ani klientami, ani serwerami SDDS. Mechanizm tolerowania błędów ma dodatkowe zastosowanie. W sieciach P2P może występować niekorzystne zjawisko nazywane „kotłowaniem" (ang. *chrun*). Polega ono na bardzo dynamicznej zmianie liczby partnerów uczestniczących w takiej sieci. Struktura SDDS LH*$_{\text{RS}}$ P2P minimalizuje skutki takiego zjawiska opóźniając przypisanie wiaderek z danymi do nowych partnerów poprzez odtwarzanie danych serwerów, które uległy awarii za pomocą informacji nadmiarowej. W tym działaniu uwzględniane są również przypadki opuszczenia i powrotu w relatywnie krótkim czasie partnerów, którzy pełnią rolę serwerów. Koncepcja SDDS stała się podstawą prac nad innymi strukturami rozproszonymi. Najbardziej zbliżonym do SDDS rozwiązaniem jest rozproszona struktura, do której zarządzania użyto algorytmu DDH (ang. *Distributed Dynamic Hashing*) [73]. W porównaniu do klasycznej SDDS LH* decyzja o podziale wiaderek jest podejmowana lokalnie przez serwer, a więc podziały nie muszą być uporządkowane. Eliminuje to „wąskie gardło" jakim jest w SDDS LH* koordynator podziałów. Schemat adresowania wiaderek przez klientów bazuje na drzewach pozycyjnych. Klucz rekordu jest wynikiem funkcji skrótu (ang. *hash function*) dla argumentu będącego zawartością rekordu. Znany klientowi poziom pliku wskazuje, ile bitów, począwszy od bitu najmłodszego, w kluczu będzie określało numer wiaderka, do którego ma być wysłane zapytanie. Pozostałe rozwiązania pochodne od SDDS nie są bezpośrednio związane z wirtualizacją rozproszonej pamięci operacyjnej. Rozproszone struktury danych DDS zaproponowane w [78] zaprojektowane zostały jako wsparcie dla usług internetowych (ang. *Internet services*), ale wirtualizują pamięć masową (dyskową) multikomputera. Protokoły P2P takie jak Chord, Symphony, CAN i Pastry, podobnie jak SDDS LH*, bazują na koncepcji rozproszonej tablicy haszującej (ang. *Distributed Hash Table* - DHT) i mogą stanowić infrastrukturę do budowy struktur danych

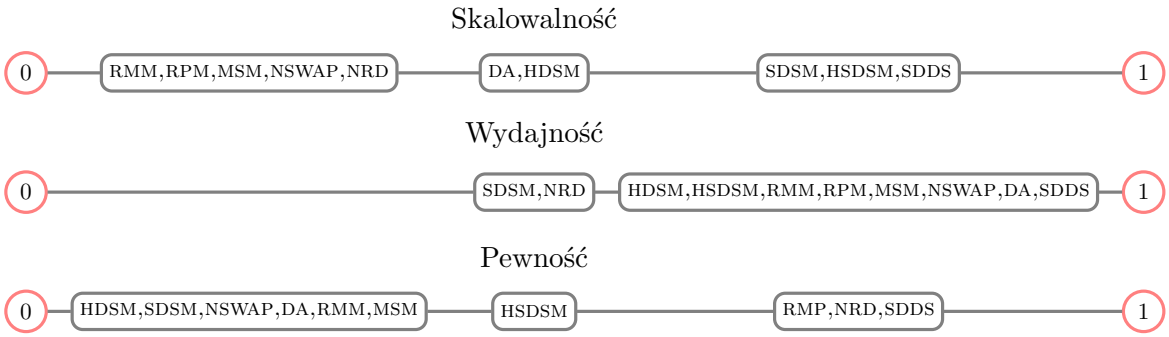
wirtualizujących RAM [79–82]. Z takiego rozwiązania skorzystali twórcy Oracle Coherence.

2.3. Podsumowanie

Problem wirtualizacji rozproszonej pamięci operacyjnej multikomputera jest przedmiotem wielu badań i znalazł liczne i różnorodne rozwiązania. Podstawowe idee zawarte w tych rozwiązaniach można podzielić na następujące kategorie:

- spójność,
- wydajność,
- odporność na błędy (pewność),
- skalowalność,
- wykorzystanie dostępnej pamięci operacyjnej.

Trzy z nich są uniwersalne dla wszystkich opisanych metod wirtualizacji rozproszonej pamięci operacyjnej (skalowalność, wydajność, odporność na błędy), a pozostałe mają znaczenie tylko dla niektórych z nich (np. spójność dla DSM, wykorzystanie nieużywanej, rozproszonej RAM dla rozproszonych RAM dysków i urządzeń wymiany). Tabela 2.1 jest zestawieniem ogólnych charakterystyk rozwiązań będących przedstawicielami każdej z opisanych klas zastosowań wirtualizacji pamięci RAM. Na podstawie jej zawartości można porównać wybrane rozwiązania dokonując szacunkowej oceny ich skalowalności, wydajności i pewności. Takie porównanie przedstawiono na Rysunku 2.2 (HDSM oznacza sprzętową implementację rozproszonej pamięci dzielonej, SDSM realizację programową DSM, a HSDSM hybrydową). Jedynym przedstawicielem metod wirtualizacji



Rysunek 2.2: Ocena rozwiązań realizujących wirtualizację rozproszonej RAM

pamięci operacyjnej multikomputera związanych ze wsparciem dla usług zdalnych są w tym porównaniu Skalowalne, Rozproszone Struktury Danych. Wybrano je, ponieważ na funkcjonalności dostarczanej przez SDDS i inne rozproszone struktury danych (np. DDH) bazuje działanie Oracle Coherence Data Grid i produktów firmy RNA Networks. Analizując porównanie warto zwrócić uwagę na wysokie oceny, jakie w każdej z branych pod uwagę kategorii uzyskały Skalowalne, Rozproszone Struktury Danych (SDDS). Wynikają one z efektywności używanych w tych strukturach algorytmów adresowania danych, braku elementów konstrukcji, które ograniczałyby ich skalowalność oraz z faktu, że w SDDS można łatwo zastosować mechanizmy tolerowania błędów. Należy zauważyć, że Skalowalne, Rozproszone Struktury Danych mogą być, tak jak rozproszone RAM dyski i urządzenia wymiany, postrzegane jako metoda wykorzystania dostępnej RAM węzłów multikomputera. W porównaniu z nimi SDDS odznaczają się lepszą skalowalnością, lecz

Wsparcie dla pamięci operacyjnej
Podstawowa jednostka danych - strona dla urządzeń wymiany (RMM, RMP, DA, MSM, Nswap), blok danych dla urządzeń blokowych. *Obsługa wielu klientów* - możliwa w większości implementacji urządzeń wymiany. Niektóre z nich obsługują nawet klientów heterogenicznych. Brak w przypadku urządzeń blokowych. *Możliwość współdzielenia danych* - brak w przypadku urządzeń blokowych, rozważana teoretycznie w przypadku urządzeń wymiany (RMM, MSM). *Skalowalność* - w większości rozwiązań dane są przenoszone na inny węzeł, jeśli zaczyna brakować miejsca bieżącemu serwerowi, co daje słabą skalowalność. Wyjątkiem jest DA, choć w tym przypadku górnym ograniczeniem jest rozmiar sieci, w której to rozwiązanie jest stosowane. *Odporność na błędy* - NRD i RMP oferują kilka mechanizmów tolerowania błędów.
Rozproszona pamięć dzielona
Podstawowa jednostka danych - dla sprzętowych DSM jest to najczęściej strona lub linia pamięci podręcznej procesora, dla programowych DSM jest to strona, segment lub zmienna (np. obiekt), w implementacjach hybrydowych może to być strona lub nawet plik. *Obsługa wielu klientów* - zazwyczaj węzły systemu stosującego DSM są równoprawne. Niektóre implementacje umożliwiają współpracę węzłów heterogenicznych. *Współdzielenie danych* - jest to podstawowe założenie koncepcji DSM. *Skalowalność* - wszystkie implementacje są w dużym stopniu skalowalne. W sprzętowych DSM skalowalność może być ograniczona ich konstrukcją oraz kosztami finansowymi, a w implementacjach programowych stosowanymi modelami spójności. Najlepsza skalowalność cechuje implementacje hybrydowe. *Odporność na błędy* - z opisanych rozwiązań jedynie ParaDiGM oferuje mechanizmy tolerowania błędów. W przypadku niektórych z pozostałych DSM tolerowanie błędów mogłoby być zrealizowane na bazie istniejącej replikacji danych. Odporność na błędy DSM oferowanych przez języki programowania może być zrealizowana na poziomie aplikacji, która są tworzone z użyciem takich języków.
Wsparcie dla usług zdalnych
Podstawowa jednostka danych - najczęściej obiekt, rekord lub strona. *Obsługa wielu klientów* - w niektórych rozwiązaniach węzły multikomputera lub systemu rozproszonego są równoprawne. Inne rozwiązania zaprojektowane są z uwzględnieniem możliwości obsługi dużej liczby klientów. *Współdzielenie danych* - możliwe w większości przypadków. *Skalowalność* - podstawowy cel projektowy dla tego rodzaju rozwiązań. *Odporność na błędy* - jest jedną z podstawowych własności, niektóre ze stosowanych mechanizmów tolerowania błędów bazują na protokołach sieci peer-to-peer.

Tabela 2.1: Podsumowanie informacji o zastosowaniach wirtualizacji rozproszonej pamięci

ich zastosowanie w aplikacjach użytkowych wymaga dodatkowych zabiegów. Pożądanym byłoby zatem opracowanie rozwiązania, które łączyłoby skalowalność i wydajność SDDS z dostępnością dla aplikacji, jaką cechują się rozproszone RAM dyski i urządzenia wymiany. To rozwiązanie zaliczałoby się równocześnie do klas zastosowań wirtualizacji rozproszonej RAM związanych ze wsparciem dla pamięci operacyjnej i ze wsparciem dla usług zdalnych. W następnym rozdziale rozprawy rozważana jest możliwość realizacji takiego rozwiązania.

3. Motywacja

Ten rozdział opisuje problem, który jest przedmiotem rozprawy oraz zawiera argumentację uzasadniającą podjęcie badań z nim związanych. W Podrozdziale 3.1 zawarto krótkie, analityczne podsumowanie informacji umieszczonych w Rozdziale 2 oraz wprowadzenie do rozpatrywanego problemu. W części 3.2 rozdziału przeanalizowano charakterystykę w kontekście SDDS problemu badawczego. Ostatnia część rozdziału zawiera uszczegółowienie tezy i celu rozprawy.

3.1. Analiza dziedziny

W Rozdziale 2 zostały przedstawione główne kierunki badań nad wirtualizacją pamięci rozproszonej multikomputerów: wsparcie dla pamięci operacyjnej, rozproszona pamięć dzielona, wsparcie dla usług zdalnych. DSM jest prawdopodobnie najlepiej zbadaną metodą wirtualizacji RAM systemów wielokomputerowych. Zaproponowano dla niej wiele sposobów realizacji na różnych poziomach systemu komputerowego. Wsparcie dla pamięci operacyjnej dotyczy wykorzystania nieużywanej, rozproszonej RAM multikomputera jako urządzenia blokowego (ang. *block device*) [30,39], które najczęściej pełni rolę urządzenia wymiany. Wnioskując po liczbie publikacji, mniej zbadanym zastosowaniem tej pamięci jest jej użycie w charakterze RAM dysku.

Wsparcie dla usług zdalnych jest stosunkowo nowym i dosyć ogólnym zagadnieniem, które obejmuje dwie wyżej przytoczone kategorie, ale również dotyczy innych rozwiązań w dziedzinie wirtualizacji pamięci. Oracle Coherence Data Grid i produkty firmy RNA Networks są przykładem połączenia takich koncepcji jak DSM, rozproszone urządzenia wymiany, rozproszone RAM dyski i sieci P2P w jedno środowisko, które wspomaga pracę oprogramowania oferującego usługi zdalne. Wydajność działania takiego środowiska jest więc zależna od składowych realizujących te podstawowe koncepcje. Wynika stąd konieczność tworzenia nowych lub udoskonalania istniejących fundamentalnych metod wirtualizacji rozproszonej RAM dla celów wspomagania usług zdalnych. Do takich metod należą Skalowalne, Rozproszone Struktury Danych.

Istnieje pewne podobieństwo między SDDS, a rozwiązaniami wspomagającymi pamięć operacyjną. W obu przypadkach jednym z głównych założeń jest wykorzystanie dostępnej RAM węzłów multikomputera. Jednakże, o ile większość rozwiązań z zakresu wspomagania pamięci operacyjnej wykorzystuje tylko RAM pojedynczego węzła, o tyle SDDS tworzą spójny obraz zasobu pamięci, którego rozmiar dostosowuje się do potrzeb aplikacji, czyli jest skalowalny. Dodatkowo SDDS nie posiadają elementów takich jak centralny katalog, które ograniczają skalowalność już na poziomie architektury rozwiązania. Czas działania algorytmów adresowania, które są używane w Skalowalnych Rozproszonych Strukturach Danych, nie jest zależny od wielkości pliku SDDS, dlatego lokalizacja danych w takich strukturach jest bardzo szybka. Kolejnym atutem SDDS jest łatwa do uzyskania odporność na błędy.

Zasadniczą wadą, która powoduje, że SDDS nie są szeroko stosowanym rozwiązaniem [76] są wymagania jakie trzeba spełnić, aby udostępnić funkcjonalność Skalowalnych, Rozproszonych Struktur Danych aplikacjom użytkowym. Oryginalna koncepcja SDDS zakłada całkowitą ich implementację w przestrzeni użytkownika. Aplikacja może więc korzystać z takich struktur

danych, tylko jeśli była tworzona z myślą o ich zastosowaniu lub jeśli została przeprowadzona odpowiednia modyfikacja jej kodu źródłowego. Inne możliwości zakładają stworzenie warstwy pośredniej oprogramowania lub udostępnienie aplikacjom SDDS jako usługi systemu operacyjnego. Pierwsze rozwiązanie tylko częściowo eliminuje konieczność ingerencji w kod źródłowy aplikacji [74].

Tę niedogodność można próbować usunąć udostępniając SDDS przez interfejs, który jest „znany" aplikacjom użytkowym, taki jakim są wywołania systemowe (ang. *system calls*) [2–4]. Dodanie nowych wywołań do zbioru istniejących w systemie operacyjnym nie stanowi jednakże rozwiązania problemu, gdyż ich użycie również wymagałoby dokonania zmian w kodzie źródłowym aplikacji [30]. Należy posłużyć się takimi, które już istnieją w systemie i są często używane przez programy użytkowe. Większość współczesnych systemów operacyjnych zbudowana jest w oparciu o paradygmat „wszystko jest plikiem", więc do tej grupy wywołań systemowych zaliczają się te, które związane są z obsługą plików. Aby za ich pomocą umożliwić aplikacjom korzystanie z SDDS należy stworzyć klienta SDDS, który byłby elementem systemu operacyjnego aktywowanym przez te wywołania. Istnieją dwa rodzaje takich elementów, które w tym przypadku mogą być wzięte pod uwagę: systemy plików i sterowniki urządzeń blokowych. Klient SDDS w postaci sterownika urządzenia blokowego jest rozwiązaniem niskopoziomowym, za to bardziej uniwersalnym od systemu plików. Tworzy on wirtualne urządzenie, na którym można osadzić dowolny lokalny system plików, zależnie od potrzeb wynikających z przeznaczenia tego urządzenia. W szczególności może ono zostać użyte jako przestrzeń wymiany w klasycznej implementacji pamięci wirtualnej. Za wyborem takiego rozwiązania przemawia również wykorzystanie sterowników urządzeń blokowych w innych implementacjach wirtualizacji pamięci operacyjnej multikomputera [26, 32, 38, 42].

3.2. Charakterystyka SDDS

Koncepcja którą przedstawiono w poprzednim podrozdziale zakłada przedstawienie aplikacjom użytkowym Skalowalnych, Rozproszonych Struktur Danych (SDDS) jako urządzenia blokowego. Oprócz przytoczonych argumentów taki wybór uzasadniają również analogie w działaniu i budowie SDDS oraz urządzeń blokowych. Do takich urządzeń zalicza się urządzenia o dostępie swobodnym (ang. *random access*), których wielkość podstawowej jednostki informacji jest rzędu co najmniej kilkuset bajtów. W tej kategorii znajdują się urządzenia pamięci masowej. Mimo wspólnych cech różnią się one budowa i parametrami fizycznymi [2–4, 30]. Urządzenie blokowe jest więc abstrakcją, która stanowi wspólny interfejs dla takiej klasy urządzeń. Pozwala ona przedstawić urządzenie pamięci masowej jako tablicę, której każdy element, nazywany sektorem, ma unikalny indeks (adres) będący liczbą naturalną oraz wielkość wynoszącą co najmniej 512 bajtów. Urządzenie blokowe pozwala przeprowadzić pojedynczą operację zapisu lub odczytu na więcej niż jednym sektorze. Taką grupę przyległych sektorów, na których wykonywana jest operacja, określa się mianem bloku, a operację nazywa się blokową. SDDS odpowiadają tej charakterystyce. Pojedynczą jednostką danych w takich strukturach jest rekord identyfikowany za pomocą unikatowego klucza - odpowiednik bloku lub sektora. Zbiór wszystkich wiaderek wchodzących w skład SDDS postrzegany jest jako plik o swobodnym dostępie lub rozproszona tablica haszująca DHT. Te podobieństwa umożliwiają i ułatwiają implementację klienta SDDS w postaci sterownika urządzenia blokowego.

Między SDDS i klasycznymi implementacjami pamięci wirtualnej, w szczególności stronicowaniem na żądanie, też występują pewne analogie. Zostały one zestawione w Tabeli 3.1.

Istnieją trzy architektury SDDS, które zostały opisywane w Podrozdziale 1.3. Każda z nich posiada cechy, które określają stopień jej zgodności z modelem urządzenia blokowego i zasadność oraz możliwość udostępnienia jej implementacji aplikacjom użytkownika za pomocą ta-

Stronicowanie na żądanie	**SDDS**
Rozmiar strony jest stały.	Rozmiar rekordu może być stały.
W przypadku braku miejsca w pamięci operacyjnej część stron jest wycofywana do pamięci pomocniczej (urządzenia wymiany). Operacją tą steruje algorytm wymiany.	W przypadku braku miejsca w wiaderku część rekordów przesyłana jest do innego wiaderka. Operacją tą steruje algorytm podziału wiaderka.
Adresy wirtualne stron przekształcane są na adresy fizyczne. Translacji dokonuje jednostka MMU.	Klucz rekordu zamieniany jest na adres logiczny wiaderka. Translacji dokonuje klient SDDS.

Tabela 3.1: Analogie między stronicowaniem na żądanie, a SDDS

kiego interfejsu. SDDS SD-Rtree służy do zarządzania danymi adresowanymi za pomocą wielu indeksów. Urządzenie blokowe nie zapewnia wykorzystania wszystkich możliwości tej struktury. Dwie pozostałe architektury lepiej pasują do modelu urządzenia blokowego. SDDS RP* grupuje rekordy o kolejnych kluczach w obrębie tego samego wiaderka. Ta cecha ma zarówno pozytywne, jak i negatywne następstwa. Korzyścią z niej płynącą jest możliwość łatwej implementacji pojedynczych operacji, które obejmują duże grupy przyległych rekordów. Z drugiej strony ta cecha powoduje w przypadku takich operacji lokalność odwołań do wiaderek, co może skutkować obciążeniem serwerów SDDS, które nimi zarządzają i tym samym niekorzystnie wpływać na wydajność całej struktury. Odwrotnie przedstawia się sytuacja w przypadku SDDS LH*. Kiedy wykonywane są operacje dotyczące przyległych rekordów, to dzięki zastosowaniu funkcji haszujących obciążenie komunikacją klient-serwer jest równomiernie rozłożone na wszystkie serwery SDDS, co jest obserwacją wynikającą z Chińskiego Twierdzenia o Resztach [83]. Ta sama cecha jednocześnie utrudnia prostą implementację tych operacji - wymaga odpytania większej liczby serwerów zamiast jednego.

3.3. Podsumowanie

Skalowalne, Rozproszone Struktury Danych są metodą wirtualizacji rozproszonej pamięci systemu wielokomputerowego, która jest typowym rozwiązaniem stosowanym na poziomie aplikacji. Ta cecha ogranicza zastosowanie tej metody do programów użytkowych, które podczas tworzenia zostały wyposażone w część kliencką SDDS lub które zostały przystosowane do używania tych struktur poprzez odpowiednie modyfikacje. Klasyczne techniki wirtualizacji pamięci, takie jak stronicowanie na żądanie, realizowane są na poziomie systemu operacyjnego i sprzętu co zapewnia ich przezroczystość dla aplikacji użytkownika. *Powstaje zatem pytanie jak przenieść zarządzanie wiaderkami SDDS na poziom systemu operacyjnego (analogicznie do zarządzania stronami w stronicowaniu na żądanie) i jaka będzie efektywność takiego rozwiązania.* Rozprawa daje odpowiedź na to pytanie. Rozwiązanie oparto na spostrzeżeniu, że oprogramowanie klienckie SDDS może być wykonane w postaci sterownika wirtualnego urządzenia blokowego. Aby osiągnąć zamierzony cel opracowano architekturę rozwiązania oraz przykładową implementację. Prototyp poddano testom, które dobrano mając na uwadze tezę rozprawy.

Przedstawiona w następnym rozdziale architektura może posłużyć do realizacji proponowanego wariantu SDDS dla dowolnego systemu operacyjnego wspierającego paradygmat urządzeń blokowych (np. wszystkie odmiany Uniksa, systemy rodziny MS Windows). Prototyp rozwiązania został opracowany dla systemu Linux. Wybór systemu operacyjnego został podyktowany następującymi względami:

- duża zgodność z systemem Unix, z którego wywodzi się koncepcja urządzeń blokowych,

- szerokie zastosowanie w systemach multikomputerowych [84],
- dostępność kodu źródłowego jądra systemu, co ułatwia jego analizę [85],
- możliwość tworzenia sterowników urządzeń w postaci modułów, które mogą być dynamicznie ładowane do jądra, co eliminuje konieczność restartu systemu.

Rozwiązanie o nazwie SDDSfL zostało oparte na architekturze SDDS LH*. Cześć kliencka implementacji SDDSfL jest sterownikiem urządzenia blokowego, którego działanie jest podobne do działania sterowników innych urządzeń tego typu, również w zakresie reagowania na awarie. Pozostałe elementy SDDSfL, czyli koordynator podziałów (SC) i serwery zostały opracowane jako oprogramowanie wykonywane na poziomie aplikacji.

Testy jakim poddano prototyp SDDSfL miały na celu zbadanie wydajności tego rozwiązania, porównanie jej z efektywnością zarówno typowych urządzeń blokowych, jaki i rozproszonych systemów plików. Pomiar efektywności dokonany został dla aplikacji zorientowanych na wejście-wyjście i wymagających swobodnego dostępu do danych oraz dla aplikacji zorientowanych na obliczenia [2] i wymagających swobodnego dostępu do danych. Do testów zastosowano także różne konfiguracje systemu wielokomputerowego.

4. Architektura SDDSfL

W tym rozdziale zawarto opis architektury SDDSfL. Pierwszy podrozdział jest wprowadzeniem do SDDS LH*, na której bazuje SDDSfL. W drugim dokonano analizy możliwości przeniesienia idei SDDS w przestrzeń systemu operacyjnego. W podrozdziale trzecim opisano wysokopoziomową strukturę SDDSfL. Trzy kolejne podrozdziały zawierają opisy odpowiednio klienta, serwera i koordynatora podziałów SDDSfL. Ostatni podrozdział zawiera opis protokołów warstwy aplikacji stosowanych w Skalowalnej, Rozproszonej Strukturze Danych dla Linuksa.

4.1. SDDS LH*

Skalowalne, Rozproszone Struktury Danych LH* oparte są na algorytmie haszowania liniowego (ang. *linear hashing* - LH) uogólnionym na systemy rozproszone i wielokomputerowe [27, 28, 86]. Podstawowy algorytm LH zakłada, że dane są zgromadzone w pliku umieszczonym w pamięci operacyjnej lub masowej. Plik ten podzielony jest na wiaderka (ang. *buckets*), które zawierają rekordy. Haszowanie liniowe pozwala zlokalizować dane, w postaci rekordu za pomocą par funkcji haszujących oznaczonych symbolami h_i i h_{i+1}, gdzie $i = 0, 1, 2 \ldots$. Ogólna postać takich funkcji najczęściej jest dana Wzorem 4.1, gdzie C jest unikatowym kluczem rekordu, a N początkową liczbą wiaderek ($N \geqslant 1$).

$$h_i(C) \leftarrow C \bmod N \cdot 2^i \tag{4.1}$$

Tak zbudowany i adresowany plik może zmieniać swój rozmiar bez negatywnego wpływu na czas dostępu do danych w nim zgromadzonych oraz bez dużego zapotrzebowania na pamięć. Zwiększenie rozmiaru pliku odbywa się za pomocą operacji nazwanej podziałem wiaderka (ang. *bucket split*). Do podziału dochodzi, jeśli pojemność jednego z wiaderek zostanie przekroczona w wyniku wstawiania do niego nowych rekordów. Takie zdarzenie nazwane jest kolizją (ang. *collision*). Podziałowi ulega wiaderko, którego numer jest wyznaczony parametrem n nazwanym wskaźnikiem podziału (ang. *split pointer*). Wartość tego wskaźnika jest wyliczana według Wzoru 4.2, przy czym początkowo wynosi ona zero.

$$n \leftarrow (n + 1) \bmod N \cdot 2^i \tag{4.2}$$

Przebieg czynności podziału jest następujący:

1. tworzone jest nowe wiaderko,
2. rekordy w wiaderku wskazywanym przez n są dzielone na dwie grupy z użyciem funkcji h_{i+1} i jedna z tych grup jest przesyłana do nowego wiaderka,
3. zwiększana jest wartość n według Wzoru 4.2.

Wskaźnik podziału wskazuje kolejne wiaderka, do których adresowania jest używana funkcja h_i, aż do momentu, kiedy wszystkie ulegną podziałowi. Wówczas funkcja h_i zastępowana

jest funkcją h_{i+1}, a funkcja h_{i+1} funkcją h_{i+2} i cały cykl podziałów zaczyna się od początku. Adres (numer) wiaderka w LH zawierającego rekord o kluczu C jest wyznaczany według Algorytmu 1. Indeksy i i $i+1$ nazywane są poziomami wiaderek. Wartość poziomu wiaderka wzrasta

Algorytm 1 Adresowanie wiaderek w LH

$a \leftarrow h_i(\mathrm{C})$
if $a < n$ **then**
  $a \leftarrow h_{i+1}(\mathrm{C})$
end if

o jeden po jego podziale. Aby przedstawiony algorytm adresowania był poprawny, to poziomy wszystkich wiaderek pliku muszą się różnić co najwyżej o jeden. Gwarantuje to wskaźnik podziału, który nie dopuszcza do podziału wiaderek o poziomie $i+1$, zanim nie zostaną podzielone wszystkie wiaderka o poziomie i. Wartość tego wskaźnika jest używana w algorytmie adresowania do określenia, która z funkcji haszujących ma zostać użyta do wyznaczenia adresu wiaderka. Jeśli podziały wiaderek następują tylko w wyniku kolizji, to taki rodzaj podziałów określany jest jako podziały niekontrolowane. Podziały kontrolowane uwzględniają dodatkowo współczynnik wypełnienia pliku (ang. *load factor*), czyli stosunek wielkości pliku do jego zawartości. Jeśli jego wartość przekroczy pewien przyjęty próg i wystąpi kolizja, to następuje podział. Rozmiar pliku może również maleć, na skutek łączenia wiaderek. Jest to operacja odwrotna do podziału i polega na połączeniu zawartości wiaderka o numerze n z ostatnim wiaderkiem w pliku. Wartość wskaźnika podziału jest zmniejszana zgodnie z Algorytmem 2. Operacja kurczenia pliku,

Algorytm 2 Zmniejszanie pliku w LH

$n \leftarrow n - 1$
if $n < 0$ **then**
  $i \leftarrow i - 1$
  $n \leftarrow 2^i - 1$
end if

podobnie jak podziały kontrolowane wymaga monitorowania poziomu wypełnienia pliku. Jeśli jego wartość spadnie poniżej przyjętego progu wówczas przeprowadzane jest łączenie wiaderek.

W SDDS LH* wiaderka rozlokowane są na osobnych węzłach multikomputera nazywanych serwerami SDDS. Przyjętym jest, że każde wiaderko rezyduje w RAM takiego węzła. Architektura SDDS LH* nie określa w jaki sposób rekordy wewnątrz wiaderka są zarządzane, ale wymaga, aby każde wiaderko posiadało swój poziom, który jest liczbą naturalną oznaczoną przez j. Każdy serwer SDDS posiada swój unikalny adres logiczny, który jest oznaczony przez a i również jest liczbą naturalną. Adresy logiczne odwzorowywane są na adresy fizyczne (najczęściej adresy IP) za pomocą tablicy przydziałów fizycznych (ang. *physical allocation table*), oznaczonej jako T, która może być tworzona dynamicznie lub statycznie [87]. Zawartość tej tablicy jest znana wszystkim elementom SDDS LH*. Zbiór wszystkich wiaderek tworzy plik SDDS, który może zmieniać rozmiar, tak jak w oryginalnym algorytmie LH. Z tego pliku mogą korzystać klienci SDDS, którzy umieszczani są, podobnie jak serwery, na osobnych węzłach wielokomputera. Do adresowania wiaderek mógłby zostać użyty Algorytm 1, ale wtedy każdy z klientów musiałby znać bieżące wartości parametrów i i n pliku SDDS. Wymagałoby to stworzenia centralnego katalogu, w którym byłyby one przechowywane, co jednak jest sprzeczne z pierwszym założeniem SDDS opisanym w Podrozdziale 1.3. Katalog centralny stanowiłby „wąskie gardło” (ang. *hot-spot*) w strukturze. Natychmiastowe powiadamianie wszystkich klientów o zmianie wartości i i n również powodowałoby obniżenie wydajności SDDS. Rozwiązaniem pozbawionym takiej wady jest rozszerzenie haszowania liniowego o nazwie LH*. W LH* parametry i i n występujące w Algo-

rytmie 1 zastąpiono parametrami i' i n', stanowiącymi lokalny obraz pliku. Różnica polega na tym, że wartości tych parametrów nie muszą opisywać bieżącego stanu pliku. Są one uaktualniane dopiero wówczas, kiedy klient używający Algorytmu 1 z parametrami i' i n' popełni błąd adresowania (zasada „leniwej aktualizacji"). W takiej sytuacji serwer SDDS, który otrzymał nieprawidłowo zaadresowane żądanie wysyła klientowi komunikat korygujący obraz (ang. *Image Adjustment Message*), a żądanie przekazuje do właściwego adresata. Komunikat IAM zawiera poziom wiaderka (j) serwera, który otrzymał niewłaściwe żądanie. Klient wykonuje Algorytm 3 celem skorygowania swojego obrazu (a jest adresem logicznym serwera, który otrzymał źle zaadresowane żądanie). Po wykonaniu opisanego algorytmu klient nie popełni ponownie tego

Algorytm 3 Korekcja obrazu klienta SDDS LH*

```
i' ← j − 1
n' ← a + 1
if n' ⩾ 2^i' then
  n' ← 0
  i' ← i' + 1
end if
```

samego błędu adresowania, ale może popełnić inne. Będą one korygowane w ten sam sposób. Możliwość popełnienia błędu adresowania przez klienta wymusza na serwerach SDDS weryfikację poprawności odbieranych żądań. Dokonywana jest ona za pomocą Algorytmu 4. W jego zapi-

Algorytm 4 Weryfikacja poprawności adresu żądania przez serwer SDDS LH*

```
a' ← h_j(C)
if a' ≠ a then
  a'' ← h_{j−1}(C)
  if a'' > a and a'' < a' then
    a' ← a''
  end if
end if
```

sie a oznacza adres logiczny serwera dokonującego weryfikacji, a j poziom jego wiaderka. Jeśli $a' = a$ to żądanie jest prawidłowo zaadresowane, w przeciwnym przypadku serwer a wysyła klientowi IAM oraz przesyła jego żądanie do serwera o adresie a', który zgodnie z jego obrazem pliku, określonym poziomem jego wiaderka, powinien być właściwym adresatem żądania. To założenie niekoniecznie musi być prawdziwe, więc serwer otrzymujący przekazane żądanie również wykonuje Algorytm 4. Jeśli i on okaże się niewłaściwym adresatem, to przesyła żądanie klienta do kolejnego serwera. Nieprawidłowo zaadresowany komunikat jest przesyłany między serwerami SDDS co najwyżej dwukrotnie.

Podobnie jak w przypadku haszowania liniowego, plik SDDS LH* ulega powiększeniu na skutek podziałów wiaderek. Możliwe jest również ich łączenie i tym samym zmniejszanie wielości pliku SDDS, ale to rozwiązanie rzadko jest stosowane w praktyce i nie będzie tu opisywane. Wymagania odnośnie kolejności podziału wiaderek są podobne jak w LH - poziomy wszystkich wiaderek tworzących plik mogą się różnić co najwyżej o jeden, a więc wszystkie wiaderka o poziomie j muszą ulec podziałowi, zanim zostanie podzielone którekolwiek z wiaderek o poziomie $j + 1$. Nadzorem nad kolejnością podziałów zajmuje się element SDDS LH* nazwany koordynatorem podziałów (ang. *split coordinator* - SC). Jest on jedyną składową SDDS LH*, która zna dokładny obraz pliku SDDS, czyli aktualne wartości parametrów n i i. Jeśli wiaderko serwera ulega przepełnieniu, to ten wysyła do SC komunikat o kolizji. Koordynator wykonuje

Algorytm 5 celem wyznaczenia wiaderka, które ma się podzielić i wysyła serwerowi, który jest właścicielem wiaderka o numerze n komunikat „you split" (pol. *ty się podziel*).

Algorytm 5 Wyznaczanie przez SC wiaderka do podziału

```
n ← n + 1
if n ⩾ 2^i then
    n ← 0
    i ← i + 1
end if
```

Po otrzymaniu przez serwer takiego komunikatu wykonywana jest operacja podziału, która przebiega następująco:

1. tworzone jest nowe wiaderko o poziomie $j + 1$,
2. rekordy w wiaderku n są dzielone na dwie grupy przy użyciu funkcji h_{j+1} i jedna z tych grup, zawierająca około połowy rekordów jest wysyłana do nowego wiaderka,
3. zwiększany jest o jeden poziom wiaderka n,
4. serwer będący właścicielem wiaderka n wysyła do SC komunikat potwierdzający wykonanie operacji podziału.

4.2. Analiza możliwości przeniesienia SDDS na poziom systemu operacyjnego

W oryginalnej architekturze SDDS LH* występują trzy zasadnicze elementy: klienci, serwery i koordynator podziałów [27]. Podczas projektowania architektury SDDSfL konieczne było zadecydowanie o tym, które z nich powinny być przeniesione na poziom systemu operacyjnego. Ten podrozdział przedstawia podjęte decyzje wraz z argumentami, które zostały rozważone podczas ich podejmowania.

4.2.1. Klient SDDSfL

Jednym z głównych założeń SDDSfL jest udostępnienie aplikacjom użytkowym funkcjonalności SDDS za pomocą interfejsu, który jest już im znany. Dzięki temu usunięta zostanie konieczność dokonywania modyfikacji kodu tych aplikacji. To wymaganie sugeruje realizację klienta SDDSfL na poziomie systemu operacyjnego, który takiego interfejsu dostarcza. Pozostaje jednak kwestia tego w jakiej postaci i w której warstwie oprogramowania systemowego taki klient powinien być zrealizowany. Istnieje kilka możliwości, które należy rozważyć. Pierwszą jest realizacja poza jądrem systemu operacyjnego.

Zalety takiego rozwiązania są następujące:

- Niezawodność - oprogramowanie systemowe zrealizowane w ten sposób podlega takiej samej ochronie jak zwykłe aplikacje użytkowe, dlatego błąd w jego działaniu nie jest w stanie zakłócić pracy procesów, z którymi nie współpracuje.
- Łatwość implementacji - istnieje cały zestaw bibliotek podprogramów oraz narzędzi, które można wykorzystać do tworzenia klienta SDDSfL, usuwania błędów (ang. *debugging*) w jego kodzie źródłowym i weryfikacji poprawności działania.

Wady natomiast obejmują:

- Przełączanie kontekstu - aby żądanie aplikacji zostało zrealizowane przez klienta sterowanie musi zostać przekazane do jądra systemu operacyjnego, które przekaże je z kolei do klienta SDDSfL. Wiąże się to ze zmianą trybu procesora i przełączeniami kontekstu, co wydłuża czas obsługi takiego żądania. Dodatkowo polityka szeregowania procesów i wątków w systemie nie gwarantuje, że klient SDDSfL zostanie uaktywniony zaraz po tym jak aplikacja zgłosi zapotrzebowanie na wykonanie przez niego żądanej usługi.

- Interfejs - aby aplikacje mogły skorzystać z usług klienta SDDSfL muszą mieć możliwość komunikowania się z nim, a to może być zapewnione jedynie przez ingerencję w ich kod źródłowy.

Ostatnia z wymienionych wad jednoznacznie sugeruje, że klient SDDSfL nie może zostać zrealizowany jako oprogramowanie działające w przestrzeni użytkownika i dostarczające usług systemowych.

Druga możliwość, to realizacja klienta bezpośrednio w jądrze systemu operacyjnego, za pomocą zestawu wywołań systemowych.

Jako zalety takiego rozwiązania należy uznać:

- Interfejs - wywołania systemowe stanowią podstawowy środek komunikacji z systemem operacyjnym, więc są znanym aplikacjom użytkowym interfejsem.

- Wydajność - konieczne jest tylko jedno przełączenie kontekstu, aby rozpocząć obsługę żądania aplikacji.

Wadami są natomiast:

- Niezawodność - błąd w funkcjach implementujących wywołania systemowe może doprowadzić do awarii wszystkich procesów uruchomionych w systemie.

- Trudność realizacji - jądro systemu nie dostarcza zazwyczaj tylu bibliotek podprogramów, ile ma do dyspozycji programista tworzący aplikacje użytkowe. Część systemów operacyjnych nie pozwala na modyfikację lub dodanie nowych wywołań systemowych. Aby zrealizować usługi dostarczane przez klienta SDDSfL należy przejąć obsługę istniejących wywołań systemowych lub dodać nowe. W pierwszym przypadku konieczne jest zachowanie semantyki funkcji, które poprzednio implementowały te wywołania. Nie zawsze można to zrealizować, w szczególności, jeśli system jest przeznaczony do pracy na wielu platformach sprzętowych [30].

- Trudność użytkowania - wdrożenie nowego oprogramowania działającego w trybie jądra wiąże się w przypadku większości stosowanych systemów operacyjnych z restartem całego systemu komputerowego. Jeśli klient SDDSfL został zrealizowany poprzez przejęcie obsługi istniejących wywołań, to działanie niektórych aplikacji może być zakłócone, o ile nie zostało uwzględnione poprzednie zachowanie funkcji obsługujących te wywołania. W przypadku realizacji klienta poprzez dodanie nowych wywołań konieczna jest modyfikacja aplikacji użytkowych, aby mogły z nich skorzystać.

Ostatnie dwie z rozważonych możliwości zakładają realizację klienta SDDSfL w postaci sterownika urządzenia blokowego lub systemu plików.

Za implementacją klienta w postaci systemu plików przemawia:

- Możliwość implementacji w postaci modułu - takie oprogramowanie może być ładowane i usuwane z jądra systemu bez konieczności restartu komputera.

- Możliwość implementacji w przestrzeni użytkownika - Linux, który jest docelowym systemem dla SDDSfL pozwala na zaimplementowanie sterownika systemu plików w przestrzeni użytkownika za pomocą mechanizmu o nazwie FUSE [88]. Wprowadza to opóźnienia w działaniu systemu plików, ale pozwala uniknąć destabilizacji pracy systemu w przypadku ujawnienia się błędów w oprogramowaniu sterownika. Ponadto, po przetestowaniu w przestrzeni użytkownika takie oprogramowanie może zostać przeniesione w przestrzeń jądra.
- Interfejs - każda aplikacja pracująca z plikami może bez modyfikacji współpracować z klientem SDDSfL zrealizowanym jako system plików.

Kontrargumenty są następujące:

- Niezawodność - jeśli sterownik systemu plików jest zrealizowany w przestrzeni jądra to błędy w nim zawarte mogą doprowadzić do awarii wszystkich procesów w systemie, łącznie z jądrem systemu operacyjnego.
- Wydajność - jeśli klient SDDSfL jest zrealizowany w postaci oprogramowania działającego w przestrzeni użytkownika, to może nie być wystarczająco wydajny.
- Ograniczona funkcjonalność - pula dostępnej pamięci dostarczana przez SDDSfL może w takim rozwiązaniu być wykorzystana jedynie do składowania i zarządzania plikami. Dodatkowo każdy system plików stosuje pewien określony sposób zarządzania magazynowanymi danymi, który nie zawsze jest zgodny z wymogami aplikacji użytkowych.

Zaletami realizacji klienta SDDSfL w postaci sterownika urządzenia blokowego są:

- Analogie jednostek danych - w SDDS podstawową jednostką jest rekord, którego rozmiar może być stały, a dane w nim zawarte nie są przez SDDS interpretowane. W przypadku sterownika urządzenia blokowego podstawową jednostką danych są bloki o stałej wielkości. Ich zawartość również nie jest przetwarzana przez sterownik.
- Efektywność - sterowniki urządzeń blokowych są w Linuksie implementowane zawsze jako część jądra systemu operacyjnego, więc są równie wydajne jak wywołania systemowe.
- Możliwość implementacji w postaci modułu - tego typu oprogramowanie, podobnie jak sterowniki systemów plików może być ładowane i usuwane z systemu bez konieczności restartu komputera.
- Elastyczność - na urządzeniu tworzonym przez sterownik urządzenia blokowego można osadzić te same systemy plików, co na dyskach twardych, dobierając je do potrzeb aplikacji użytkownika. Dodatkowo urządzenie to może zostać wykorzystane do stworzenia przestrzeni wymiany dla stronicowania na żądanie.
- Interfejs - tak jak w przypadku systemów plików, każda aplikacja pracująca z plikami może bez modyfikacji korzystać z urządzenia blokowego. Jeśli tak zaimplementowany klient SDDSfL zostanie użyty w roli urządzenia wymiany, to jego interfejs dla aplikacji przestaje mieć znaczenie.

Wadami tego rozwiązania są:

- Niezawodność - tak, jak w przypadku innych elementów realizowanych na poziomie jądra systemu, błąd w takim oprogramowaniu destabilizuje pracę całego komputera.

- Brak możliwości implementacji w przestrzeni użytkownika - Linux jest systemem o jądrze monolitycznym [2–4, 89] i w przeciwieństwie do systemów takich jak MINIX 3 [89] nie pozwala na implementacje sterowników urządzeń blokowych w przestrzeni użytkownika. Taka możliwość pozwoliłaby sprawdzić działanie sterownika przed przeniesieniem jego implementacji na poziom jądra systemu.

Przedstawione powyżej argumenty przeważają na korzyść implementacji klienta SDDSfL w postaci sterownika urządzenia blokowego.

4.2.2. Serwery SDDSfL

Oprogramowanie serwerów SDDSfL może zostać zaimplementowane zarówno w przestrzeni użytkownika, jaki i w przestrzeni jądra systemu. Należy zatem rozważyć zalety i wady obu rozwiązań. Za serwerami w przestrzeni użytkownika przemawiają następujące względy:

- Niezawodność - niestabilna praca procesu serwera może doprowadzić do jego zakończenia, ale nie spowoduje to załamania pozostałych procesów użytkownika uruchomionych na tym samym komputerze (o ile nie są powiązane z procesem serwera) lub systemu operacyjnego.

- Łatwość implementacji - programista tworzący aplikacje użytkowe ma do dyspozycji gotowe biblioteki dostarczające wysokopoziomowych mechanizmów, które ułatwiają tworzenie programów. Również proces usuwania błędów łatwiej przeprowadzić w przestrzeni użytkownika, niż jądra sytemu.

Przeciw temu rozwiązaniu można przedstawić następujące argumenty:

- Przełączanie kontekstu - nadchodzące przez sieć komunikaty najpierw są odbierane przez jądro systemu, a następnie przekazywane do procesu użytkownika, wymaga to przełączenia kontekstu [2–4], co dodatkowo wydłuża czas obsługi komunikatu. Ponieważ jednak czas przełączenia kontekstu w systemie Linux jest stosunkowo krótki [30], to jego wpływ na długość trwania obsługi komunikatu jest niewielki.

- Wymiana stron - strony procesu, również te, które tworzą wiaderka, mogą zostać wycofane z RAM do urządzenia wymiany, którym najczęściej jest dysk twardy. To wpływa negatywnie na czas dostępu do danych zgromadzonych w wiaderku. Jeżeli jednak proces posiada odpowiednie przywileje może zapobiec wymianie swoich stron. Problem ten zostanie opisany w rozdziale poświęconym prototypowej implementacji SDDSfL.

Po stronie zalet serwerów rezydujących w przestrzeni jądra należy wymienić:

- Brak wymiany - wszystkie strony pamięci, które są przydzielone jądru Linuksa nie podlegają wymianie [2, 30]. Jeśli serwer SDDSfL rezydowałby w przestrzeni adresowej jądra, to dotyczyłoby to również stron przeznaczonych na wiaderka. Dzięki temu efektywny czas dostępu do danych zgromadzonych w wiaderku powinien być krótszy niż w przypadku procesów użytkownika.

- Brak przełączania kontekstu - jeżeli serwer znajduje się w przestrzeni adresowej jądra, to jest jego częścią i ma takie same uprawnienia jak ono, więc przełączanie kontekstu dla tego serwera nie jest konieczne.

Wady tego rozwiązania obejmują:

- Trudności w implementacji - jądro systemu operacyjnego oferuje programiście przeważnie niższy poziom abstrakcji niż przestrzeń użytkownika. Na przykład, w przypadku niektórych architektur sprzętowych (w tym 32-bitowych platform sprzętowych PC), ramki należące do tak zwanej „pamięci wysokiej" nie posiadają stałych adresów wirtualnych, co utrudnia korzystanie z dużych obszarów pamięci [30, 37].

- Niestabilność - błędy w kodzie powodują nie tylko awarię serwera, ale mogą również spowodować uszkodzenie procesów użytkownika i pozostałych elementów systemu operacyjnego. Proces wykrywania i usuwania błędów w przestrzeni jądra jest trudny.

Przedstawione argumenty przeważają na korzyść implementacji serwerów SDDSfL jako procesów użytkownika.

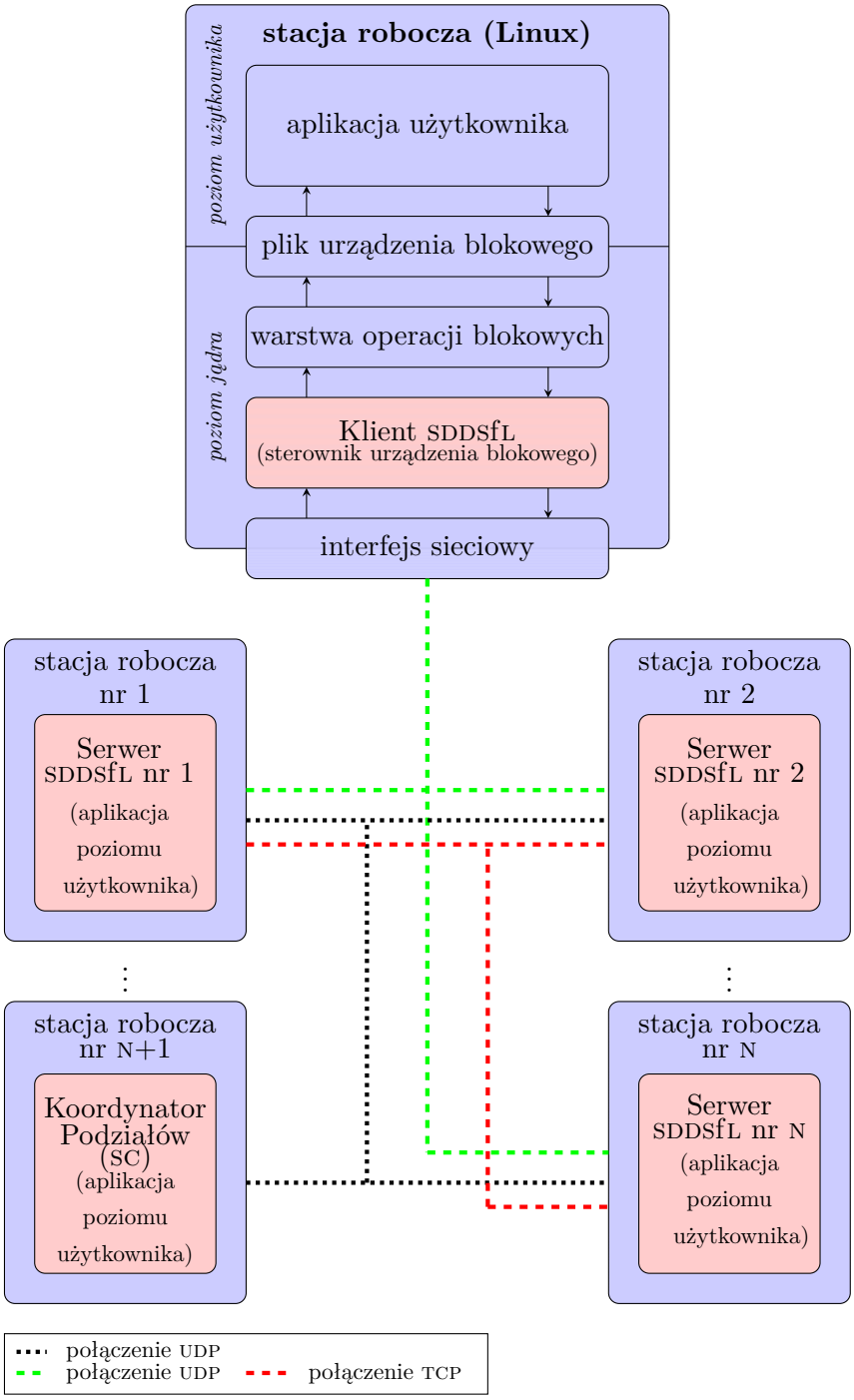
4.2.3. Koordynator podziałów

W przypadku koordynatora podziałów należy rozważyć, czy w ogóle ten element powinien być implementowany. Można go zastąpić żetonem wyznaczającym kolejność podziałów, który jest przekazywany między serwerami [27]. Jednak takie rozwiązanie nie podniosłoby znacząco ani wydajności, ani niezawodności SDDSfL, a koordynator podziałów odgrywa istotną rolę w architekturach tolerujących błędy.

Koordynator podziałów powinien być zrealizowany w przestrzeni użytkownika, gdyż jest to na tyle proste w działaniu i budowie oprogramowanie, że jego implementacja na poziomie jądra systemu operacyjnego nie przyniosłaby dodatkowych korzyści.

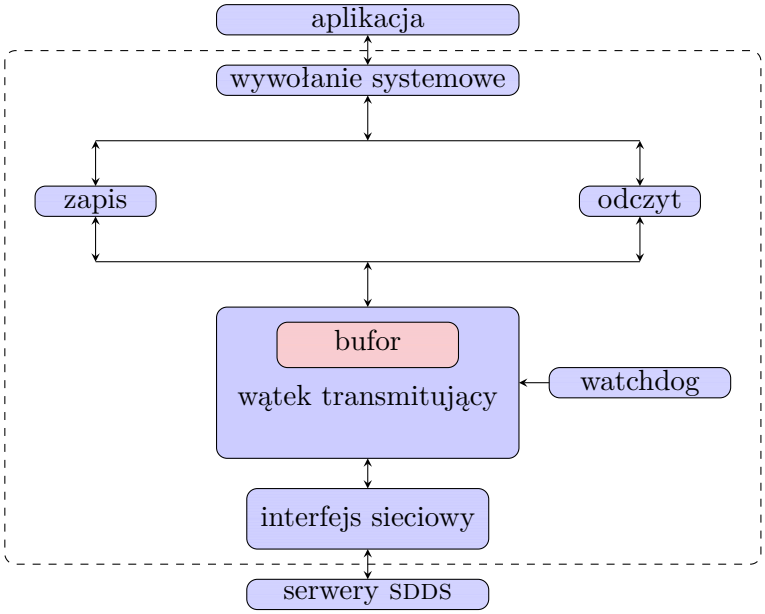
4.3. Schemat ogólny SDDSfL

Ogólny schemat Skalowalnych, Rozproszonych Struktur Danych dla systemu Linux (SDDSfL) przedstawia Rysunek 4.1. Serwery SDDSfL są oprogramowaniem pracującym w przestrzeni użytkownika i uruchamianym na osobnych węzłach systemu wielokomputerowego. Każdy serwer zarządza pojedynczym wiaderkiem. Rolę rekordów w wiaderkach pełnią bloki będące uporządkowanymi zbiorami przyległych sektorów. Identyfikatorem bloku jest numer pierwszego przynależnego sektora. Ponieważ sektory posiadają unikatowe numery i są numerowane liniowo, to również identyfikatory bloków są niepowtarzalne. Połączenia między serwerami, a klientem SDDSfL obsługiwane są za pomocą protokołu UDP/IP i za ich pomocą przekazywane są zarówno komunikaty z danymi, jak i z informacjami sterującymi. Podczas operacji podziału serwery tworzą połączenia między sobą, które używają protokołu TCP/IP i służą wyłącznie do przekazywania danych. Koordynator podziałów (SC) jest elementem SDDSfL, który podobnie jak serwery jest procesem działającym w przestrzeni użytkownika i uruchamianym na osobnym węźle multikomputera. SC komunikuje się z serwerami SDDSfL za pomocą protokołu UDP/IP. Koordynator wysyła i odbiera komunikaty zawierające jedynie informacje sterujące. Wszystkie powyżej opisane składowe architektury SDDSfL mają taką samą postać i funkcjonalność jak w klasycznej architekturze SDDS LH*. Różnica między obiema architekturami pojawia się w części klienckiej. W przypadku SDDSfL klient nie jest ani aplikacją użytkową ani warstwą pośrednią, lecz stanowi część jądra systemu operacyjnego. Jest on sterownikiem wirtualnego urządzenia blokowego, które w przestrzeni użytkownika reprezentowane jest przez specjalny plik. Klient SDDSfL postrzegany jest więc przez inne części systemu operacyjnego oraz aplikacje użytkowe jako urządzenie pamięci masowej z możliwością odczytu i zapisu. Dzięki temu może on posiadać własny, lokalny system plików lub być urządzeniem wymiany stosowanym w technice stronicowania na żądanie. Choć nie uwidoczniono tego na Rysunku 4.1, to SDDSfL może posiadać wielu klientów, rezydujących na różnych węzłach systemu wielokomputerowego, podobnie jak ma to miejsce



Rysunek 4.1: Architektura SDDSfL

w przypadku klasycznej implementacji SDDS. Wszystkie połączenia tworzone przez poszczególne elementy SDDSfL są połączeniami punktowymi (ang. *unicast*). Wszystkie połączenia wymagające wysyłania względnie krótkich komunikatów do wielu odbiorców, ale nie jednocześnie, są obsługiwane za pomocą protokołu UDP/IP. Połączenia służące do wysyłania większej ilości danych tylko do jednego odbiorcy (co ma miejsce podczas przeprowadzania operacji podziału wiaderka) używają protokołu TCP/IP.



Rysunek 4.2: Schemat organizacji klienta SDDSfL

4.4. Architektura klienta SDDSfL

Klient SDDSfL jest sterownikiem urządzenia, czyli oprogramowaniem uruchamianym na poziomie systemu operacyjnego węzła multikomputera [90]. Na Rysunku 4.2 przedstawiono schemat organizacji takiego sterownika. Po dokonaniu czynności inicjujących klient SDDSfL aktywowany jest przez aplikację, która żąda wykonania operacji wejścia-wyjścia. To żądanie przekazywane jest za pomocą wywołania systemowego związanego z plikiem urządzenia blokowego do Warstwy Operacji Blokowych systemu Linux. Stąd trafia ono do sterownika urządzenia [30, 41]. Sterownik przekształca żądania zapisu lub odczytu skierowane do urządzenia blokowego na komunikaty sieciowe. Każdy taki komunikat jest umieszczany w buforze. Za komunikację z serwerami SDDSfL odpowiedzialny jest w sterowniku wątek transmitujący. Pobiera on komunikaty z bufora i przesyła je do serwerów. Ponieważ ta komunikacja odbywa się przy pomocy protokołu UDP/IP, to istnieje ryzyko zagubienia komunikatu i zablokowania wątku w stanie oczekiwania na otrzymanie potwierdzenia odbioru przez serwer. Z tego względu sterownik musi zawierać mechanizm programowy, który po określonym czasie zarządzi retransmisję komunikatu (ang. *watchdog*). Jeśli żądanie dotyczyło zapisu, to po skończonej transmisji sterownik tylko potwierdza wykonanie zadania. Jeżeli wykonany miał być odczyt, to nadesłane przez serwer dane są przekazywane do warstwy operacji blokowych i ostatecznie do aplikacji użytkownika. Oddelegowanie prowadzenia transmisji do osobnego wątku nie jest koniecznością, ale ułatwia obsługę sytuacji wyjątkowych związanych z komunikacją oraz wprowadzanie modyfikacji do sterownika. Negatywną cechą takiej architektury jest możliwość wystąpienia niewielkiego spadku wydajności dla komputerów jednoprocesorowych i jednordzeniowych. Rozdzielenie ścieżki przetwarzania żądania zapisu i żądania odczytu też nie jest konieczne i również zostało wprowadzone celem ułatwienia ewentualnych modyfikacji struktury klienta SDDSfL. Działanie klienta SDDSfL opisane jest deterministycznym automatem skończonym, którego diagram przedstawia Rysunek 4.3. Oznaczenia wejść i stanów mają następującą interpretację:

q_0 oczekiwanie na pojawienie się żądania odczytu lub zapisu,

z_1 żądanie odczytu/ zapisu,

q_1 przygotowanie komunikatu,

z_2 wysłanie komunikatu,

q_2 oczekiwanie na potwierdzenie odbioru i dane w przypadku żądania odczytu,

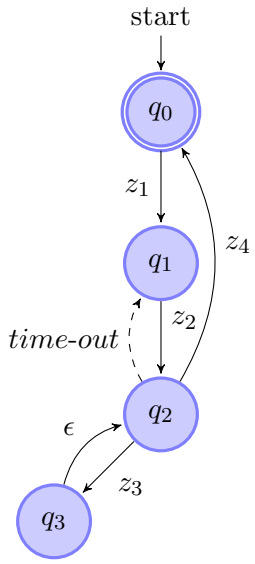
z_3 otrzymanie komunikatu IAM,

q_3 aktualizacja obrazu pliku (i' i n'),

z_4 otrzymanie potwierdzenia lub danych w przypadku żądania odczytu,

ϵ przejście bezwarunkowe do określonego stanu,

time-out przejście do określonego stanu jeśli nie pojawi się żaden inny sygnał wejściowy i upłynie zadany czas; sygnał *time-out* jest generowany przez układ watchdog będący integralną częścią klienta.



Rysunek 4.3: Automat skończony - wątek transmitujący klienta SDDSfL

4.5. Architektura serwera SDDSfL

Przyjęty schemat ogólny budowy SDDSfL zakłada zrealizowanie serwerów w postaci procesów użytkownika. Aby ułatwić obsługę komunikacji sieciowej proces serwera może zostać podzielony na trzy wątki: wątek główny obsługujący klienta, wątek odbierający dane od serwera, którego wiaderko podlega podziałowi oraz wątek wykonujący podział. Działanie pierwszych dwóch wątków opisane jest deterministycznym automatem skończonym o diagramie, który przedstawia Rysunek 4.4, przy czym stany i wejścia mają następujące znaczenie:

q_0 oczekiwanie na dane z podzielonego wiaderka,

z_1 nadesłanie danych z podzielonego wiaderka,

q_1 oczekiwanie na komunikat od klienta,

z_2 odebranie komunikatu od klienta,

q_2 weryfikacja poprawności i stanu zapełnienia wiaderka,

z_3 nieprawidłowy adres,

q_3 przesłanie komunikatu do innego serwera,

q_4 wysłanie IAM klientowi,

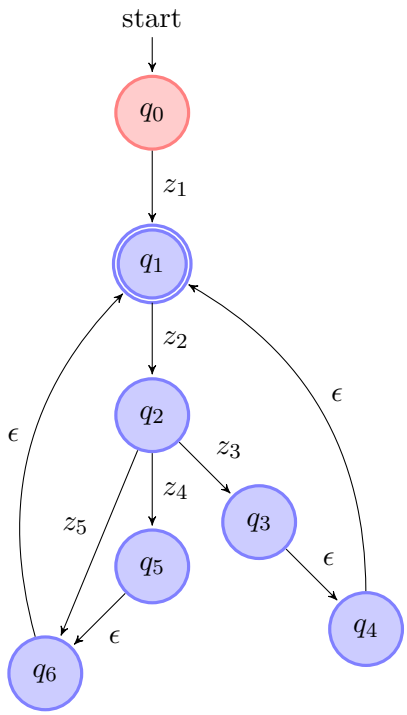
z_4 przepełnienie,

q_5 wysłanie komunikatu „collision” do SC,

z_5 prawidłowy adres danych,

q_6 przetworzenie danych od klienta (odczyt lub zapis) i wysłanie odpowiedzi,

ϵ przejście bezwarunkowe do określonego stanu.



Rysunek 4.4: Automat skończony - serwer SDDSfL

Stan q_0 i wejście z_1 związane są z wątkiem oczekującym na dane z podziału wiaderka. Nie występują one w przypadku serwera zarządzającego pierwszym wiaderkiem, czyli tym, które ma adres logiczny $a = 0$. Dla niego pierwszym stanem jest stan q_1.

Trzeci wątek wykonuje operację podziału, która obejmuje czynności opisane w Podrozdziale 4.1.

4.6. Architektura koordynatora podziałów SDDSfL

Koordynator podziałów jest procesem użytkownika, którego działanie można w uproszczony sposób opisać skończonym automatem deterministycznym, którego graf przedstawia Rysunek 4.5, gdzie:

q_0 oczekiwanie na komunikat o kolizji (przepełnieniu) wiaderka,

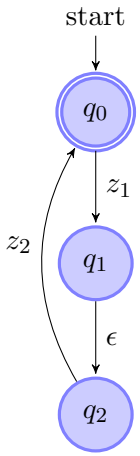
z_1 otrzymanie komunikatu o przepełnieniu,

q_1 wyznaczenie zgodnie z Algorytmem 5 wiaderka do podziału i wysłanie serwerowi, który nim zarządza komunikatu „you split”,

q_2 oczekiwanie na zakończenie operacji podziału wiaderka,

z_2 otrzymanie komunikatu o zakończeniu dzielenia wiaderka,

ϵ przejście bezwarunkowe do określonego stanu.



Rysunek 4.5: Automat skończony - koordynator podziałów (SC)

Oczekiwanie na potwierdzenie przez serwer zakończenia operacji podziału wiaderka ma istotne znaczenie dla poprawności aktualizacji obrazu klienta. Dzięki temu potwierdzeniu wszystkie podziały są uszeregowane, co oznacza, że nowy podział nie zostanie rozpoczęty do momentu zakończenia poprzedniego. Pozwala to zachować poprawne poziomy wiaderek w pliku SDDS. Gdyby ta poprawność została naruszona, to komunikaty IAM podchodzące od serwera, którego wiaderko ma nieprawidłowy poziom (np. wyższy o 2 od pozostałych wiaderek) doprowadziłyby do zafałszowania obrazu pliku posiadanego przez klienta. To prowadziłoby do przypadków wysyłania przez klienta żądań do wiaderek, które nie zostały jeszcze utworzone.

4.7. Protokoły SDDSfL

W komunikacji między składowymi SDDSfL wykorzystywane są protokoły TCP/IP i UDP/IP warstwy transportowej [91]. Służą one do przesyłania pakietów protokołów warstwy aplikacji, które zostały zaprojektowane na potrzeby tego rozwiązania i które zostaną opisane w kolejnych częściach tego podrozdziału.

4.7.1. Komunikacja klient-serwer

Pakiety przesyłane są za pomocą bezpołączeniowego protokołu UDP/IP. Jeśli wysyłane są przez klienta to zawierają jego numer będący liczbą naturalną, adres IP w postaci łańcucha znaków zawierającego cztery liczby dziesiętne rozdzielone znakami kropki, znacznik typu operacji

(zapis/odczyt), numer sektora identyfikujący blok danych oraz wielkość danych, których operacja ma dotyczyć. Jeśli pakiet dotyczy operacji zapisu danych do wiaderka, to zawiera również te dane. Adres IP klienta jest wykorzystywany, kiedy serwery muszą przesłać między sobą nieprawidłowo zaadresowane żądanie. Pakiet opisujący to żądanie oznaczany jest specjalną flagą pozwalającą rozpoznać prawidłowemu odbiorcy, że odpowiedź powinien wysłać nie bezpośredniemu nadawcy, lecz klientowi SDDSfL.

Serwer SDDSfL w odpowiedzi na komunikat klienta wysyła potwierdzenie wykonania operacji, które zawiera znacznik potwierdzenia, numer sektora identyfikujący blok danych, którego dotyczyła operacja oraz liczbę bajtów danych zaangażowanych w tą operację. Jeśli żądanie dotyczyło odczytu danych z wiaderka to te dane są dołączane do potwierdzenia. Jeśli klient popełnił błąd adresowania, to serwer wysyła mu pakiet zawierający poziom wiaderka oraz jego adres logiczny, zgodnie z opisem z Podrozdziału 4.1. Listing 4.1 zawiera model protokołu połączenia klient-serwer zapisany w języku Promela [92]. Procesy klienta i serwera zostały zaimplementowane w postaci namiastek (ang. *stub*), które wykonują tylko te operacje, które są niezbędne do prowadzenia komunikacji.

Listing 4.1: Model protokołu połączenia klient-serwer

```
mtype = {read, write, read_ack, write_ack};

typedef header {
        mtype type;
        byte client_nr, flags, ip[16];
        int begin, iam[2], number;
};

typedef request {
        header hdr;
        byte data[4096];
};

chan fs = [0] of {request};
chan ts = [0] of {request};

proctype client_stub(chan in, out)
{
  request rq;
  do
  :: do
     :: /* Read request. */
        rq.hdr.type = read;
        out ! rq;
        in ? rq;
        rq.hdr.type == read_ack -> break;
     od
  :: do
     :: /* Write request. */
        rq.hdr.type = write;
        out ! rq;
        in ? rq;
        rq.hdr.type == write_ack -> break;
     od
  od
}

proctype server_stub(chan in, out)
{
  request rq;
```

```
  do
  :: in ? rq;
     if
     :: rq.hdr.type == read ->
          rq.hdr.type = read_ack;
          out ! rq;
     :: rq.hdr.type == write ->
          rq.hdr.type = write_ack;
          out ! rq;
     fi
  od
}

init{run client_stub(fs,ts); run server_stub(ts,fs);}
```

4.7.2. Komunikacja serwer-serwer

Tego typu połączenia wykorzystywane są podczas przeprowadzania operacji podziału do przesyłania danych dla nowego wiaderka. Dane te przesyłane są strumieniem za pomocą protokołu TCP/IP. Jako pierwsza przesyłana jest wartość poziomu nowego wiaderka. Następnie przesyłane są właściwe dane poprzedzone nagłówkiem. Nagłówek ten zawiera flagę określającą, czy są to ostatnie dane wysłane podczas transmisji, numer sektora określający blok, do którego należą dane, informacje indeksujące dane w wewnętrznej strukturze wiaderka oraz numer klienta SDDSfL będącego właścicielem tych danych. Listing 4.2 przedstawia model opisywanego protokołu zapisany w języku Promela. Serwery uczestniczące w komunikacji są zaimplementowane w postaci namiastek, których działanie ogranicza się do transmisji komunikatów.

Listing 4.2: Model protokołu połączenia serwer - serwer

```
typedef split_data {
        bool end;
        byte sector,index,client;
        byte data[4096];
};

chan link = [0] of {byte, split_data};

proctype Send(chan out)
{       byte j = 0; /* Bucekt level */
        split_data msg;

        j++;
        out ! j;
        do
        :: /* Get data from bucket. */
           out ! msg
        :: msg.end = true;
           out ! msg;
           break
        od
}

proctype Receive(chan in)
{
     byte j;
     split_data msg;
```

```
        in ? j;
        do
        :: in ? msg;
           /* Put data into bucket. */
           if
           :: msg.end == true -> break
           :: else -> skip;
           fi
        od
}

init {run Send(link); run Receive(link);}
```

4.7.3. Komunikacja koordynator podziałów-serwer

Ten rodzaj komunikacji zachodzi z użyciem protokołu UDP/IP i polega on na przesyłaniu prostych znaczników między obiema stronami. Pakiet rozpoczynający komunikację pochodzi zawsze od serwera i jest to znacznik informujący o wystąpieniu przepełnienia wiaderka. W przypadku, gdyby podziały były realizowane w sposób kontrolowany komunikat zawierałby dodatkowo informacje o poziomie wiaderka, współczynniku jego załadowania oraz adres logiczny serwera. Koordynator potwierdza nadawcy odebranie informacji o kolizji wysłaniem pakietu ze znacznikiem powiadamiający o przyjęciu komunikatu „collision". Wysłanie takiego potwierdzenia nie zostało ujęte w grafie automatu z Rysunku 4.5, gdyż jest ono podyktowane użyciem protokołu UDP/IP, który nie gwarantuje dostarczenia pakietu odbiorcy. Takiej konieczności można uniknąć stosujący protokół warstwy transportowej gwarantujący niezawodność transmisji. Po otrzymaniu informacji o kolizji SC wyznacza adres serwera, którego wiaderko powinno się podzielić i wysyła mu komunikat „you split" nakazujący podział. Następnie przechodzi w stan oczekiwania na informację od serwera, że dzielenie wiaderka zostało zakończone. Po jej otrzymaniu SC powraca do stanu oczekiwania na komunikat o przepełnieniu wiaderka. Model opisywanego protokołu, zapisany w języku Promela, przedstawia Listing 4.3. Koordynator podziałów został zaimplementowany w całości, natomiast serwer jest namiastką wykonującą jedynie te operacje, które niezbędne są z punktu widzenia komunikacji.

Listing 4.3: Model protokołu połączenia koordynator podziałów - serwer

```
mtype = {collision, yousplit, split_ack, collision_ack}

chan svch = [0] of {mtype};
chan scch = [0] of {mtype, byte};

proctype sc(chan in, out)
{
        byte i = 0, n = 0;
        bool no_split = true;
        do
        :: in ? collision ->
                out ! collision_ack;
                if
                :: no_split == true ->
                        out ! n,yousplit;
                        no_split = false;
                        n++;
                        if
                        :: n >= (i<<i) ->
                                n = 0;
                                i++;
```

```
                        :: else -> skip
                        fi
                :: else -> skip
                fi
        :: in ? split_ack -> no_split = true
        od
}

proctype server_stub(chan in, out; byte id)
{
        do
        :: out ! collision
        :: in ? collision_ack -> skip
        :: in ? eval(id),yousplit -> out!split_ack
        od
}

init { run sc(svch,scch);
        run server_stub(scch,svch,0); run server_stub(scch,svch,1);
        run server_stub(scch,svch,2); run server_stub(scch,svch,3); }
```

4.8. Podsumowanie

W tym rozdziale przedstawiono architekturę SDDSfL, która jest realizacją zawartej w tezie rozprawy koncepcji przeniesienia SDDS na poziom systemu operacyjnego. W wyniku przeprowadzonej analizy możliwości zaprojektowania rozwiązania spełniającego postawione wymogi przyjęto, że klient SDDSfL zostanie zrealizowany w postaci oprogramowania będącego częścią jądra systemu operacyjnego, a serwery i koordynator podziałów będą zaimplementowane jako aplikacje poziomu użytkownika. Pozwoli to zmniejszyć liczbę problemów, które pojawiłyby się, gdyby wszystkie elementy architektury musiały zostać zrealizowane na poziomie systemu operacyjnego. Jednocześnie, to rozwiązanie nie wpływa negatywnie na wydajność SDDSfL. Podjęto również decyzję o realizacji klienta w postaci sterownika urządzenia blokowego. Zwiększa to liczbę możliwych zastosowań SDDSfL oraz umożliwia ładowanie i usuwanie oprogramowania klienckiego bez konieczności restartu systemu komputerowego. Dodatkowo wszystkie aplikacje użytkowe, które korzystają z plików mogą współpracować z SDDSfL bez konieczności ich modyfikacji.

5. Implementacja SDDSfL

Ten rozdział zawiera opis prototypowej implementacji architektury SDDSfL przedstawionej w Rozdziale 4. Pierwszy podrozdział stanowi krótkie wprowadzenie do zagadnień związanych z implementacją SDDSfL w systemie Linux. Drugi poświęcony jest klientowi, trzeci traktuje o realizacji serwera, czwarty o koordynatorze podziałów, a piąty o realizacji protokołów warstwy aplikacji opisanych w Podrozdziale 4.7. Następny podrozdział dotyczy odporności na awarie prototypu SDDSfL. Rozdział kończy się krótkim podsumowaniem.

5.1. Wprowadzenie

Wszystkie elementy prototypowej implementacji SDDSfL zostały stworzone przy użyciu języka C, którego wybór podyktowany był potrzebą zaimplementowania klienta na poziomie jądra systemu operacyjnego.

Klient SDDSfL został zrealizowany w postaci modułu jądra, który jest sterownikiem wirtualnego urządzenia blokowego. Komunikacja z innymi węzłami multikomputera dokonywana jest w tym module przy pomocy gniazd obsługiwanych za pomocą niskopoziomowego API. Oznacza to nie tylko różnice w nazwach i argumentach poszczególnych funkcji, ale również konieczność napisania odpowiedników niektórych podprogramów, które dostępne są na poziomie aplikacji użytkowych. Wymiana informacja za pomocą sieci komputerowej zwiększa prawdopodobieństwo powstania błędów w jądrze, co może prowadzić do załamania (ang. *crash*) systemu. Zgodnie z projektem klienta SDDSfL, który został przedstawiony w Podrozdziale 4.4 ograniczono skutki wystąpienia błędu związanego z komunikacją umieszczając kod odpowiedzialny za transmisję w osobnym wątku oraz realizując mechanizm retransmisji zagubionych komunikatów za pomocą kolejek prac (ang. *work queue*). Wprowadzenie wątku transmisji wymagało uwzględnienia faktu, że wywłaszczanie wątków jądra jest w systemie Linux opcjonalne oraz opracowania schematu synchronizacji między wątkiem i pozostałymi częściami sterownika[1]. Decyzje podjęte na tym etapie implementacji mają wpływ na efektywność działania klienta SDDSfL. Innym czynnikiem, od którego zależy jego wydajność, jest maksymalny rozmiar jednostki danych przesyłanych między klientem, a serwerami SDDSfL. W przypadku urządzeń blokowych w Linuksie jest on ograniczony rozmiarem strony. Konieczne również było opracowanie komunikacji między sterownikiem, a procesami przestrzeni użytkownika. Szczegóły implementacji klienta znajdują się w Podrozdziałach 5.2 i 5.6.

Przy opracowywaniu implementacji serwera SDDSfL kluczowe decyzje dotyczyły określenia struktury wewnętrznej wiaderka. Od niej zależy wydajność lokalizacji danych oraz operacji podziału. Opis implementacji znajduje się w Podrozdziale 5.3.

Najważniejszym czynnikiem, który miał wpływ na implementacje koordynatora podziałów jest zawodność protokołu UDP/IP. Szczegóły zostały opisane w Podrozdziale 5.4.

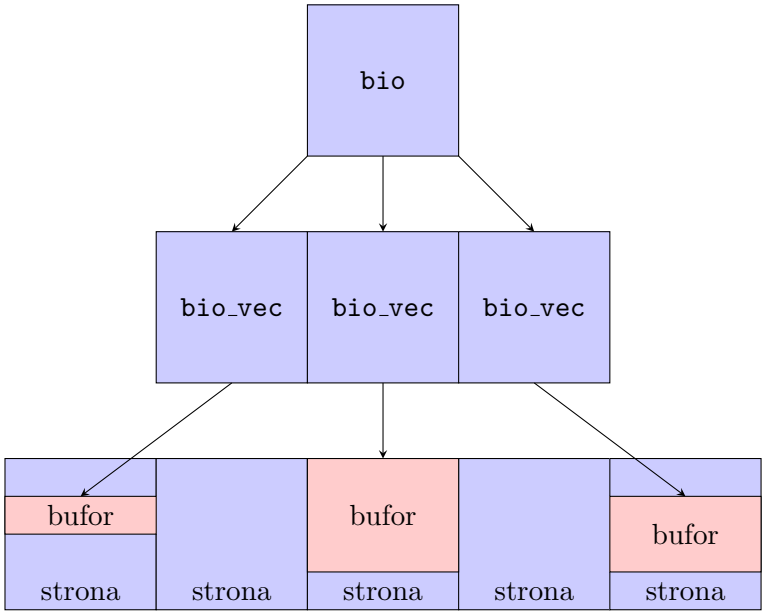
[1] Wywłaszczanie wątków może zostać włączone lub wyłączone na etapie kompilacji jądra systemu.

5.2. Implementacja klienta SDDSfL

Klient SDDSfL jest sterownikiem urządzenia będącym częścią jądra systemu operacyjnego. W przypadku systemu Linux tego typu oprogramowanie jest często realizowane w postaci modułu, który może być załadowany lub usunięty z jądra bez konieczności restartu systemu [39]. Moduł ten udostępnia aplikacjom użytkownika urządzenie blokowe, które jest plikiem SDDSfL i z którym komunikacja jest dokonywana za pomocą sieci komputerowej [30, 39, 40]. Tego typu rozwiązania są wykorzystywane również w innych zastosowaniach [93, 94]. Aplikacje użytkowe uzyskują dostęp do urządzeń blokowych za pomocą takich funkcji jak `read()` i `write()`, które z kolei aktywują wywołania systemowe o podobnych nazwach [30]. Te ostatnie korzystają z funkcji Wirtualnego Systemu Plików (ang. *Virtual File System* - VFS), by określić rzeczywisty system plików, do którego należy przesłać żądanie aplikacji. Każdy kod odpowiedzialny za obsługę rzeczywistego systemu plików posiada własne procedury pozwalające ustalić, których bloków na nośniku danych należy użyć, aby spełnić to żądanie. Informacje o tych blokach są następnie przekazywane Warstwie Operacji Blokowych, która odpowiada w jądrze Linuksa za ich przekształcenie do postaci zlecenia dla sterownika urządzenia blokowego [30,41]. W szczególności Warstwa Operacji Blokowych (ang. *Block Operations Layer*) tworzy struktury BIO (ang. *Block* I/O) opisujące operacje, które należy wykonać na przyległych blokach. Pojedyncze żądanie nie musi dotyczyć ciągłych obszarów na nośniku danych, które są tworzone przez przyległe bloki. Dla takiego żądania może być utworzonych wiele struktur `bio`, które są łączone w większą strukturę o nazwie `request`. Te struktury są łączone w kolejkę, która zanim zostanie przetworzona przez sterownik urządzenia blokowego jest porządkowana przez planistę wejścia-wyjścia (ang. I/O *scheduler*) [2–4, 30].

Powyższy opis przebiegu operacji blokowych pomija wiele szczegółów, z których kilka jest istotnych dla treści tego podrozdziału. Nie wszystkie urządzenia blokowe potrzebują porządkowania kolejki żądań przez planistę wejścia-wyjścia. Takie działanie korzystne jest w przypadku urządzeń, dla których czas dostępu do poszczególnych lokacji (bloków) danych tylko w przybliżeniu jest zawsze taki sam, a w rzeczywistości zależy od położenia danego bloku na nośniku. Takimi urządzeniami są dyski twarde. W przypadku plików SDDS czasy dostępu do poszczególnych bloków zależą od algorytmów użytych do wyszukiwania lokacji danych w RAM, ale ich wariancja jest na tyle mała w porównaniu z dyskami twardymi, że stosowanie planisty wejścia-wyjścia nie jest uzasadnione. Jeżeli nie jest używany taki planista, to również nie jest potrzebna kolejka struktur `request` ani nawet takie struktury. Sterownik urządzenia blokowego może operować bezpośrednio na strukturach `bio`. Budowa takiej struktury wymaga więc dokładniejszego opisu. Jej schemat pokazuje Rysunek 5.2. Jądro systemu Linux rezerwuje pewną liczbę stron pamięci operacyjnej na bufory I/O, które są powiązane z blokami danych na nośniku. Ponieważ rozmiar bufora nie może przekroczyć wielości strony, która w Linuksie określona jest stałą PAGE_SIZE, to również blok danych urządzenia nie może być większy. Minimalny rozmiar bufora jest równy rozmiarowi pojedynczego sektora tworzącego blok. W przypadku architektur PC te wartości wynoszą 4096 bajtów (wielkość strony) i 512 bajtów (rozmiar sektora). Oznacza to, że blok może składać się maksymalnie z ośmiu sektorów. W operacji wejścia-wyjścia może uczestniczyć tylko fragment bufora składający się z kilku przyległych sektorów, nazwany segmentem[2]. Segment może również opisywać obszar składający się z fragmentów przyległych do siebie buforów. Położenie pojedynczego segmentu opisuje struktura `bio_vec` za pomocą trzech wartości: adresu strony, przesunięcia względem jej początku oraz rozmiaru segmentu. Zawiera ona również informacje o położeniu na nośniku fizycznym odwzorowanego w buforze bloku. Każda struktura `bio` dysponuje listą struktur `bio_vec`, która opisuje obszar pamięci złożony z segmentów. Ten obszar może być nieciągły. Sterownik urządzenia blokowego przetwarzający bezpośrednio struktury `bio`

[2]Segmenty nie zostały pokazane na Rysunku 5.2.

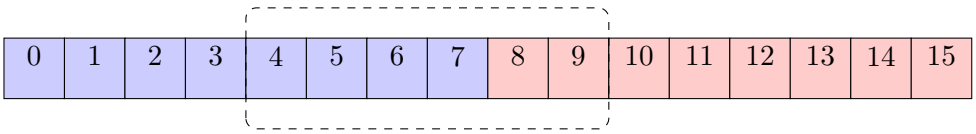


Rysunek 5.1: Struktura `bio`

musi implementować funkcję `make_request()`. Ta funkcja wykonuje operację wejścia-wyjścia, którą opisuje struktura `bio`. W ogólnym ujęciu jej działanie sprowadza się do określenia kierunku operacji (zapis/odczyt) za pomocą wywołania funkcji `bio_data_dir()`, wykonaniu odczytu lub zapisu danych z każdego segmentu opisywanego przez listę struktur `bio_vec` oraz powiadomieniu Warstwy Operacji Blokowych o stanie zakończenia operacji reprezentowanej przez strukturę `bio` za pomocą wywołania funkcji `bio_endio()`. Sposób realizacji odczytu lub zapisu segmentów zależny jest od budowy urządzenia blokowego.

W przypadku klienta SDDSfL istnieją dwie osobne funkcje wywoływane przez implementację funkcji `make_request()`. Jedna z nich związana jest z obsługą odczytu segmentu, a druga z obsługą zapisu. Każda przygotowuje pakiet danych opisujący żądanie, które ma zostać skierowane do serwera SDDSfL i zapisuje go do bufora, z którego pobiera go wątek transmitujący. Architektura tego wątku została opisana w Podrozdziale 4.4. Ponieważ obie funkcje korzystają z tego samego bufora i są wykonywane współbieżnie, to konieczna jest synchronizacja ich działania, która zapewniona została za pomocą muteksu (ang. *mutex*). Funkcje muszą również czekać na zakończenie przesyłania pakietu przez wątek transmitujący. Oczekiwanie to zrealizowano stosując zmienną sygnałową (ang. *conditional variable*).

Ze względu na organizację pliku SDDSfL funkcje obsługi odczytu i zapisu segmentów muszą dokonywać fragmentacji niektórych żądań. Istotę problemu ilustruje Rysunek 5.2. Serwery SDDSfL przechowują bloki danych składające się z ośmiu sektorów, których unikatowym identy-



Rysunek 5.2: Fragmentacja żądania

fikatorem jest, tak jak to opisano w Podrozdziale 3.2 numer pierwszego sektora. Te numery są wielokrotnościami liczby osiem (0,8,16,24,32, ...). Żądanie opisywane przez pojedynczą strukturę `bio_vec`, które na Rysunku 5.2 oznaczono wyoblonym prostokątem narysowanym przerywaną

linią, może dotyczyć odczytu lub zapisu kilku sektorów znajdujących się w sąsiednich blokach. W przypadku pliku SDDSfL te bloki nie muszą znajdować się w tym samym wiaderku i powstaje konieczność wysłania zapytań do dwóch różnych serwerów SDDSfL. Takie żądanie można wykryć sprawdzając, czy pierwszy sektor jakiego ono dotyczy jest również pierwszym sektorem w bloku. W tym celu należy sprawdzić, czy jego numer jest podzielny przez osiem. Jeśli nie, to trzeba zbadać, czy segment opisany w żądaniu mieści się w bloku, do którego należy sektor. Jeżeli i tym razem odpowiedź będzie negatywna, to żądanie jest dzielone na dwa fragmenty. Pierwszy obejmuje sektory od początkowego do tego, którego numer podzielny jest przez osiem, ale bez niego. Drugi fragment zawiera ten sektor i pozostałe. Ponieważ pojedyncze żądanie może dotyczyć maksymalnie ośmiu sektorów, to liczba fragmentów nigdy nie przekracza dwóch.

Przesłaniem żądania do serwera SDDSfL zajmuje się wątek transmitujący. Jest to wątek jądra, który jeśli nie ma zleconego żadnego zadania, to oczekuje w funkcji `wait_event_interruptible()` na aktywację przez funkcję obsługi odczytu lub zapisu segmentu. Jeśli otrzyma zlecenie, to wyznacza adres logiczny wiaderka, do którego skierowane jest żądanie, stosując Algorytm 1 z parametrami i' i n', który opisano w Podrozdziale 4.1. Ponieważ numery pierwszych sektorów, stanowiące identyfikatory bloków, są podzielne przez osiem, to jako klucz C w algorytmie adresowania używana jest wartość numeru pierwszego sektora, którego dotyczy żądanie, przesunięta o trzy bity w prawo (podzielona przez 8). Na podstawie adresu logicznego wiaderka i za pomocą tablicy T wątek wyznacza adres fizyczny (IP) serwera i wysyła do niego żądanie, a następnie oczekuje na odpowiedź. Jeśli będzie to komunikat IAM, to wątek uaktualni swój obraz pliku i powróci do oczekiwania na właściwy pakiet. Po otrzymaniu informacji związanej z realizacją żądania powiadamia za pomocą wspomnianej zmiennej sygnałowej funkcje odczytu lub zapisu segmentu, że jest gotów do obsłużenia kolejnych żądań, a w przypadku odczytu, że są już dostępne zażądane dane. Wątek sprawdza poprawność odpowiedzi badając trzy atrybuty otrzymanego pakietu: typ, numer identyfikacyjny bloku oraz rozmiar danych, których dotyczyło wysłane do serwera żądanie.

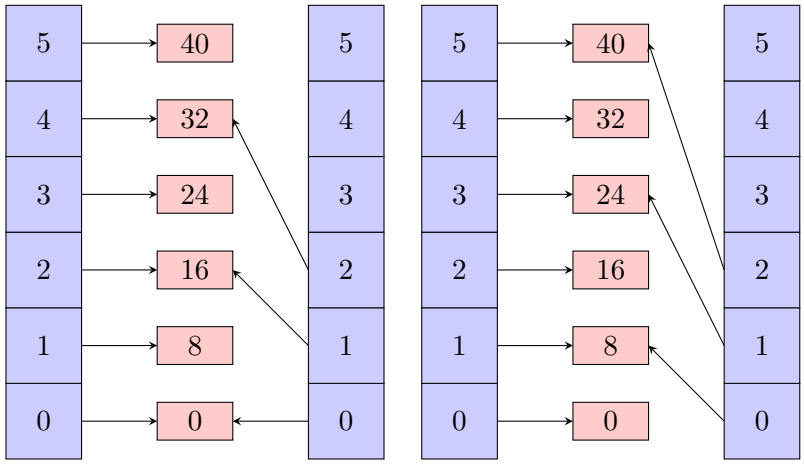
Komunikacja z serwerem bazuje na protokole UDP/IP i odbywa się za pomocą niskopoziomowych funkcji obsługi gniazd. Nadanie pakietu wykonuje funkcja o nazwie `udp_sendmsg()`, a za odbiór odpowiedzialna jest funkcja `recvmsg()`. Stworzenie gniazda do komunikacji następuje podczas inicjacji modułu, kiedy również tworzone jest wirtualne urządzenie blokowe. Wspomniana tablica przydziałów fizycznych (T) wypełniana jest za pomocą interfejsu z przestrzenią użytkownika, jaki stanowi system plików `sysfs`. Za pomocą plików stworzonych przez moduł w odpowiednich podkatalogach katalogu `/sys` użytkownik może określić maksymalną liczbę potencjalnych serwerów SDDSfL, ich adresy fizyczne oraz przekazać klientowi jego adres logiczny (liczba naturalna nazywana dalej numerem klienta) wraz z adresem IP komputera, na którym został uruchomiony. Po zakończeniu inicjacji te pliki są usuwane przez sterownik, aby zapobiec przypadkowym zmianom zawartości tablicy T podczas jego działania.

5.3. Implementacja serwera SDDSfL

Tak jak opisano to w Podrozdziale 4.5 serwer SDDSfL jest procesem użytkownika podzielonym na trzy wątki. Wątek główny obsługuje komunikację klient-serwer, a dwa pozostałe odpowiedzialne są za przeprowadzenie operacji podziału. Wszystkie te wątki są tworzone po uruchomieniu serwera, w trakcje jego inicjacji. Wyjątkiem jest pierwszy serwer zarządzający wiaderkiem o adresie logiczny równym zero. W jego przypadku nie jest tworzony wątek nasłuchujący na dane z podziału innego wiaderka. Wszystkie wątki serwera są tworzone za pomocą biblioteki `pthreads`, która jest zgodna ze standardem POSIX [95].

Wewnętrznie dane zmagazynowane w wiaderku przechowywane są w osobnych obszarach pamięci o wielkości 4 KiB, które są alokowane dynamicznie. Aby zapewnić, że strony, na których

się one znajdują nie zostaną usunięte z RAM w ramach wymiany (ang. *swap*), to są one blokowane (ang. *lock*) w pamięci operacyjnej za pomocą wywołania funkcji `mlockall()`. Te obszary są odpowiednikami buforów wejścia-wyjścia jądra sytemu operacyjnego, ale należą do przestrzeni adresowej użytkownika. Dla uproszczenia będą one również nazywane buforami. Do zarządzania tymi obszarami służą trzy tablice, które także są tworzone dynamicznie. Każdy element tych tablic zawiera wskaźnik na bufor danych, numer klienta, który jest właścicielem tego bufora, znacznik stanu oraz klucz identyfikujący dane przechowywane w tym buforze. Numer klienta jest wykorzystywany w sytuacji, kiedy z pliku SDDSfL korzysta wielu klientów. Ten przypadek będzie opisany dalej w tym podrozdziale. Aby uprościć opis najpierw zostanie przedstawione działanie serwera, kiedy SDDSfL używa tylko jeden klient. Znacznik stanu określa, czy w obszarze pamięci znajdują się informacje pochodzące od klienta, czy też jest on nieużywany, wypełniony domyślnie zerami. Klucz jest przesuniętym o trzy bity w prawo (podzielonym przez osiem) numerem pierwszego sektora identyfikującego blok danych zapisany w buforze. Jedna z trzech tablic jest tablicą główną i zawiera wskaźniki na wszystkie bufory, które zostały utworzone w pamięci komputera. Druga zawiera wskaźniki na te bufory, których zawartość pozostanie w pamięci węzła po przeprowadzeniu operacji podziału. Elementy trzeciej tablicy wskazują bloki, których zawartość zostanie wysłana do nowo utworzonego wiaderka w ramach podziału. Rysunek 5.3 ilustruje powiązania między tymi tablicami. Kolorem czerwonym zaznaczono bufory przechowujące dane. Liczby wewnątrz buforów są numerami sektorów identyfikującymi umieszczone w buforach bloki danych. Cześć 5.3a rysunku pokazuje tablicę główną i tablicę przechowującą wskaźniki na dane, które pozostaną w wiaderku po podziale (tablica następna), natomiast część 5.3b przedstawia tablicę główną i tą, która wskazuje dane do wysłania podczas podziału do nowego wiaderka (tablica podziału).



(a) Główna tablica i tablica następna (b) Główna tablica i tablica podziału

Rysunek 5.3: Zależności między tablicami w wiaderku

Dane w tych tablicach są adresowane przez wątek główny serwera z użyciem odmiany techniki haszowania, która nazwana została adresowaniem otwartym [33,83]. W adresowaniu otwartym kolizje, czyli sytuacje, w których dwa lub więcej obiektów danych odwzorowywanych jest na ten sam element tablicy, rozwiązywane są poprzez ponowne wyszukanie pustego elementu, do którego będzie można wstawić dane. Tą operację można zrealizować na kilka sposobów. W prototypie SDDSfL wybrano metodę haszowania podwójnego. Algorytm 6 opisuje adresowanie otwarte z haszowaniem podwójnym. W przestawionym algorytmie zmienna A jest tablicą, i i j jej indeksami, M jej rozmiarem (liczbą elementów), δ wartością funkcji haszującej h_2, a K kluczem identyfikującym dane. Przyjęto również, że słowo kluczowe **return** oznacza zwrócenie

Algorytm 6 Adresowanie otwarte z haszowaniem podwójnym

```
i ← h1(K)
if (A[i].state=empty) or ((A[i].state = occupied) and (A[i].key = K)) then
  return i
end if
δ ← h2(K)
j ← i
repeat
  j ← j − δ
  if j < 0 then
    j ← j + M
  end if
until (A[j].state=empty) or ((A[j].state = occupied) and (A[j].key = K))
return j
```

wartości i zakończenie wykonania algorytmu. Jeśli adresowany jest element w tablicy głównej, to funkcja h_1 wyrażona jest Wzorem 5.1. W tym wzorze S jest numerem pierwszego sektora bloku, którego dotyczy realizowane żądanie zapisu, >> przesunięciem bitowym w prawo o zadaną liczbę pozycji, a j poziomem wiaderka.

$$h_1(\mathrm{S}) \leftarrow \mathrm{S} >> (3 + j) \bmod M \tag{5.1}$$

Liczba elementów tablicy (M) powinna być tak dobrana, aby była potęgą dwójki[3]. Wówczas wartość funkcji h_2 można wyliczyć według Algorytmu 7, gdzie $\mathrm{K} \leftarrow \mathrm{S} >> (3+j)$. Taka postać tej funkcji gwarantuje, że jej wartości będą względnie pierwsze z M, co z kolei ogranicza grupowanie

Algorytm 7 Obliczanie wartości funkcji h_2

```
i ← 1 + K mod (M − 1)
if i mod 2 = 0 then
  i ← i + 1
end if
return i
```

danych w elementach tablicy [83]. Dzięki temu odnalezienie elementu pustego lub zawierającego określone dane wymaga mniejszej liczby iteracji pętli **repeat**... **until** z Algorytmu 7, co daje lepszą efektywność tej metody.

Algorytm 6 pozwala zlokalizować element tablicy głównej, który zawiera wskaźnik na bufor, którego dotyczy żądanie klienta. Ten bufor podzielony jest na osiem sektorów, przy czym numer pierwszego z nich jest podzielny przez osiem. Otrzymane przez serwer SDDSfL żądanie klienta może dotyczyć któregoś z dalszych sektorów, ale dzięki przedstawionej w Podrozdziale 5.2 fragmentacji żądań gwarantowanym jest, że nie będzie przekraczało rozmiaru bufora. Położenie pierwszego sektora, którego dotyczy żądanie względem adresu początkowego bufora jest ustalane poprzez pomnożenie wartości trzech najmłodszych bitów numeru tego sektora przez rozmiar pojedynczego sektora.

Jeśli żądanie dotyczy odczytu, a Algorytm 6 zwróci wskaźnik na nieużywany element, to nie musi to oznaczać błędu. Przykładowo, Warstwa Operacji Blokowych przy inicjowaniu działania urządzenia blokowego nakazuje odczyt kilku sektorów z początku i końca przestrzeni adresowej

[3]Jej wartość może być modyfikowana przez użytkownika SDDSfL.

tego urządzenia. Spodziewa się, że sektory, które tam się znajdują będą zawierać zera. Serwer SDDSfL zachowuje się zgodnie z tymi oczekiwaniami i jeśli znajdzie nieużywany bufor podczas realizacji żądania odczytu, to wysyła w odpowiedzi sektory zawierające zera.

Podczas realizacji żądania zapisu serwer SDDSfL musi przypisać wskaźniki na bufor, w którym umieścił dane do tablicy podziału lub tablicy następnej. Jeżeli wartość klucza bloku, który umieszczony jest w buforze po przesunięciu bitowym o j pozycji w prawo jest parzysta, to ten blok po podziale pozostanie w bieżącym wiaderku i wskaźnik na bufor go zawierający powinien być zapisany w tablicy następnej. W przeciwnym przypadku wskaźnik powinien być zapisany w tablicy podziału. Element tablicy, który będzie zawierał ten wskaźnik w obu przypadkach jest wyznaczany za pomocą Algorytmu 6, ale w Funkcji 5.1 przyjmuje się poziom wiaderka o jeden większy. Wykonanie opisanej czynności pozwala uniknąć rozdzielania bloków danych za pomocą funkcji h_{j+1} (Podrozdział 4.1) dopiero na etapie wykonywania operacji podziału wiaderka. Aby wykazać, że obie metody są równoważne należy zauważyć, że funkcja h_j generuje z klucza bloku liczbę j-bitową, a funkcja h_{j+1} liczbę $j+1$ bitową, czyli o tym, czy blok o danym kluczu pozostanie po podziale w wiaderku, czy nie, decyduje wartość $j+1$ bitu jego klucza. Stąd wniosek, że badanie parzystości wartości klucza przesuniętej o j pozycji w prawo jest operacją równoważną.

Serwer SDDSfL utrzymuje informacje o poziomie załadowania wiaderka, który określany jest przez liczbę zajętych elementów tablicy głównej i zwiększany przy każdym zapisie nowego bloku danych. Jeśli przekroczy on pewną wartość progową, to serwer zgłasza do koordynatora podziałów przepełnienie wiaderka. Wynika stąd, że komunikat o przepełnieniu wiaderka wysyłany jest zanim faktycznie to nastąpi. Takie działanie podyktowane jest tym, że w plikach SDDS mogą znajdować się wiaderka o niższym poziomie niż to, które uległo przepełnieniu. Koordynator podziałów nakaże więc podział najpierw tym wiaderkom. Przeprowadzenie tych wszystkich operacji podziału mogłoby spowodować przestój w działaniu wiaderka, które się przepełniło. Podobny przypadek mógłby wystąpić, jeśli SC otrzymałby komunikat o przepełnieniu wiaderka, kiedy już byłaby realizowana operacja podziału któregoś z pozostałych wiaderek. Jest więc korzystnym utrzymywanie pewnego zapasu miejsca w wiaderku, aby uniknąć takich sytuacji. Powstaje pytanie, jaki próg załadowania przyjąć. Ponieważ efektywność adresowania otwartego spada, kiedy liczba elementów zajętych stanowi 75% liczby wszystkich elementów tablicy, to taki próg właśnie przyjęto [33]. Po jego osiągnięciu wysyłany jest pierwszy komunikat o przepełnieniu wiaderka, który SC musi potwierdzić ze względu na zawodność komunikacji opartej na protokół UDP/IP. To potwierdzenie nie jest równoznaczne z nakazem podziału. Jeśli serwer nie odbierze komunikatu „you split" po wysłaniu „collision", to ponownie wysyła ten ostatni po zrealizowaniu pewnej liczby żądań klienta. Działanie to powtarza do momentu, aż dostanie od SC nakaz podziału wiaderka.

Realizacją operacji podziału zajmuje się osobny wątek serwera SDDSfL. W momencie odebrania od SC komunikatu „you split" nawiązuje połączenie z serwerem na innym węźle multikomputera, który będzie zarządzał nowym wiaderkiem. Następnie wysyła do niego poziom wiaderka $(j+1)$, blokuje pozostałym wątkom dostęp do danych w swoim wiaderku za pomocą muteksu i przesyła do nowego wiaderka zawartość tych wszystkich buforów, które wskazywane są przez tablicę podziału, wraz z identyfikatorem bloku skojarzonego z danym buforem oraz z indeksem tablicy głównej, pod którym był zapisany wskaźnik na ten bufor. Wątek po stronie odbiorcy zapisuje te dane w tablicach i buforach swojego wiaderka, jednocześnie licząc poziom załadowania. Po zakończeniu transmisji serwer zarządzający nowym wiaderkiem jest gotów do przyjmowania zleceń od klienta. Wątek po stronie nadawcy po wysłaniu danych do nowego wiaderka musi zmienić konfigurację tablic. Najpierw przepisuje wskaźniki buforów wskazywanych przez tablicę podziału do nieużywanych elementów tablicy następnej i oznacza wskazywane przez nie bufory jako niezajęte. W kolejnym kroku zamienia wskaźniki na tablicę główną i tablicę następną, co oznacza, że zamieniają się one rolami. Potem zeruje zawartość tablicy podziału i (nowej) ta-

blicy następnej, sprawdza, które elementy (nowej) tablicy głównej wskazują na zajęte bufory i zapisuje ich wskaźniki odpowiednio do tablicy podziału lub tablicy następnej. Po wykonaniu tych czynności oblicza nowy poziom załadowania wiaderka (przyjmuje, że jest to połowa poprzedniego poziomu), powiadamia SC o zakończeniu operacji podziału i zwalnia mutex. Serwer przechodzi do obsługi żądań klienta.

Podobnie jak klient, serwer SDDSfL korzysta ze statycznej tablicy przydziałów fizycznych (T). W jego przypadku ta tablica jest odczytywana podczas inicjacji z pliku tekstowego i oprócz adresów fizycznych wiaderek zawiera adres koordynatora podziałów oraz adres logiczny jego wiaderka.

Serwer SDDSfL może zostać wyłączony na skutek wystąpienia wyjątku (ang. *exception*) lub działań użytkownika (np. naciśnięcia klawiszy Ctrl i C). O takich zdarzeniach procesy w systemie Linux powiadamiane są za pomocą krótkich komunikatów nazywanych sygnałami [89, 91, 95]. Sygnały są programowymi odpowiednikami przerwań sprzętowych [89]. W prototypie SDDSfL przejęto obsługę większości sygnałów jakie oprogramowanie serwera może otrzymać. Procedura implementująca ich nową obsługę powoduje zapisanie zawartości wiaderka do pliku, dzięki czemu może ono zostać odtworzone po przywróceniu działania serwera.

Obsługa wielu klientów przez serwer SDDSfL zrealizowana jest za pomocą podziału wiaderka na segmenty. Każdy klient otrzymuje swój segment, do którego może zapisywać lub z którego może odczytywać dane. Segmenty te mają równe wielkości (z dokładnością do pojedynczego elementu). Dla każdego z nich osobno jest liczony poziom załadowania. Do adresowania elementów segmentów używany jest Algorytm 6, ale rozmiar tablicy (M) w funkcjach h_1 i h_2 zastąpiony jest rozmiarem segmentu. W przypadku podziału jednego z segmentów dzielone są również pozostałe segmenty należące do tego samego wiaderka. W przyjętym rozwiązaniu klienci nie współdzielą bezpośrednio danych. Jeśli jest to konieczne, to taki mechanizm może zostać zrealizowany na poziomie aplikacji korzystających z wirtualnego urządzenia blokowego tworzonego przez moduł klienta SDDSfL.

Implementacja współdzielenia danych przez klientów na poziomie sterowników SDDSfL wiąże się z wprowadzeniem określonej polityki spójności informacji, a to może wpłynąć niekorzystnie na wydajność SDDSfL oraz nie zawsze odpowiadać potrzebom aplikacji, które korzystają z tego rozwiązania. Dodatkowo wszyscy klienci musieliby posiadać ten sam system plików. Aby utrzymać jego spójny obraz konieczne byłoby sterowanie z poziomu sterownika urządzenia programowymi pamięciami podręcznymi zawierającymi struktury danych systemu plików, takie jak np. wpisy katalogowe (ang. *d-entry*) [30]. Realizacja takiego sterowania nie jest prosta dla obecnej struktury jądra Linuksa.

Sugestie odnośnie możliwości realizacji współdzielenia danych na poziomie sterowników SDDSfL zawarte są także w Rozdziale 7.

5.4. Implementacja koordynatora podziałów

Działanie koordynatora podziałów w prototypowej implementacji SDDSfL odbiega w niewielkim stopniu od tego, które zostało opisane automatem skończonym przedstawionym na Rysunku 4.5. Zmiany w działaniu wymuszone są tym, że komunikaty przesyłane za pomocą protokołu UDP/IP mogą ulec zagubieniu. W związku z tym SC zawsze potwierdza komunikat „collision" otrzymany od serwera SDDSfL. Po otrzymaniu takiego komunikatu możliwe są dwa scenariusze działania koordynatora:

1. W chwili otrzymania komunikatu o przepełnieniu wiaderka żaden serwer nie przeprowadza operacji podziału. W takim przypadku SC potwierdza nadawcy odebranie komunikatu „collision", wyznacza wiaderko do podziału zgodnie z Algorytmem 5 i wysyła serwero-

wi, który zarządza tym wiaderkiem komunikat „you split”. Po wykonaniu tych czynności przechodzi w stan oczekiwania na potwierdzenie zakończenia podziału lub na kolejne komunikaty o przepełnieniu.

2. W chwili otrzymania komunikatu „collision” trwa wykonanie operacji podziału przez inny serwer. W takim przypadku SC jedynie potwierdza odebranie komunikatu i natychmiast przechodzi do oczekiwania na kolejne komunikaty o przepełnieniu lub na potwierdzenie zakończenia podziału.

Tak jak opisano to w Podrozdziale 5.3 serwer SDDSfL będzie ponawiał komunikaty o przepełnieniu do momentu, aż koordynator podziałów wyśle mu nakaz przeprowadzenia podziału.

5.5. Realizacja protokołów

Protokołami warstwy transportu używanymi w prototypie SDDSfL są TCP/IP i UDP/IP, przy czym większą rolę w komunikacji między poszczególnymi elementami prototypu odgrywa ten drugi. Dla każdego z typów połączeń zdefiniowano osobny protokół warstwy aplikacji oraz przydzielono osobne numery portów. Implementacje protokołów są przedstawione w tym podrozdziale.

5.5.1. Protokół połączenia klient-serwer

Każdy komunikat przesyłany między klientem i serwerem, niezależnie od kierunku komunikacji, posiada nagłówek, który zdefiniowany jest strukturą o nazwie `udp_header` (Listing 5.1).

Listing 5.1: Struktura udp_header

```
struct udp_header {
        uint8_t type;
        uint8_t flags, client_nr;
        char original_address[16];
        union
        {
                uint64_t begin;
                uint32_t iam[2];
        };
        uint16_t number;
};
```

Użyte w definicji struktury typy `uint8_t`, `uint16_t`, `uint32_t` oraz `uint64_t` są zgodnymi ze standardem ISO C99 typami zmiennych o ustalonej, niezależnej od platformy sprzętowej i systemowej liczbie bitów, dla liczb całkowitych dodatnich (naturalnych) [39]. Pole `type` może przyjmować jedną z pięciu wartości określających zapis, odczyt, potwierdzenie zapisu, potwierdzenie odczytu oraz komunikat IAM. Dwa pierwsze typy komunikatów wysyła klient, pozostałe serwer SDDSfL. Komunikat zapisu oraz komunikat potwierdzenia odczytu oprócz nagłówka zawierają dane, których maksymalny rozmiar wynosi 4 KiB. Pole `flags` jest używane przez serwer do oznaczenia pakietu, który został nieprawidłowo zaadresowany i jest przesyłany do innego serwera. To pole wykorzystywane jest również w trybie diagnostyki (ang. *debug*) przez klienta do oznaczania pakietów retransmitowanych. Pole `client_nr` zawiera numer identyfikacyjny klienta, który wysłał pakiet, a pole `original_address` jego adres IP w dziesiętnej notacji kropkowej (ang. *dot-decimal notation*). Pole `number` zawiera rozmiar danych, których dotyczy żądanie zapisu lub odczytu wyrażony w liczbie bajtów. Pole `begin` zawiera numer pierwszego sektora

określającego dane, których dotyczy żądanie. Jeśli pakiet jest komunikatem IAM to to pole zastępowane jest dwuelementową tablicą, która zawiera numer i poziom wiaderka nadzorowanego przez serwer, który otrzymał nieprawidłowo zaadresowane żądanie.

5.5.2. Protokół połączenia serwer-serwer

Ten rodzaj połączenia oparty jest na protokole TCP/IP i wykorzystywany podczas operacji podziału wiaderka. Komunikaty przesyłane za jego pomocą zdefiniowane są strukturą o nazwie `split_data` (Listing 5.2).

Listing 5.2: Struktura split_data

```
struct split_data {
        unsigned char end;
        uint64_t sector, index;
        uint8_t client;
        unsigned char data[4096];
};
```

Pole `end` jest znacznikiem określającym, czy bieżący komunikat jest ostatnim, czy będą po nim nadsyłane kolejne. Pole `sector` zawiera klucz, czyli numer pierwszego sektora bloku przesunięty o trzy bity w prawo, a pole `index` zgodnie z nazwą określa położenie elementu tablicy podziału, który wskazywał na przesyłane dane. Taki sam indeks będzie miał element tablicy głównej, który w nowym wiaderku będzie wskazywał te dane. Pole `client` zawiera numer klienta, który jest właścicielem danych. Ostatnie pole jest tablicą zawierającą dane pobrane z bufora.

Zanim zostanie rozpoczęte przesyłanie komunikatów określonych taką strukturą, to do nowego wiaderka przesyłany jest jego poziom, zapisany w zmiennej typu `uint32_t`.

5.5.3. Protokół połączenia serwer-koordynator podziałów

Komunikacja między tymi dwoma elementami SDDSfL polega na wymianie prostych znaczników. Wyjątkiem od tej reguły jest komunikat o przepełnieniu wiaderka, który może zawierać dodatkowe informacje. Znaczniki używane w tym połączeniu zdefiniowane są za pomocą typu wyliczeniowego o nazwie `msg_type` (Listing 5.3).

Listing 5.3: Typ msg_type

```
enum msg_type {
        collision = 0,
        yousplit = 1,
        split_ack = 2,
        collision_ack = 3
};
```

Znacznik `collision` oznacza komunikat o przepełnieniu wiaderka. Jak wspomniano wcześniej pakiet ten może zawierać dodatkowe informacje, które używane są w podziałach kontrolowanych. Ten typ podziałów nie został jednak oprogramowany w prototypie SDDSfL. Pakiet `collision` wysyła serwer do koordynatora podziałów, który potwierdza jego odbiór wysłaniem znacznika `collision_ack`. Serwer zarządzający wiaderkiem, które powinno zostać podzielone otrzymuje od SC znacznik `you_split`, na który odpowiada znacznikiem `split_ack` po wykonaniu operacji podziału wiaderka.

5.6. Odporność na błędy SDDSfL

Odporność na błędy (ang. *fault-tolerance*) Skalowalnych, Rozproszonych Struktur Danych jest przedmiotem wielu badań. Przegląd architektur SDDS związanych z tym zagadnieniem znajduje się w Dodatku A. Przy opracowywaniu prototypu SDDDSfL skupiono się na zapewnieniu, że klient w sytuacji awaryjnej będzie zachowywał się podobnie jak większość sterowników typowych urządzeń blokowych. Prace nad tolerowaniem błędów przez ten prototyp były więc bardziej ukierunkowane na obsługę sytuacji wyjątkowych niż zapewnienie nieprzerwanej dostępności usługi.

Tabela 5.1 zawiera analizę przyczynowo-skutkową (ang. *cause-effect*) możliwych błędów klienta SDDSfL. Z tej tabeli wynika, że najpoważniejsze skutki mają cztery kategorie błędów: utrata komunikatu, brak odpowiedzi od serwera, błąd klienta w obliczaniu adresu serwera oraz podanie przez użytkownika złych adresów IP serwerów. Jeśli wystąpi którykolwiek z wymienionych błędów, to klient SDDSfL pozostanie w stanie oczekiwania na odbiór pakietu, co uniemożliwi mu wykonywanie innych zadań, włącznie z reagowaniem na polecenia wydawane przez użytkownika uprzywilejowanego. Jedynym sposobem usunięcia modułu klienta z jądra systemu w takiej sytuacji jest restart węzła multikomputera, na którym dany moduł działał. Błąd związany z utratą komunikatu można usunąć poprzez retransmisję. W prototypowej implementacji SDDSfL wykorzystano kolejkę prac (ang. *work queue*) do wykonania tej czynności [30, 39]. Pozwala ona uaktywnić określoną funkcję po upływie zadanego czasu. Jądro systemu Linux ofe-

Przyczyna	**Skutek**
Użytkownik podaje złe adresy IP węzłów-serwerów.	Klient wysyła dane do niewłaściwych komputerów. Możliwe nieskończone oczekiwanie na odpowiedź. Moduł nie może być usunięty z jądra systemu (jest używany).
Użytkownik podaje niekompletne dane serwerów.	Plik urządzenia blokowego nie jest tworzony. Działanie systemu nie jest zagrożone. Moduł można usunąć z systemu.
Sterownik odbiera niewłaściwą odpowiedź na żądanie.	Jeśli błąd dotyczy metadanych żądania, to komunikat jest odrzucany. Jeśli błąd dotyczy właściwych danych, to jest propagowany do aplikacji, która odpowiedzialna jest za obsługę wyjątku.
Brak odpowiedzi ze strony serwera (głuchy serwer (ang. *deaf server*)).	Klient czeka w nieskończoność na odpowiedź, mimo ponawianych zapytań. Awaryjne zakończenie działania aplikacji jest trudne (czeka na realizację wywołania systemowego). Nie można usunąć modułu z jądra (jest używany).
Błąd klienta w obliczaniu adresu logicznego serwera.	Jeśli adres jest większy niż rozmiar tablicy przydziałów fizycznych, to skutek jest taki sami jak w przypadku głuchego serwera lub złego adresu fizycznego serwera.
Utrata komunikatu.	Klient czeka nieskończenie długo na odpowiedź serwera. Moduł nie może być usunięty z systemu (jest używany).

Tabela 5.1: Analiza przyczynowo-skutkowa potencjalnych błędów klienta SDDSfL

ruje domyślny mechanizm kolejki prac obsługiwany przez wątek o nazwie `events`, ale stwarza również możliwość utworzenia innej instancji takiego mechanizmu. Ponieważ domyślna kolejka prac może być obciążona przez prace zlecone przez inne moduły, klient SDDSfL korzysta z drugiego rozwiązania i tworzy swoją prywatną kolejkę. Funkcja realizowana w ramach prac umieszczanych w tej kolejce ma za zadanie powtórzyć transmisję ostatnio wysłanego pakietu. Ta czynność nie może być wykonana bezpośrednio przez wątek transmitujący, gdyż jest on zablokowany w oczekiwaniu na odpowiedź. Kolejka prac jest w przypadku klienta SDDSfL realizacją mechanizmu watchdog przedstawionego na Rysunku 4.2 z Podrozdziału 4.4. Komunikat, jeśli zachodzi taka konieczność, może być retransmitowany wielokrotnie. Czas każdej kolejnej retransmisji dobierany jest zgodnie z regułą wykładniczego wycofywania binarnego [2, 91]. Jeśli wartość tego czasu przekroczy przyjęty próg (około pół minuty), to błąd uznawany jest za permanentny. Najczęściej jest on wtedy związany z jedną z trzech pozostałych przyczyn, czyli błędnym adresem fizycznym lub logicznym serwera bądź jego awarią. W takim przypadku przerywane jest działanie wątku transmitującego oraz aplikacji użytkownika oczekujących na zakończenie operacji wejścia-wyjścia związanych ze sterownikiem SDDSfL, a Warstwa Operacji Blokowych, powiadamiana jest o wystąpieniu błędu I/O. Każda kolejna próba skorzystania przez aplikację użytkownika z SDDSfL spowoduje ponowne zasygnalizowanie tego błędu. Moduł klienta może zostać usunięty z jądra systemu. Sytuacja wyjątkowa nie ma wpływu na działanie pozostałych procesów użytkownika ani innych podsystemów jądra Linuksa.

5.7. Podsumowanie

Rozdział opisuje prototypową implementację SDDSfL. Oprogramowanie klienckie zostało zrealizowane w postaci modułu jądra będącego sterownikiem urządzenia blokowego. Tworzy on osobny wątek, który obsługuje transmisję danych między klientem, a serwerem SDDSfL. Retransmisją zagubionych pakietów zajmuje się dedykowany mechanizm kolejek prac, który jest realizacją mechanizmu watchdog. Informacje o trwałych błędach sterownika, spowodowanych awarią serwerów są przekazywane aplikacjom użytkowym za pomocą sygnałów. Komunikacja z przestrzenią użytkownika jest również konieczna w przypadku inicjacji tablicy przydziałów fizycznych klienta. Do jej realizacji wykorzystano system plików `sysfs`.

Serwery SDDSfL są programami uruchamianymi na osobnych węzłach multikomputera jako osobne procesy użytkownika. Wiaderka, którymi one zarządzają zrealizowano poprzez użycie tablic wskaźników, dynamicznej alokacji pamięci i blokowania wymiany stron. Podstawową jednostką danych przechowywaną przez wiaderko jest blok danych składający się z ośmiu sektorów o wielkości 512 bajtów. Do adresowania bloków wewnątrz wiaderka użyto algorytm adresowania otwartego z podwójnym haszowaniem. Kod serwera podzielono na trzy wątki. Główny wątek zajmuje się obsługą żądań klienta pozostałe dwa odpowiedzialne są za wykonanie operacji podziału wiaderka.

Koordynator podziałów również działa jako proces użytkownika. Jego tablica przydziałów fizycznych jest inicjowana wartościami odczytywanymi z pliku. To samo rozwiązanie zastosowano w serwerze.

Do komunikacji użyto tych samych protokołów warstwy transportu, co w przypadku oryginalnych implementacji SDDS.

Głównym źródłem trudności związanych z implementacją SDDSfL stanowiło osadzenie klienta na poziomie jądra systemu operacyjnego. Błąd w jego kodzie lub nawet w oprogramowaniu serwera SDDSfL zazwyczaj prowadził do destabilizacji lub nawet załamania całego systemu, co utrudniało lokalizację usterki. Na poziomie jądra systemu nie są także dostępne biblioteki podprogramów, które ma do dyspozycji programista aplikacji użytkowych, choć istnieją odpowiedniki niektórych z nich. Inne utrudnienie stanowiła rozproszona konstrukcja oprogramowania

SDDSfL. W tego typu rozwiązaniach często występują trudne do wykrycia i odtworzenia błędy związane z komunikacją.

6. Ocena eksperymentalna

Rozdział 6 zawiera wyniki testów, jakim została poddana prototypowa implementacja SDDSfL. Pierwsza część opisuje metodologię testów, druga przedstawia wyniki wraz z ich analizą, a ostatnia stanowi krótkie podsumowanie uzyskanych rezultatów.

6.1. Metodologia

Koncepcja Skalowalnych, Rozproszonych Struktur Danych powstała w wyniku zapotrzebowania na przechowywanie dużych ilości danych w sposób skalowalny i przy zachowaniu możliwości szybkiego dostępu do informacji [27]. Prototyp SDDSfL został poddany testom, aby ocenić w jakim zakresie posiada on takie cechy. Ocena wydajności została wykonana w odniesieniu do efektywności tradycyjnych rozwiązań służących do magazynowania danych, takich jak lokalne dyski twarde i rozproszone systemy plików. Do testów użyto dysków wyposażonych w magistrale SATA, SATA 2, PATA (UDMA/133) i SCSI Ultra-320 [96–98] oraz następujących rozproszonych systemów plików: NFS, GlusterFS i LUSTRE [5,12,21,99]. Na dyskach twardych, podobnie jak na urządzeniu blokowym SDDSfL osadzony był lokalny system plików ext3 [100].

Wydajność każdego z wymienionych rozwiązań zależy od konfiguracji systemu komputerowego, który je stosuje oraz od aplikacji, które z nich korzystają. Prototyp SDDSfL przetestowano z następującym oprogramowaniem:

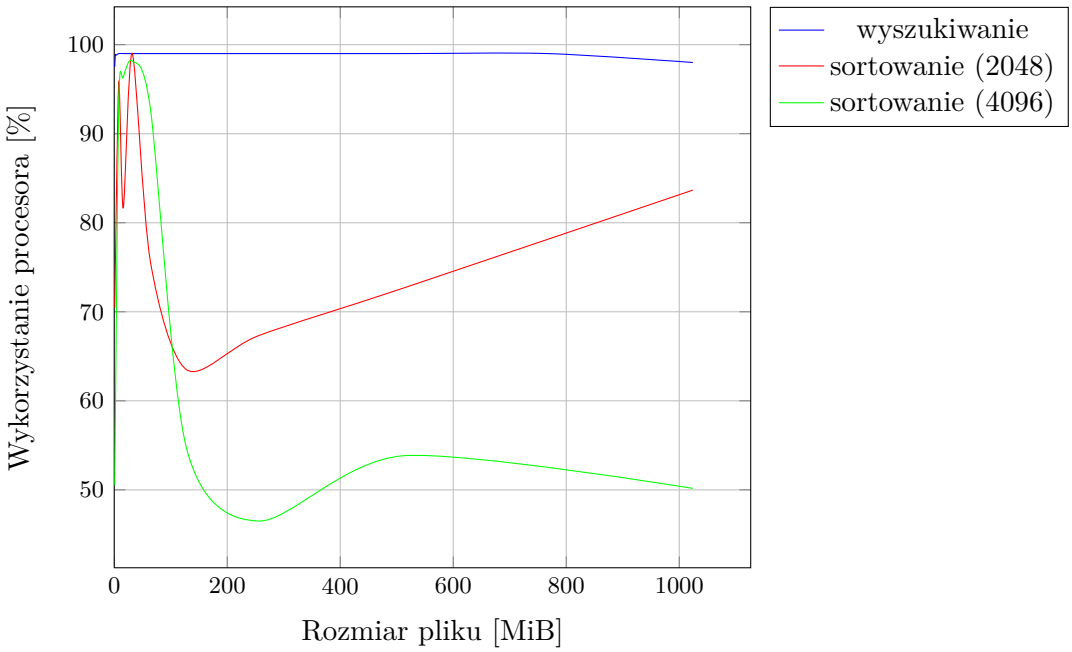
1. transakcyjne bazy danych,
2. procesy zorientowane na wejście-wyjście [2],
3. procesy zorientowane na obliczenia [2],
4. podsystem wymiany stron pamięci (ang. *swap*) [2–4].

W testach z pierwszej kategorii użyto systemu baz danych PostgreSQL [101] oraz programu testującego `pgbench` [102], który stosuje schemat testowania transakcji bazujący na normie TPC-B [103]. Badaną wielkością była liczba transakcji wykonanych w ciągu sekundy.

Jako program zorientowany na wejście-wyjście posłużyła aplikacja sortująca pliki rekordów w oparciu o algorytm QuickSort [33,83]. Podczas testu użyto dwóch zestawów plików. Pierwszy zawierał pliki o wielkościach od 1 MiB do 1 GiB zbudowane z rekordów o rozmiarze 2 KiB. W drugim zestawie pliki miały identyczne wielkości, ale rozmiar rekordu wynosił 4 KiB.

Zbiory o takim samym zakresie rozmiarów, lecz zawierające tekst zostały wykorzystane w teście należącym do trzeciej kategorii. Wykonano go przy pomocy aplikacji poszukującej wystąpień wzorców tekstowych o długościach od 2 do 1024 znaków, przy czym długości te były kolejnymi potęgami dwójki. Do wyszukiwania zastosowano algorytm Boyera-Moore'a [83].

W przypadku obu opisanych aplikacji testujących mierzoną wielkością był czas wykonania. Do jego pomiaru wykorzystano polecenie systemowe `time`. Ponieważ bada ono również stopień wykorzystania procesora, to za jego pomocą można potwierdzić charakterystykę działania



Rysunek 6.1: Wykorzystanie procesora w wyszukiwaniu wzorców i sortowaniu plików

obu aplikacji, którą przedstawiono wcześniej. Wykres 6.1 przedstawia wyrażone w procentach obciążenie procesora przez obydwie aplikacje w zależności od rozmiaru sortowanego lub przeszukiwanego pliku. Wielkość ta wyraża stopień obciążenia procesora w całkowitym czasie trwania danej operacji. Pomiar został dokonany dla procesora dwurdzeniowego. Na podstawie wykresu można wywnioskować, że sortowanie jest procesem zorientowanym na wejście-wyjście ze względu na mniejsze wykorzystanie procesora w porównaniu z wyszukiwaniem wzorca, które jest zorientowane na obliczenia. Rekordy w plikach wejściowych dla programu sortującego zostały wygenerowane losowo, co oznacza, że ich kolejność początkowa jest różna dla każdego z utworzonych plików. Od tej kolejności zależy intensywność korzystania z procesora w algorytmie QuickSort, dlatego na Wykresie 6.1 kształty linii odpowiadającym wynikom sortowania są różne.

W testach z czwartej kategorii użyto programu zapisującego zera w przydzielony mu obszarze pamięci. Wielkość tego obszaru jest porównywalna z rozmiarem RAM. Program zapisuje obszar stopniowo, za każdym razem zaczynając od jego początku i zwiększając wielkość zapisywanego fragmentu o 1 MiB. Wielkością mierzoną jest czas wykonania operacji zapisu. Pomiaru dokonuje program przy użyciu wywołania systemowego `gettimeofday()`.

Skalowalność SDDSfL została oceniona przy pomocy aplikacji sortującej pliki. Operację tę przeprowadzano dla różnej liczby i pojemności serwerów SDDSfL oraz dla różnej liczby klientów SDDSfL.

Testy zostały wykonane na multikomputerach zbudowanych w oparciu o typowe komputery klasy PC i sieci LAN oraz na systemie wielokomputerowym klasy MPP, który wyposażony jest zarówno w sieć lokalną Gigabit Ethernet, jak i InfiniBand.

Szczegółowe dane na temat konfiguracji środowiska testowego oraz badanych rozwiązaniach zostały umieszczone w opisie wyników w następnym podrozdziale.

6.2. Wyniki testów

Wydajność SDDSfL oraz rozproszonych systemów plików uwarunkowana jest w dużym stopniu szybkością transmisji lokalnych sieci, na których oparta jest budowa systemu wielokomputerowego. Tabela 6.1 zawiera zestawienie, sporządzone na podstawie [2, 10, 96–98], maksymalnych szybkości transmisji jakie mogą zaoferować sieci komputerowe oraz magistrale I/O użytych w testach dysków twardych. Zarówno w przypadku dysków twardych jaki i sieci komputerowych

Interfejs	**Szybkość przesyłania danych**
Gigabit Ethernet	125 MiB/s
PATA UDMA/133	133 MiB/s
SATA	150 MiB/s
SATA 2	300 MiB/s
SCSI Ultra-320	320 MiB/s
InfiniBand	1,25 GiB/s

Tabela 6.1: Maksymalna szybkości przesyłania danych dla sieci i magistrali

osiągnięcie takich wartości w praktyce może być trudne.

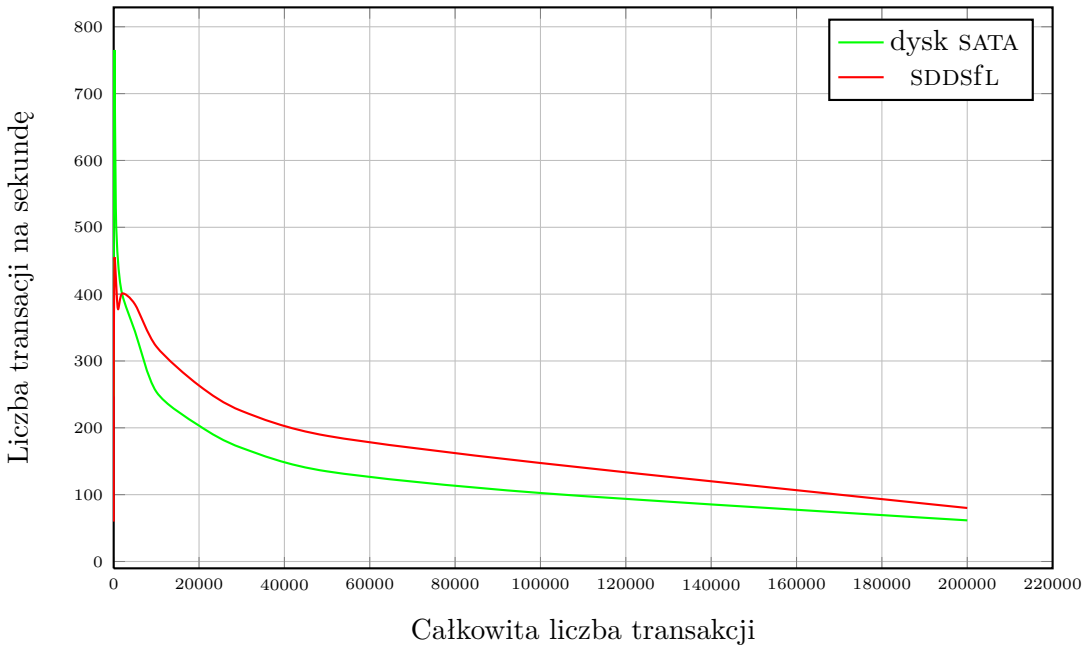
Źródła [2–4] wskazują trzy parametry, od których w największym stopniu zależy wydajność dysków twardych. Są to czas wyszukiwania, czas oczekiwania i czas transferu. Wszystkie te wielkości związane są z mechaniczną naturą tych urządzeń. Czas wyszukiwania (ang. *seek time*) jest czasem potrzebnym na umieszczenie głowicy dysku nad właściwą ścieżką. Czas oczekiwania nazywany również czasem opóźnienia rotacyjnego (ang. *rotational delay*) związany jest z czekaniem aż pod głowicą znajdzie się sektor, który ma być jako pierwszy odczytany ze ścieżki. Czas transferu (ang. *transfer time*) jest czasem potrzebnym na odczyt lub zapis danych w określonym sektorze. Spośród tych trzech składowych czasu dostępu największą jest czas wyszukiwania. Jego minimalizacją zajmuje się planista wejścia-wyjścia. Autorzy [104] przeprowadzili dokładną analizę czynników wpływających na wydajność współczesnych dysków twardych. Ich badania wykazały, że zastosowanie rozwiązań zmniejszających liczbę fizycznych operacji I/O, takich jak buforowanie zapisów, pamięć podręczna odczytów i odczyt z wyprzedzeniem (ang. *read ahead*) ma większe znaczenie dla skrócenia czasu dostępu do danych na dysku niż poprawa którejś z opisanych wcześniej składowych.

W przypadku lokalnych sieci komputerowych o faktycznej prędkości przesyłania danych decydują szczegóły związane z implementacją obsługi LAN na poziomie systemu operacyjnego, budowa protokołów komunikacyjnych i obciążenie sieci komunikacją. W Zestawieniu 6.1 sieć InfiniBand jest dziesięciokrotnie szybsza od Gigabit Ethernet. Nie każde oprogramowania potrafi jednak obsługiwać w sposób bezpośredni tę sieć. Prototyp SDDSfL korzysta z niej za pomocą stosu TCP/IP co powoduje straty w wydajności. Tabela 6.2 zawiera czasy przesyłu pakietów ICMP dla sieci Gigabit Ethernet i InfiniBand obsługiwanej za pomocą stosu protokołów TCP/IP.

Rozmiar	**Rodzaj sieci**	
	InfiniBand	Gigabit Ethernet
56 bajtów	0.047/0.059/0.082 [ms]	0.068/0.086/0.103 [ms]
4300 bajtów	0.093/0.112/0.151 [ms]	0.206/0.226/0.251 [ms]

Tabela 6.2: Czas przesyłu pakietu ICMP dla sieci InfiniBand i Gigabit Ethernet

Test wykonano na multikomputerze wyposażonym w obie sieci. Wyniki wykazują, że czasy przesyłania pakietów o względnie małych rozmiarach są w obu sieciach porównywalne, natomiast



Rysunek 6.2: Wydajność transakcji dla bazy danych PostgreSQL

pakiety o rozmiarze zbliżonym do maksymalnej wielkości pakietów stosowanych przez SDDSfL przesyłane są w sieci InfiniBand w czasie około dwukrotnie krótszym niż w Gigabit Ethernet.

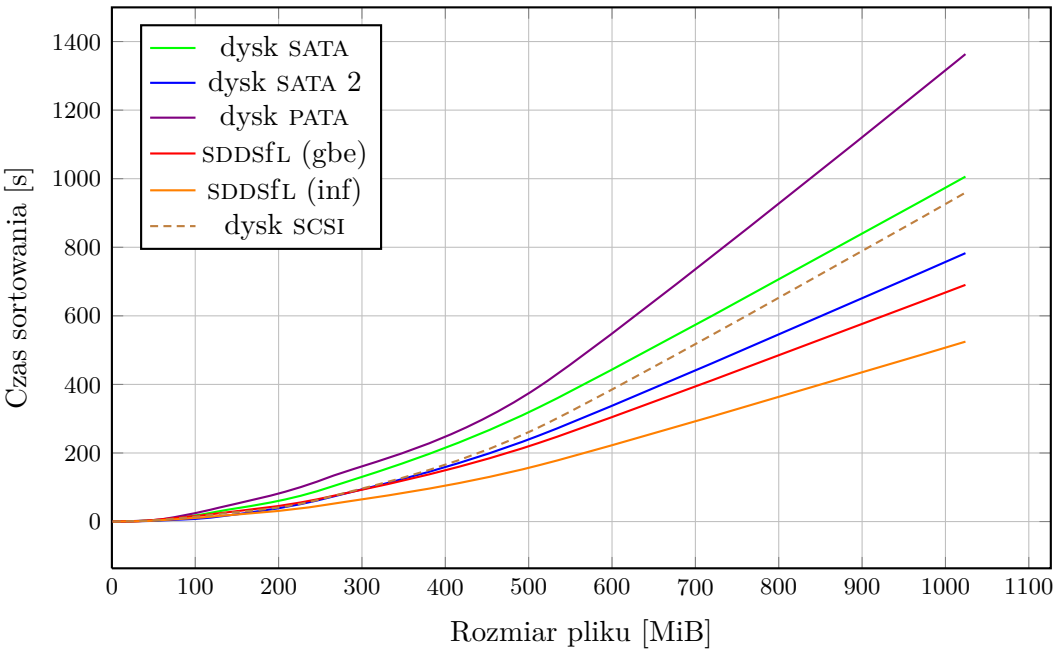
Aplikacje, które posłużyły do testów SDDSfL zostały dobrane tak, aby uwzględnić charakterystyki dysków twardych i sieci komputerowych, które zostały podane w dwóch poprzednich akapitach. Jest wśród nich zarówno oprogramowanie, które wymaga intensywnego przesyłania informacji i swobodnego dostępu do danych, jak również i to, które wykonuje sporadycznie operacje wejścia-wyjścia i ogranicza się jedynie do sekwencyjnego dostępu do danych.

Systemy transakcyjnych baz danych należą do programów, które wymagają swobodnego dostępu do danych na nośniku i inicjują wiele operacji I/O, które dotyczą zapisu lub odczytu małych porcji informacji. Wykres 6.2 przedstawia liczbę transakcji na sekundę, które wykonuje baza PostgreSQL używając jako nośnika danych dysku twardego lub prototypu SDDSfL. Test przeprowadzono na multikomputerze zbudowanym na bazie sieci Gigabit Ethernet z komputerów klasy PC. Każdy z nich był wyposażony w zintegrowaną kartę sieciową. Dysk twardy użyty w testach to SeaGate Barracuda o pojemności 80 GB [1], prędkości kątowej 7200 rpm i wyposażony w magistralę SATA. Wyniki testu wskazują na większą wydajność bazy, gdy współpracuje z prototypem SDDSfL (większa liczba transakcji na sekundę).

Algorytm QuickSort, podobnie jak transakcyjne bazy danych potrzebuje swobodnego dostępu do danych. Jednocześnie jest on efektywny także w systemach, które stosują klasyczną pamięć wirtualną [83]. Wskazuje to na dobrą lokalność odwołań do danych jakie wykonuje ten algorytm. Możliwe jest zatem wykorzystanie go do sortowania plików na nowoczesnych dyskach twardych.

Wykres 6.3 został sporządzony na podstawie danych uzyskanych w teście sortowania plików składających się z rekordów o wielkości 2 KiB. W badaniu porównano wydajności prototypu SDDSfL uruchomionego na węźle multikomputera typu MPP, który wyposażono w 2 GiB RAM, dysku twardego SeaGate Barracuda, SATA 2, 160 GB, 7200 rpm zainstalowanego w tym samym węźle, dysków SeaGate Barracuda SATA, 80 GB i SeaGate Cheetah, SCSI Ultra-320, 73 GB,

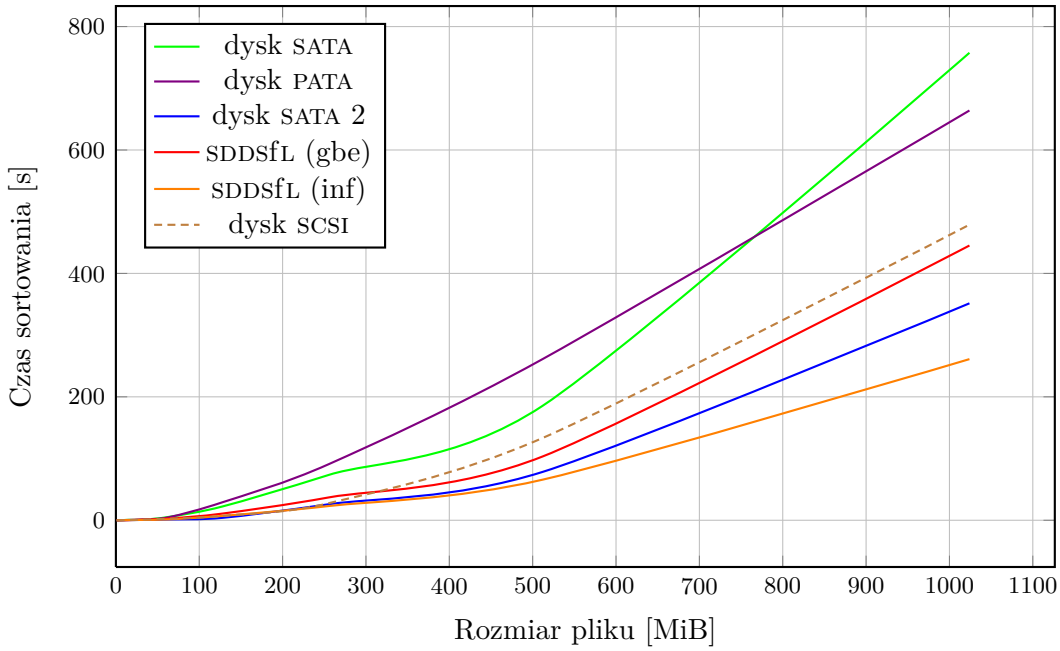
[1] Przedrostki w oznaczeniach pojemności dysków należą do układu SI.



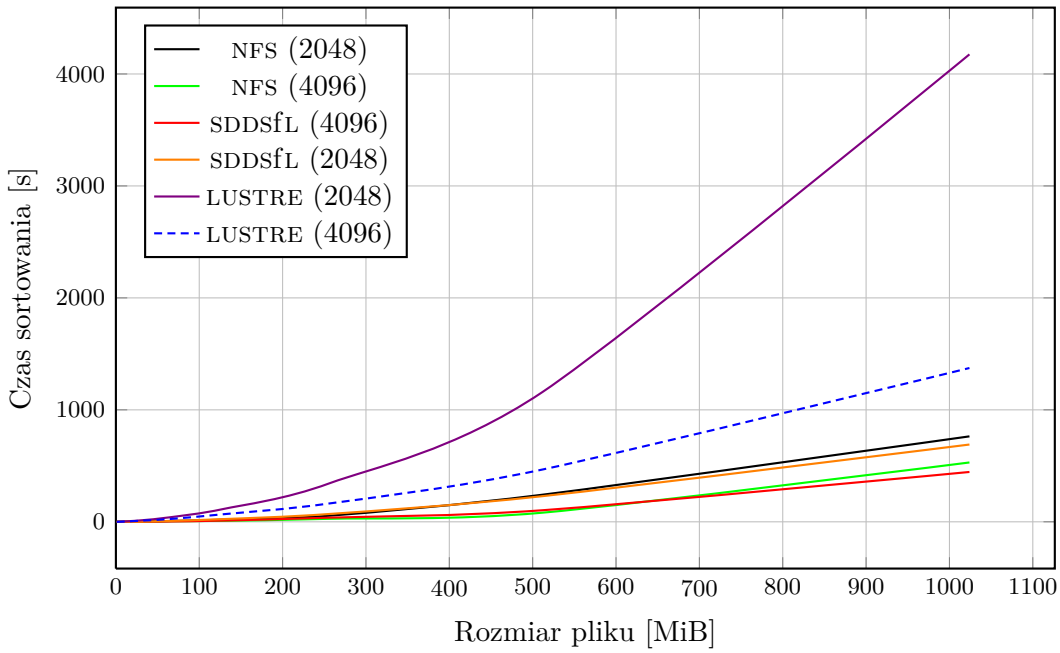
Rysunek 6.3: Czas sortowania plików o rozmiarze rekordów 2 KiB

15000 rpm zainstalowanych w komputerze klasy PC wyposażonym w 1,5 GiB RAM oraz dysku Maxtor, PATA UDMA/133, 160 GB, 7200 rpm zainstalowanego w komputerze klasy PC z 2 GiB pamięci operacyjnej. Test SDDSfL został zrealizowany zarówno dla sieci Gigabit Ethernet (gbe), jak i dla InfiniBand (inf). Analiza wykresu pozwala stwierdzić, że prototyp SDDSfL jest szybszy od dysków twardych (sortowanie plików trwa krócej), lecz różnica między jego wynikami, a wynikami uzyskiwanymi przez nowoczesne dyski przeznaczone dla rozwiązań serwerowych (dysk z łączem SATA 2) jest niewielka, szczególnie, gdy SDDSfL korzysta z sieci Gigabit Ethernet. Jeżeli rozmiar rekordu zostanie podwojony (4 KiB), to dysk SATA 2 okazuje się być szybszy w tym teście od SDDSfL uruchomionego na sieci Gigabit Ethernet, co ilustruje Wykres 6.4. Lokalność odwołań do danych, która cechuje algorytm QuickSort działa w tym teście na korzyść dysków twardych. Choć Warstwa Operacji Blokowych zapewnia buforowanie zapisów dla wszystkich urządzeń blokowych, to napędy dysków mogą wykorzystać dodatkowe bufory sprzętowe, w które są wyposażone, a których SDDSfL jest pozbawiony.

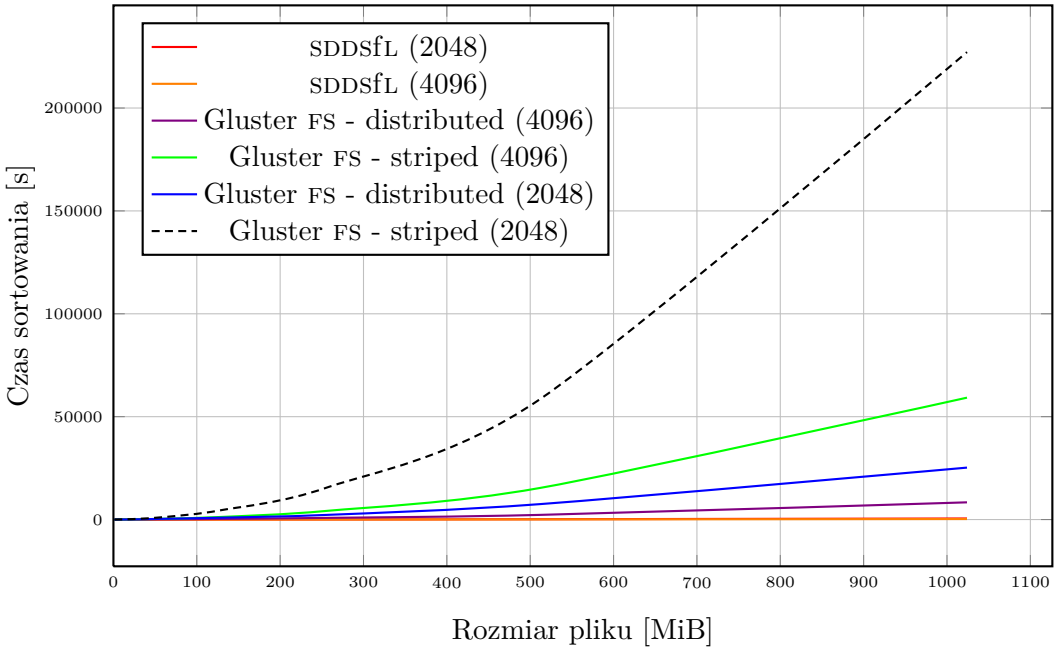
Wykres 6.5 sporządzono na podstawie testów sortowania plików przeprowadzonych dla prototypu SDDSfL oraz dwóch rozproszonych systemów plików, NFS i LUSTRE. Wszystkie testy zostały przeprowadzone przy pomocy sieci Gigabit Ethernet na systemie multikomputerowym klasy MPP. Na wykresie umieszczono wyniki dla sortowania plików zbudowanych z rekordów zarówno o wielkości 2 KiB, jak i 4 KiB. W systemie NFS dane są przechowywane w sposób scentralizowany na pojedynczym dysku twardym, ale może z nich korzystać wielu klientów [5,12]. System LUSTRE stosuje rozproszenie danych i metadanych plików między napędami dyskowymi zainstalowanymi w węzłach systemu wielokomputerowego. Porównanie wyników poszczególnych rozwiązań wypada na korzyść SDDSfL i NFS, przy czym dla plików o dużych rozmiarach (większych niż 512 MiB) przewagę uzyskuje SDDSfL. Warte odnotowania są również różnice między wynikami tych rozwiązań, a rezultatami LUSTRE. Budowa tego systemu plików jest zapewne bardziej dostosowana do aplikacji, które potrzebują sekwencyjnego dostępu do danych. W przypadku programów wymagających swobodnego dostępu konieczność wykonywania osobnych odwołań do metadanych pliku i bloków danych rozlokowanych na osobnych dyskach węzłów multikom-



Rysunek 6.4: Czas sortowania plików o rozmiarze rekordów 4 KiB



Rysunek 6.5: Czas sortowania plików (Gigabit Ethernet)



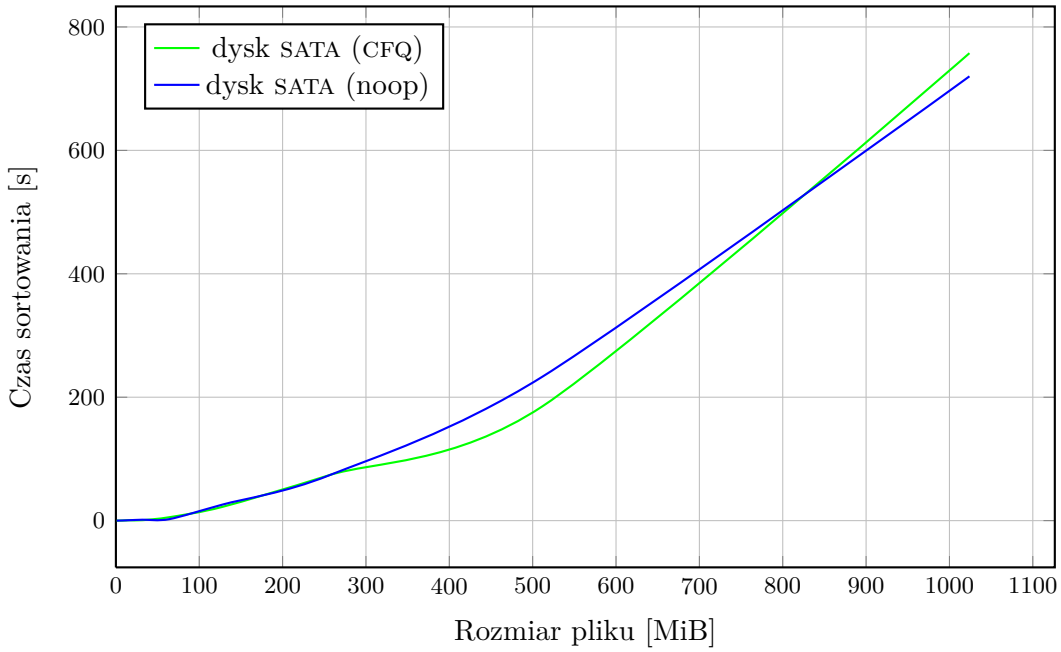
Rysunek 6.6: Czas sortowania plików (InfiniBand)

putera wpływa niekorzystnie na czas działania tych aplikacji. Ten wpływ jest jeszcze większy, jeśli operacje na pliku nie dają się przeprowadzić równolegle. Prototyp SDDSfL również dokonuje rozproszenia danych między komputerami wchodzącymi w skład systemu wielokomputerowego, ale urządzeniem przechowującym informacje jest w jego przypadku pamięć operacyjna, która ma czasy dostępu wielokrotnie mniejsze od dysku twardego. Stąd wynika znacząca przewaga tego rozwiązania nad LUSTRE w teście z sortowaniem plików.

Wykres 6.6 zawiera wyniki testu sortowania plików dla prototypu SDDSfL i rozproszonego systemu plików GlusterFS [21], przeprowadzonego przy pomocy sieci InfiniBand. Również w przypadku tego testu występuje duża różnica szybkości sortowania na korzyść SDDSfL. Wprawdzie oprogramowanie GlusterFS działa w przestrzeni użytkownika wykorzystując mechanizm FUSE [88], ale ten system plików korzysta z trybu RDMA sieci InfiniBand [10], co powinno korzystnie wpłynąć na jego wydajność. Ponownie przyczyną niskiej efektywności rozproszonego systemu plików jest prawdopodobnie jego architektura, która przystosowana jest dla aplikacji korzystających z sekwencyjnego dostępu do danych. W przypadku aplikacji wymagających swobodnego dostępu nie sprawdza się również zastosowanie paskowania (ang. *striping*). W rozproszonych systemach plików wykorzystanie tej techniki oznacza rozlokowanie fragmentów pliku na osobnych nośnikach dyskowych celem umożliwienia równoległego wykonywania na nich operacji I/O. Konfiguracja GlusterFS korzystająca z tej metody jest wolniejsza od konfiguracji, która rozmieszcza całe pliki na osobnych napędach. W przypadku tej ostatniej słaba wydajność jest prawdopodobnie spowodowana narzutami czasowymi, których przyczyną są mechanizmy systemu plików zajmujące się zarządzaniem danymi. Dla wersji GlusterFS z włączonym paskowaniem wykonano tylko jedną serię testów ze względu na długość ich trwania.

W Rozdziale 5 stwierdzono, że klient SDDSFfL, w przeciwieństwie do dysków twardych nie korzysta z planisty I/O. System Linux umożliwia wybór jednego z czterech planistów wejścia-wyjścia, w zależności od charakterystyki wykorzystania dysku przez wykonywane aplikacje [30, 105]. Przeprowadzono dodatkowe badania, aby ocenić, jaki wpływ ma wybór planisty I/O na wyniki dysku w teście polegającym na sortowaniu plików. Wykonano je na dysku z łączem SATA

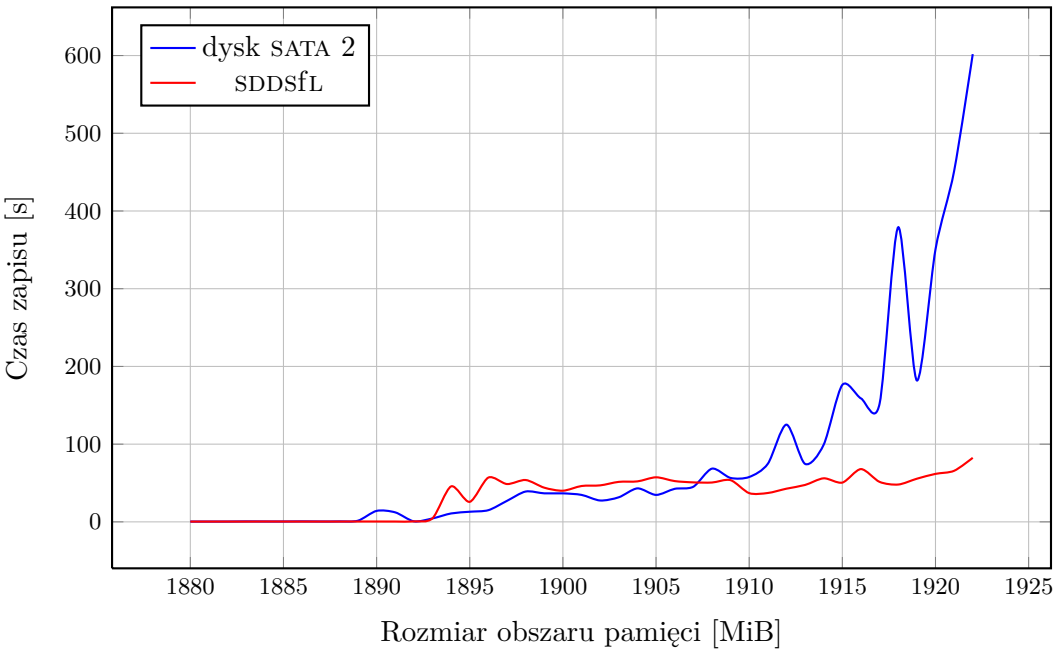
zainstalowanym w komputerze klasy PC wyposażonym w 1,5 GiB pamięci operacyjnej. W eksperymencie użyto dwóch planistów wejścia-wyjścia: planisty CFQ (ang. *Completely Fair Queuing*) i noop. Pierwszy z nich jest domyślnym planistą I/O w nowszych wersjach jądra Linuksa. Stosuje on szeregowanie żądań wejścia-wyjścia, które stanowi połączenie tradycyjnego algorytmu windy [30] z planowaniem w oparciu o wielopoziomowe kolejki [4]. Dodatkowo przypisuje on wyższy priorytet operacjom synchronicznym, na których zakończenie muszą czekać aplikacje, a niższy asynchronicznym, które tego nie wymagają [105–107]. Planista noop (ang. *no operation*) przeprowadza wyłącznie operację łączenia żądań, które dotyczą tych samych cylindrów na dysku [105, 107]. Po uzyskaniu rezultatów badań sporządzono Wykres 6.7. Na jego podstawie można stwierdzić, że dla sortowania plików o rozmiarze rekordu równym 4 KiB nie



Rysunek 6.7: Czas sortowania w zależności od planisty I/O

istnieje prosta zależność między użytym planistą I/O, a osiąganymi przez dysk twardy wynikami. Podobnego zachowania należy spodziewać się również w przypadku plików składających się z rekordów wielkości 2 KiB. Uzyskane wyniki zgodne są również z wnioskami zawartymi w [104]. Rodzaj zastosowanego planisty może mieć jednak znaczenie dla innych parametrów systemu, takich jak sprawiedliwość w dostępie do dysku dla aplikacji użytkownika, czy krótki czas odpowiedzi.

Jednym z zastosowań wirtualizacji rozproszonej pamięci multikomputera jest użycie dostępnej, rozproszonej RAM jako urządzenia wymiany. Prototyp SDDSfL został poddany testowi, którego celem było zbadanie jego wydajności w takim zastosowaniu w porównaniu z dyskiem twardym. Testy przeprowadzono na multikomputerze zbudowanym z komputerów klasy PC wyposażonych w 2 GiB pamięci operacyjnej i karty sieciowe Intel PRO/1000, które umożliwiają pracę w sieci Gigabit Ethernet. Jako napędu dyskowego użyto SeaGate Barracuda, SATA 2, 80 GiB, 7200 rpm. Eksperyment polegał na uruchomieniu pojedynczego procesu korzystającego z dużej ilości pamięci, którego działanie opisano w Podrozdziale 6.1. Rysunek 6.8 zawiera wyniki testu w formie wykresu. Kiedy rozmiar zapisywanego przez proces obszaru przekroczył 1850 MiB rozpoczęła się intensywna wymiana stron pamięci. Dysk twardy wykazywał niewielką przewagę do czasu, kiedy program zaczął zapisywać fragmenty pamięci o wielkości przekracza-



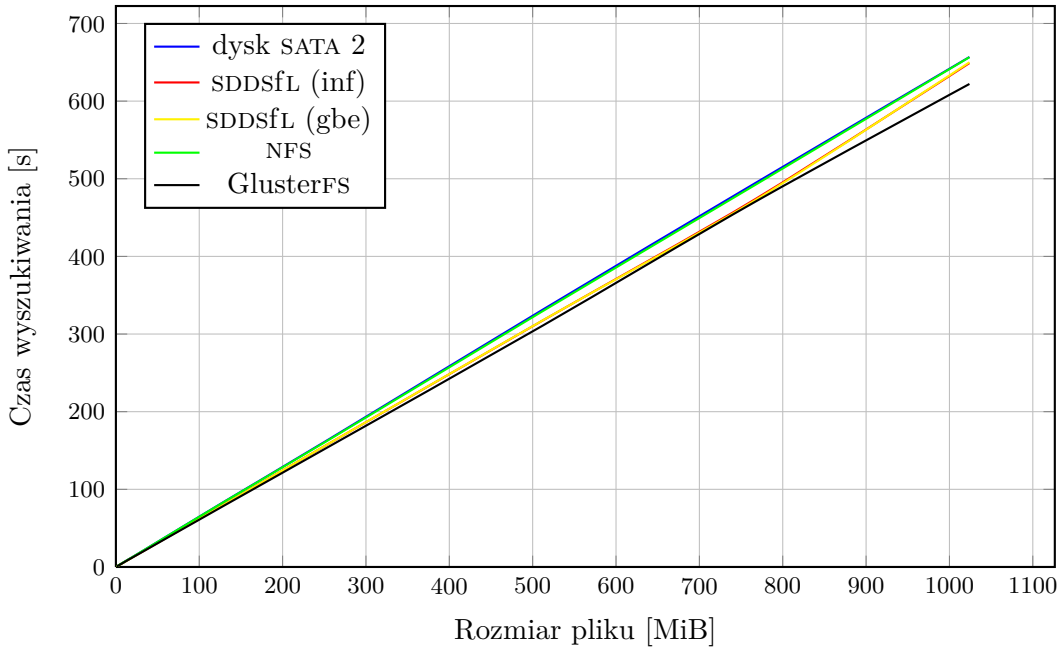
Rysunek 6.8: Czas zapisu dużego obszaru RAM

jącej 1907 MiB. Wówczas jego wydajność zaczęła gwałtownie maleć. Prototyp SDDSfL utrzymywał ją na prawie stałym poziomie, co ostatecznie dało ponad sześciokrotnie lepszy rezultat niż w przypadku nośnika dyskowego.Wynik ten jest porównywalny z opublikowanymi w artykułach [36,38,42] rezultatami Distributed Anemone, MemX i Nswap[2]. Niestety, testy ujawniły że zastosowanie prototypu SDDSfL jako przestrzeni wymiany może prowadzić do załamania (ang. *crash*) systemu, który z niego korzysta. Takie zachowanie zostało odnotowane także dla innych rozwiązań, takich jak NFS lub Network Block Device [43,93]. W przypadku SDDSfL istnieją dwie przyczyny tego zjawiska. Pierwszą są narzuty czasowe związane z obsługą podziałów wiaderek. Ponieważ w systemie Linux rozmiar przestrzeni wymiany przyjmuje się jako dwukrotnie większy od wielkości RAM, to problem ten można ominąć. W tym celu należy najpierw użyć SDDSfL jako zwykłego urządzenia blokowego. Kiedy osiągnie ono określony rozmiar i wystąpią wszystkie niezbędne do tego podziały, to wtedy można je przekształcić w partycję wymiany. Takie rozwiązanie zastosowano w eksperymencie. Drugą przyczyną jest zakleszczenie (ang. *deadlock*), które może się pojawić, kiedy w systemie uruchomionych jest kilka procesów intensywnie korzystających z RAM. Zjawisko to jest prawdopodobnie spowodowane rywalizacją między procesem wymiany, a podsystemem obsługi sieci. Linux stosuje algorytm wymiany o nazwie PFRA (ang. *Page Frame Reclamation Algorithm*), który dba, aby w systemie zawsze były dostępne wolne ramki [2,37]. Podsystem obsługi sieci stara się z kolei pozyskać wolne ramki na strony buforujące pakiety wychodzące lub przychodzące. Distributed Anemone, który jest rozwiązaniem dedykowanym dla wymiany stron rozwiązuje ten problem poprzez stosując własny protokół sieciowy. Jego użycie pozwala na zastosowanie statycznej liczby buforów, ale wiąże się też z bezpośrednią obsługą urządzenia sieciowego.

Ostatnia kategoria testów wydajności związana jest z aplikacjami, które wymagają sekwencyjnego dostępu do danych i są zorientowane na obliczenia. Takie aplikacje najczęściej wykonują operacje I/O, które dotyczą danych o niewielkich rozmiarach [2,4]. Jako przedstawiciela tej klasy aplikacji użyto programu wyszukującego wystąpienia wzorców w pliku tekstowym, który

[2]Rozwiązania te są opisane w Rozdziale 2.

stosuje algorytm Boyera-Moore'a, w wersji zarówno jedno, jaki i dwu wątkowej. Eksperyment przeprowadzono na węźle systemu wielokomputerowego klasy MPP wyposażonym w dwurdzeniowy procesor i RAM o wielkości 2 GiB. Testom poddano dysk twardy SeaGate Barracuda, SATA 2, 160 GiB, 7200 rpm, prototyp SDDSfL korzystający zarówno z sieci Gigabit Ethernet, jak i InfiniBand oraz dwa rozproszone systemy plików: NFS, który korzystał wyłącznie z sieci Gigabit Ethernet i GlusterFS z włączoną opcją paskowania, który wykorzystywał sieć InfiniBand do transmisji.

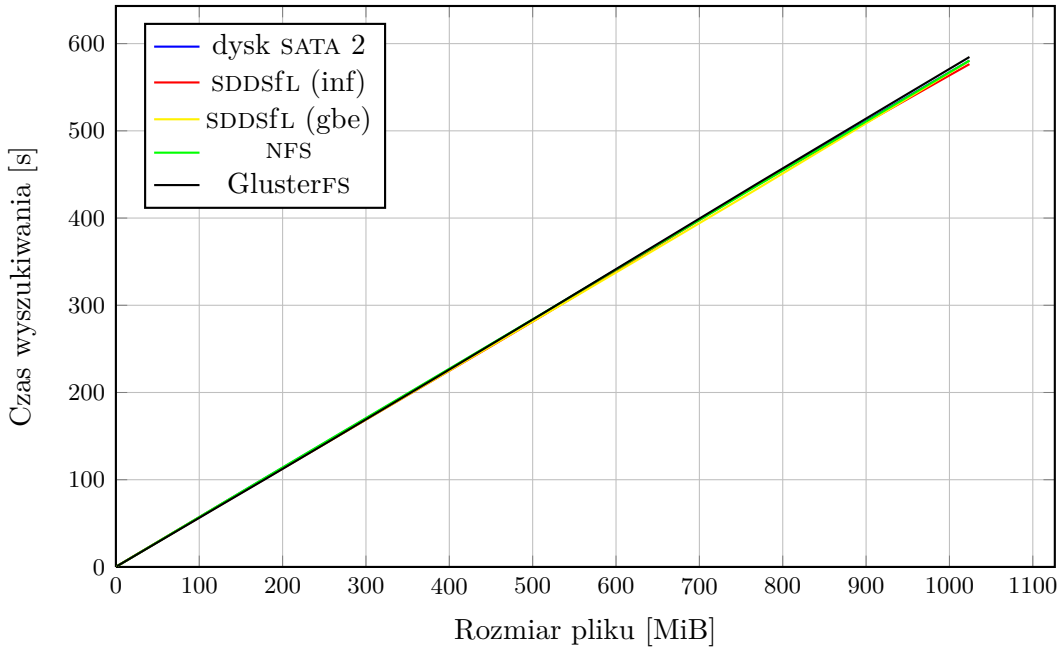


Rysunek 6.9: Wyszukiwanie wzorca w pliku tekstowym (jeden wątek)

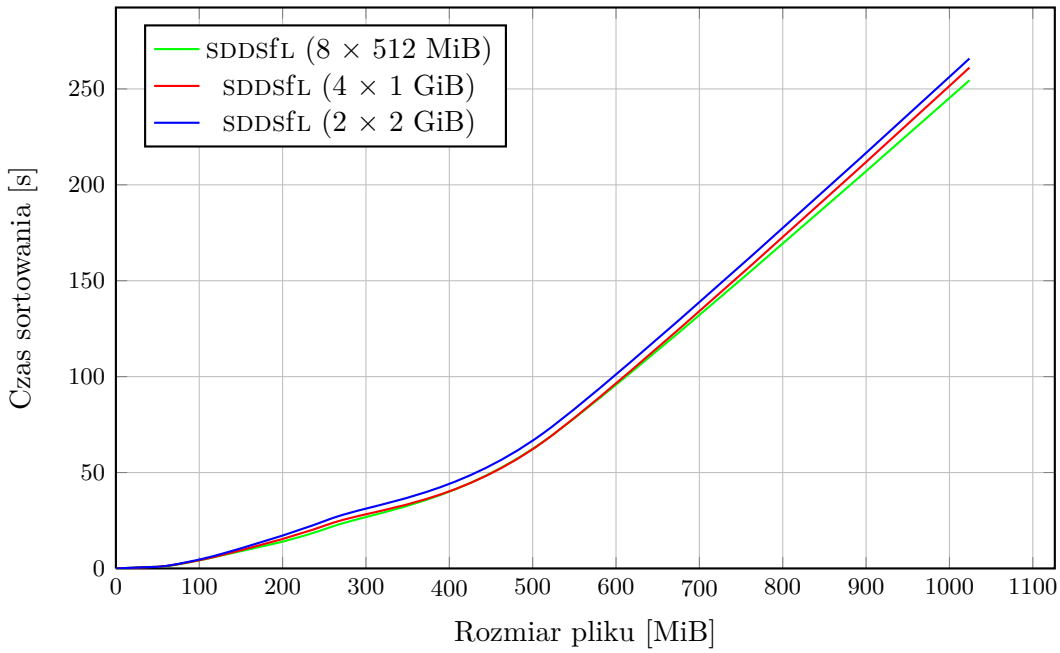
W testach z użyciem jednowątkowej aplikacji niewielką przewagę nad pozostałymi rozwiązaniami uzyskał GlusterFS (Wykres 6.9). W przypadku użycia programu korzystającego z dwóch wątków rezultaty uzyskiwane przez poszczególne rozwiązania są zbieżne (Wykres 6.10).

Ważną cechą SDDS jest skalowalność. Implementacja SDDSfL powinna również posiadać tę cechę. Aby to sprawdzić, prototyp SDDSfL został poddany testom z wykorzystaniem sortowania plików. Pierwsza część eksperymentów badała czasy wykonania sortowania dla różnych konfiguracji serwerów SDDSfL pracujących w jednej z dwóch sieci - Gigabit Ethernet lub InfiniBand. Druga część dotyczyła badania zachowania SDDSfL pracującego z jednym i z dwoma klientami w sieci InfiniBand. Wszystkie testy przeprowadzono na dedykowanym multikomputerze klasy MPP. Wykres 6.11 przedstawia wyniki sortowania plików o wielkości rekordu równej 4 KiB przy użyciu sieci InfiniBand i dla trzech konfiguracji pliku SDDSfL. W pierwszej plik SDDSfL składał się z ośmiu wiaderek o rozmiarach 512 MiB każde, w drugiej z czterech wiaderek, które miały pojemność 1 GiB (ta konfiguracja była stosowana również we wcześniejszych testach), w trzeciej występowały dwa wiaderka o rozmiarze 2 GiB. Dla każdej z konfiguracji uzyskane czasy sortowania były porównywalne. Jeszcze bardziej jest to widoczne na Wykresie 6.12, który zawiera wyniki sortowania plików o rozmiarze rekordu równym 2 KiB, dla tych samych konfiguracji pliku SDDSfL oraz tego samego rodzaju sieci komputerowej.

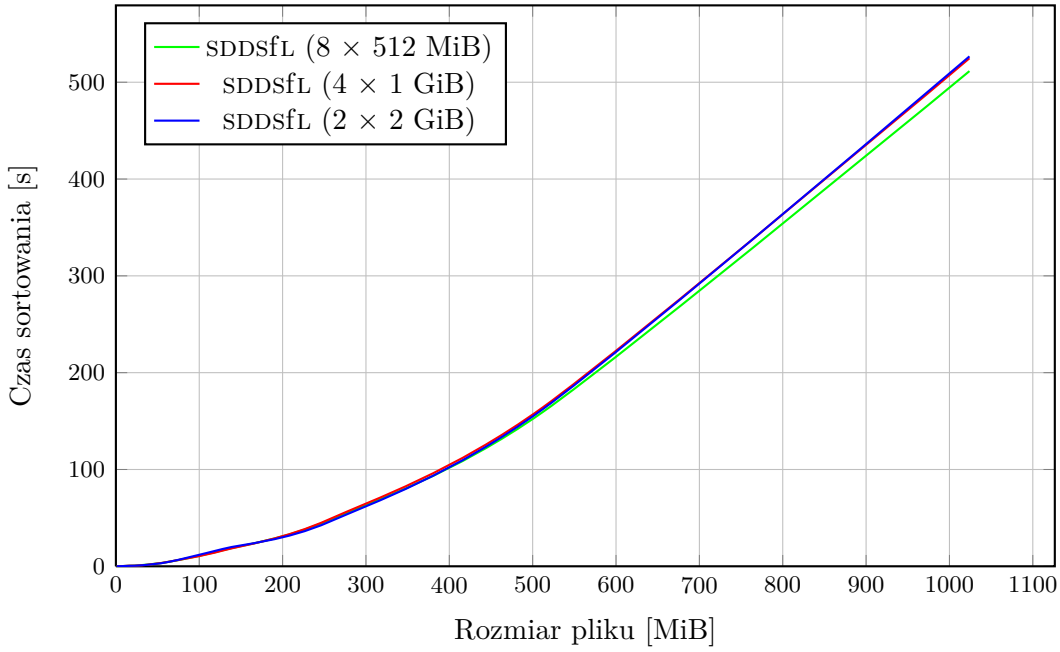
W przypadku sieci Gigabit Ethernet uzyskano podobne rezultaty przy sortowaniu plików o rozmiarze rekordu równym 2 KiB (Wykres 6.14). Dla plików o wielkości rekordu 4 KiB (Wykres 6.13) niewielką przewagę dla dużych zbiorów zyskała konfiguracja składająca się z ośmiu



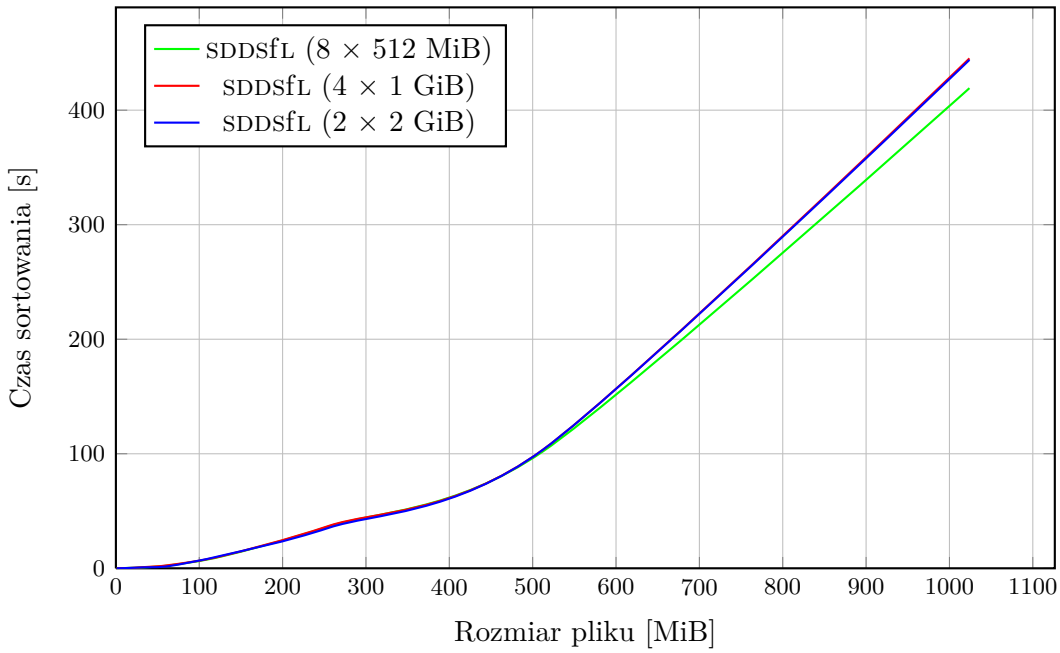
Rysunek 6.10: Wyszukiwanie wzorca w pliku tekstowym (dwa wątki)



Rysunek 6.11: Skalowalność serwerów SDDSfL dla sieci InfiniBand (pliki rekordów o wielkości 4 KiB)



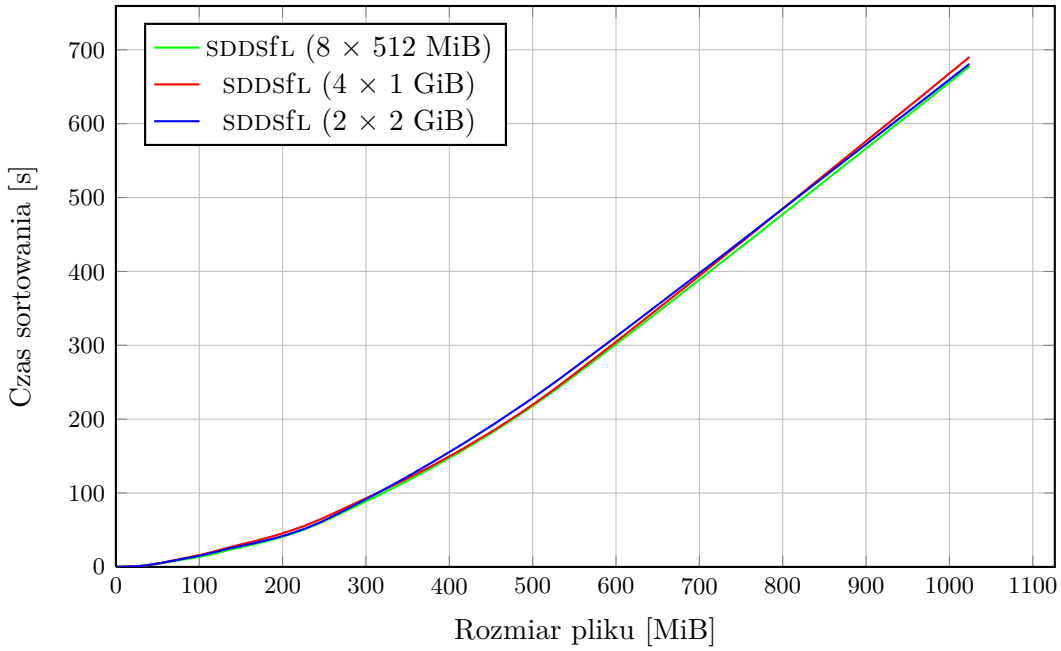
Rysunek 6.12: Skalowalność serwerów SDDSfL dla sieci InfiniBand (pliki rekordów o wielkości 2 KiB)



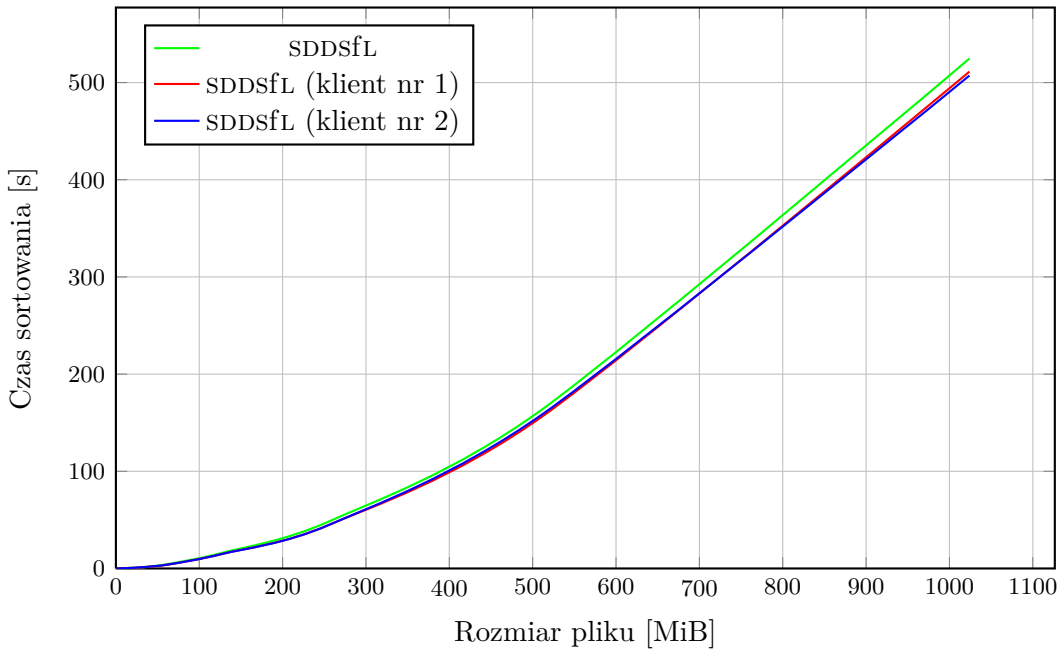
Rysunek 6.13: Skalowalność serwerów SDDSfL dla sieci Gigabit Ethernet (pliki rekordów o wielkości 4 KiB)

wiaderek o pojemności 512 MiB.

Wykres 6.15 prezentuje wyniki testu dla prototypu SDDSfL, który pracuje z jednym i z dwoma klientami. Test przeprowadzono w sieci InfiniBand. Ponownie czasy działania dla obu konfiguracji są porównywalne. Ten eksperyment ujawnił jednak pewną wadę przyjętego schematu obsługi wielu klientów (Podrozdział 5.3). W przyjętym rozwiązaniu, kiedy klientów jest wielu, to może



Rysunek 6.14: Skalowalność serwerów SDDSfL dla sieci Gigabit Ethernet (pliki rekordów o wielkości 2 KiB)



Rysunek 6.15: Skalowalność klientów SDDSfL dla sieci InfiniBand (pliki rekordów o wielkości 2 KiB)

dojść do większej liczby podziałów wiaderek niż wtedy, kiedy z pliku SDDSfL korzysta tylko jeden klient. Teoretycznie pociąga to za sobą konieczność zwiększenia liczby serwerów (wiaderek) c-krotnie, gdzie c oznacza liczbę klientów. W praktyce, w czasie testu zastosowano jedno dodatkowe wiaderko.

6.3. Podsumowanie

Wyniki testów, które zostały przedstawione w tym rozdziale pozwalają stwierdzić, że SDDSfL jest efektywną metodą wirtualizacji pamięci, która dzięki swojej funkcjonalności może zastąpić rozproszone systemy plików lub nawet urządzenia blokowe w zastosowaniach wymagających intensywnych operacji wejścia-wyjścia i swobodnego dostępu do danych. W przypadku aplikacji zorientowanych na obliczenia i korzystających z dostępu sekwencyjnego wydajność SDDSfL nie jest gorsza od innych zbadanych rozwiązań.

Ponieważ podziały wiaderek SDDSfL podczas testów sortowania plików występowały podczas ich kopiowania lub w trakcie pierwszej serii testów, to trudno jest ocenić wpływ czasu podziału na całkowity czas wykonania aplikacji. Podziały występują jednak stosunkowo rzadko. Można więc przyjąć, że poza nielicznymi wyjątkami, jak zastosowanie SDDSfL jako urządzenia wymiany, czasy trwania podziałów nie wpływają znacząco na efektywność działania aplikacji.

Przeprowadzone eksperymenty potwierdziły także dobrą skalowalność SDDSfL. Wydajność tego rozwiązania nie zależy znacząco do liczby i pojemności zastosowanych wiaderek oraz od liczby obsługiwanych klientów.

Opracowana implementacja SDDSfL nie została porównana doświadczalnie z innymi realizacjami SDDS, gdyż jedyną upublicznioną implementację SDDS opracowano dla systemu Windows NT i oparto na architekturze RP*. Ponadto, jest to rozwiązanie całościowo opracowane dla przestrzeni użytkownika [108]. Wydaje się, że porównanie implementacji tak różnych koncepcji nie ma głębszego uzasadnienia, a w razie zaistnienia takiej potrzeby powinno być przedmiotem odrębnych analiz.

7. Wnioski

Ten rozdział stanowi podsumowanie pracy. Pierwszy podrozdział nawiązuje do tezy rozprawy i rozwiązania postawionego w niej problemu oraz wkładu, jaki stanowi ta praca w badania nad wirtualizacją rozproszonej pamięci multikomputera. Druga część zawiera krótką charakterystykę SDDSfL. Trzeci podrozdział traktuje o możliwych kierunkach badań nad Skalowalnymi, Rozproszonymi Strukturami Danych dla systemu Linux.

7.1. Podsumowanie wyników badań

Analiza stanu wiedzy na temat wirtualizacji rozproszonej pamięci systemów wielokomputerowych (Rozdział 2) pozwoliła sklasyfikować trzy obszary zastosowania tej techniki (wsparcie dla pamięci operacyjnej, rozproszoną pamięć dzieloną DSM i wsparcie dla usług zdalnych) oraz ustalić trzy główne kryteria, które są istotne w takich zastosowaniach (skalowalność, wydajność i odporność na błędy). Przegląd rozwiązań w zakresie wirtualizacji rozproszonej RAM multikomputera pozwolił ocenić, że Skalowalne, Rozproszone Struktury Danych spełniają te wymagania w znaczącym stopniu. Główną przeszkodą w stosowaniu ich na szeroką skalę jest konieczność dokonywania modyfikacji aplikacji, aby mogły one współpracować z SDDS.

W Rozdziale 3 rozprawy stwierdzono, że istnieje analogia między klasycznymi technikami pamięci wirtualnej [2–4], a Skalowalnymi, Rozproszonymi Strukturami Danych [14, 27, 29]. SDDS są więc metodą wirtualizacji rozproszonej RAM systemu wielokomputerowego stosowaną na poziomie aplikacji użytkowej. Ponieważ w większości przypadków zarządzanie pamięcią wirtualną realizowane jest przez system operacyjny, to zostało postawione pytanie o możliwość przeniesienia oprogramowania nadzorującego SDDS na poziom systemowy i o efektywność takiego rozwiązania.

W wyniku analizy możliwości realizacji takiego założenia powstało rozwiązanie nazwane SDDSfL. Jest to wersja SDDS LH*, w której rolę klienta-zarządcy SDDS pełni programowy sterownik wirtualnego urządzenia blokowego [90]. Taka realizacja oprogramowania klienckiego pozwala uniknąć konieczności przystosowywania aplikacji użytkowych do współpracy z SDDS, a nawet zastosować SDDSfL w roli urządzenia wymiany dla klasycznych metod pamięci wirtualnej, takich jak stronicowanie na żądanie. Oznacza to również, że rozszerzony został zakres zastosowań SDDS na obszar wspomagania pamięci operacyjnej.

Rozdział 4 zawiera opis architektury zaproponowanego rozwiązania problemu postawionego w tezie rozprawy. Aby uprościć implementację SDDSfL na poziomie jądra systemu operacyjnego umieszczono jedynie klienta. Pozostałe elementy tego oprogramowania osadzone są w przestrzeni użytkownika. W strukturze oprogramowania klienckiego wyróżniono osobny wątek zajmujący się komunikacją z węzłami multikomputera, na których uruchomione są serwery SDDSfL oraz mechanizm watchdog odpowiedzialny za retransmisję zagubionych komunikatów. Zaproponowano także podzielenie procesu serwera na trzy wątki, z których jeden zajmuje się obsługą żądań klientów, a pozostałe dwa operacjami podziałów wiaderek.

Opis opracowanej prototypowej implementacji SDDSfL zamieszczony jest w Rozdziale 5 rozprawy. Klient-sterownik urządzenia blokowego został zrealizowany w postaci modułu, który może być załadowany i usunięty z jądra bez konieczności restartu systemu. Tworząc ten moduł rozwiązano problemy związane z prowadzeniem komunikacji z innymi węzłami multikomputera przy pomocy gniazd, synchronizacji wątku transmitującego i innych elementów sterownika oraz komunikacji z przestrzenią użytkownika. W serwerze struktura wiaderka została zrealizowana za pomocą tablic wskaźników i dynamicznej alokacji pamięci. Do lokalizowania bloków danych w wiaderku użyto algorytmu adresowania otwartego z podwójnym haszowaniem.

W ramach prac badawczych przeprowadzono eksperymenty, w których porównano efektywność SDDSfL, innych urządzeń blokowych oraz rozproszonych systemów plików dla różnych klas zastosowań. Testy wykazały, że w przypadku aplikacji zorientowanych na wejście-wyjście [2] prototyp SDDSfL wykazuje efektywność wyższą od większości nośników dyskowych i rozproszonych systemów plików. Do tej grupy aplikacji zaliczane są również systemy baz danych, które są głównym zastosowaniem oryginalnych implementacji SDDS [74]. W przypadku aplikacji zorientowanych na obliczenia efektywność SDDSfL w przeprowadzonych testach była porównywalna z dyskami twardymi i rozproszonymi systemami plików. Zgodnie z definicją SDDS prototyp SDDSfL odznacza się dobrą skalowalnością, co również zostało wykazane doświadczalnie.

Oryginalnym wkładem pracy w dziedzinę badań nad wirtualizacją rozproszonej pamięci systemów wielokomputerowych jest zaproponowane rozwiązanie problemu zarządzania plikiem SDDS na poziomie systemu operacyjnego oraz opracowanie architektury i prototypowej implementacji tego rozwiązania.

7.2. Zalety i wady SDDSfL

Przyjęte decyzje projektowe wpływają na właściwości implementacji SDDSfL. Poniżej przedstawiona jest lista jej zalet i wad z punktu widzenia aplikacji systemów wielokomputerowych.

7.2.1. Zalety

W przeciwieństwie do oryginalnych implementacji Skalowalnych, Rozproszonych Struktur Danych implementacja SDDSfL nie wymaga ingerencji w kod źródłowy aplikacji celem dostosowania jej do współpracy z tym rozwiązaniem. Dotyczy to programów, które korzystają z takich urządzeń blokowych takich jak dyski twarde, czy macierze RAID. Istnieje również możliwość użycia SDDSfL jako urządzenia wymiany, co umożliwia współpracę praktycznie każdej aplikacji z tym rozwiązaniem. Niestety, w takim zastosowaniu SDDSfL nie zapewnia jeszcze stabilnej pracy systemu operacyjnego.

Prototyp SDDSfL wykazuje dobrą skalowalność. Ma to istotne znaczenie dla aplikacji, które muszą zapisywać informacje na urządzeniach blokowych, ale nie można przewidzieć jaki będzie rozmiar tych informacji. Dodatkowo SDDSfL oferuje czasy dostępu porównywalne lub mniejsze od tych, które osiągane są w tradycyjnych napędach dyskowych.

Ponieważ SDDSfL jest urządzeniem blokowym, to można na nim osadzić praktycznie dowolny system plików, który jest stosowany na lokalnych dyskach twardych. Dzięki temu taki system może być dobierany zgodnie z wymaganiami aplikacji, które będą z nim współpracowały. Można również zastosować dowolny mechanizm zapewniania spójności danych przechowywanych przez SDDSfL.

Architekturę SDDSfL można uogólnić na inne systemy operacyjne niż Linux. W przypadku systemów kompatybilnych z Uniksem (ang. *Unix-like*) możliwe jest również bezpośrednie przeniesienie części implementacji (serwerów SDDSfL i koordynatora podziałów) z Linuksa.

Serwery SDDSfL są zwykłymi aplikacjami działającymi w trybie użytkownika. Takie rozwiązanie jest korzystne w porównaniu z alternatywą jaką byłoby zaimplementowanie serwerów na poziomie jądra systemu operacyjnego. Przestrzeń użytkownika (ang. *user space*) posiada proste API, które pozwala wygodnie przydzielać i zarządzać dużymi obszarami pamięci operacyjnej. W aplikacjach użytkowych można łatwo lokalizować i usuwać błędy. Ujawnienie błędu w programie użytkownika nie powoduje destabilizacji pracy pozostałych elementów systemu. Implementacja w trybie użytkownika oprogramowania komunikującego się poprzez sieć jest także pożądana ze względu na bezpieczeństwo systemu. W przypadku serwera SDDSfL taka korzyść nie ma miejsca, jeśli zostanie on uruchomiony z prawami użytkownika uprzywilejowanego, aby uniknąć wymiany stron pamięci przydzielonych na wiaderko.

Implementacja klienta SDDSfL w postaci modułu jądra pozwala na wdrożenie tego oprogramowania do użytku bez konieczności restartu komputera i kompilacji kodu źródłowego jądra.

7.2.2. Wady

Sposób realizacji oprogramowania klienckiego SDDSfL stanowi równocześnie jego zaletę i wadę. Błąd w tym oprogramowaniu może spowodować załamanie całego systemu operacyjnego węzła multikomputera, na którym zostało ono uruchomione. Utrudnia to znajdowanie i usuwanie usterek w kodzie klienta. Ponieważ klient SDDSfL komunikuje się z serwerami przez sieć, to może stać się również celem ataków zmierzających do naruszenia bezpieczeństwa systemu. Dosyć częste zmiany w API jądra Linuksa utrudniają konserwację (ang. *maintenance*) kodu klienta SDDSfL [109].

Obsługa wielu klientów przez SDDSfL jest obecnie dosyć prymitywna. Klienci nie mogą współdzielić obrazu systemu plików osadzonego na SDDSfL. Dodatkowo przyjęte rozwiązanie może wymagać użycia większej liczby wiaderek niż wtedy, kiedy z SDDSfL korzysta tylko jeden klient. Należy nadmienić, że w niektórych zastosowaniach brak współdzielenia danych przez klientów może być korzystny, np. klienci mogą mieć zainstalowane diametralnie różne systemy plików.

7.3. Kierunki dalszych badań

Choć można uznać, że wydajność prototypu SDDSfL w porównaniu z nośnikami dyskowymi i rozproszonymi systemami plików jest zadowalająca, to interesujące byłoby zbadanie kilku możliwości jej zwiększenia. Jedną z nich jest oparcie implementacji SDDSfL o architekturę RP* zamiast LH*. W tej architekturze występuje duża lokalność odwołań (ang. *references*), która mogłaby wpłynąć korzystnie na efektywność SDDSfL współpracującego z aplikacjami wymagającymi sekwencyjnego dostępu do danych. Inna podejście polega na wykorzystaniu możliwości tworzenia przez klienta równoległych zapytań do serwerów. W oprogramowaniu serwera można zmienić sposób zarządzania i adresowania rekordami wewnątrz wiaderka. Dosyć częstym rozwiązaniem tej kwestii jest zastosowanie algorytmu LH [86].

Wyniki badań zamieszczone w [104] sugerują, że największy wpływ na wydajność dysków twardych ma pamięć podręczna dla operacji odczytów i buforowanie zapisów. W systemie Linux istnieje Warstwa Operacji Blokowych, która automatycznie przydziela bufory na operacje I/O pełniące wyżej wymienione funkcje, ale dyski twarde posiadają zazwyczaj takie rozwiązania zaimplementowane również na poziomie sprzętowym. Urządzenie blokowe tworzone przez klienta SDDSfL nie posiada takiego mechanizmu, ale możliwe jest jego symulowanie. Wymagałoby to zastąpienia prostego bufora w wątku transmisyjnym strukturą danych umożliwiającą szybkie wyszukiwanie danych, taką jak np. drzewo pozycyjne (ang. *radix tree*) [33].

Prototypowa implementacja SDDSfL jest dostosowana do korzystania z sieci komputerowych, które obsługiwane są z pomocą stosu protokołów TCP/IP. Podczas testów umożliwiło to uruchomienie SDDSfL zarówno z siecią Gigabit Ethernet, jak i InfiniBand. W przypadku tej ostatniej to rozwiązanie nie pozwala wykorzystać wszystkich możliwości oferowanych przez tę sieć. W szczególności nie jest dostępny tryb RDMA umożliwiający wielokrotnie szybszą transmisję danych [10]. Korzystnym byłoby więc dostosowanie klienta SDDSfL do bezpośredniego korzystania z tej sieci.

Jednym z zastosowań prototypu SDDSfL jest praca w charakterze urządzenia wymiany dla realizacji klasycznej pamięci wirtualnej (np. dla stronicowania na żądanie). Niestety, w takim zastosowaniu może on spowodować naruszenie stabilności systemu. Podobny problem jest spotykany również w przypadkach zastosowania innych rozwiązań, jak rozproszony system plików NFS lub Network Block Device (NBD) [43] w charakterze urządzeń wymiany. Zostały opracowane poprawki (ang. *patch*), które usuwają jego przyczynę, ale wymagają one kompilacji jądra systemu i restartu komputera. Inne podejście zastosowano w Distributed Anemone (MemX), gdzie klient w postaci modułu jądra samodzielnie obsługuje urządzenie sieciowe, pomijając odpowiedzialne za to sterowniki [38, 42]. Dzięki temu jest on w stanie utrzymywać niewielką i statyczną ilość pamięci przeznaczonej na bufory dla pakietów odbieranych i wysyłanych. To rozwiązanie pozwala wyeliminować zakleszczenia (ang. *deadlocks*) z podsystemem odpowiedzialnym za wymianę stron. Pożądane byłoby przeniesienie takiego rozwiązania do klienta SDDSfL.

Opracowano szereg architektur SDDS odpornych na różne kategorie błędów [87, 110–114], które szerzej są opisane w Dodatku A. Zastosowanie mechanizmów uodparniających na błędy, oferowanych przez te architektury w prototypie SDDSfL wykracza poza zakres tej pracy. Mimo to, warte uwagi jest zbadanie możliwości zastosowania tych rozwiązań w SDDSfL.

Tak jak wspomniano w poprzednim podrozdziale bieżąca realizacja obsługi wielu klientów w prototypie SDDSfL nie umożliwia bezpośredniego współdzielenia plików przez węzły wielokomputera z uruchomionym oprogramowaniem klienckim. Podstawowym problemem związanym z realizacją takiej możliwości jest zapewnienie spójności współdzielonych danych, a w szczególności powiadamianie klientów o zmianach obrazu systemu plików. Interfejs sterowników urządzeń blokowych w systemie Linux definiuje prototypy dwóch funkcji, o nazwach `media_changed()` oraz `revalidate_disk()`, które przeznaczone są do obsługi zdarzeń powiązanych ze zmianą nośnika lub odłączeniem urządzenia wymiennego [39]. Zbadanie możliwości implementacji współdzielenia danych przez klientów z wykorzystaniem tych funkcji powinno być jednym z celów przyszłych prac na klientem SDDSfL.

Implementację SDDSfL można wzbogacić o funkcję automatycznego wykrywania dostępnych w sieci klientów, serwerów i koordynatorów podziałów. Rozwiązania tego typu określane są mianem odkrywania usług (ang. *service discovery*). Zastosowano je w Distributed Anemone [38, 42] oraz (w formie serwera nazw [5]) w publicznie dostępnej implementacji SDDS RP* dla systemu Windows NT [108]. Wdrożenie takiego mechanizmu pomogłoby uprościć czynności związane z konfigurowaniem SDDSfL.

Dodatek A.

Architektury SDDS odporne na błędy

Problem tolerowania błędów (ang. *fault-tolerant*) w Skalowalnych, Rozproszonych Strukturach Danych SDDS wykracza poza zakres badań prowadzonych w ramach niniejszej pracy. Jest to jednak istotne zagadnienie, podobnie jak tolerowanie błędów w innych elementach wielokomputerów i systemów rozproszonych [5, 12, 87, 115]. Bieżący rozdział zawiera krótki przegląd najbardziej istotnych rozwiązań jakie w tej dziedzinie opracowano dla SDDS LH*. Jego celem jest jedynie zasygnalizowanie problematyki.

Błędy występujące w SDDS mogą być podzielone na dwie kategorie: dotyczące sterowania i dotyczące danych [110]. Pierwsza część rozdziału traktuje o błędach sterowania, druga o błędach danych.

A.1. Błędy sterowania

W oryginalnej architekturze SDDS LH* pojedynczy punkt awarii (ang. *single point of failure*) stanowi koordynator podziałów (SC). W [27] zaproponowano wariant struktury LH*, w którym podziały nie są nadzorowane przez SC, lecz za pomocą przekazywanego między serwerami SDDS LH* tokena. Kolejność przekazywania wyznaczana jest za pomocą Wzoru 4.2. Takie rozwiązanie, choć eliminuje potrzebę istnienia koordynatora podziałów, to nie czyni SDDS LH* całkowicie odporną na błędy sterowania - token może zostać zagubiony w trakcie przesyłania przez sieć lub zniszczony na skutek awarii serwera będącego w jego posiadaniu. Koordynator podziałów odgrywa znaczącą rolę w wariantach architektury SDDS LH* tolerujących błędy danych - jest odpowiedzialny za odtwarzanie wiaderek i rekordów. Może również wykrywać awarie serwerów zarządzających wiaderkami. Jego wyeliminowanie nie jest więc jednoznacznie korzystne, potrzebne są inne rozwiązania. Praca [110] zawiera analizę możliwych błędów sterowania w SDDS LH* oraz definicje protokołów, które zapobiegają skutkom najpoważniejszych uszkodzeń. W przypadku SC błędy sterowania mogą polegać na wyłączeniu koordynatora (niemy SC) lub na wysyłaniu błędnych komunikatów „you split". Skutki takich błędów można wyeliminować stosując Potrójną Nadmiarowość Modularną (ang. *Triple Modular Redundancy* - TMR) [5, 115]. W przypadku SDDS zastosowanie tej techniki oznacza, że w normalnym trybie pracy, kiedy nie występują błędy, nadzór nad podziałami sprawują dwa SC. Aby nastąpił podział wiaderka, serwer, który nim zarządza musi otrzymać dwa zgodne komunikaty od obu SC. Koordynatory podziałów dodatkowo przesyłają między sobą informacje o decyzjach jakie podejmują. Jeśli zostaną wykryte rozbieżności, to powoływany jest trzeci koordynator i decyzja podjęta przez większość jest uważana za właściwą. Schemat ten pozwala na zastąpienie wadliwego SC nowym. W pracy [110] zaproponowano również poprawkę w Algorytmie 4, która zapobiega przesłaniu

żądania klienta poza plik SDDS. Do takiej sytuacji może dojść, jeśli popełni on błąd adresowania polegający na wygenerowaniu adresu o większej wartości, niż adres logiczny ostatniego wiaderka w pliku. Oryginalny algorytm weryfikacji poprawności adresu żądania przez serwer zakłada, że nieprawidłowy adres logiczny zawsze będzie miał wartość mniejszą od prawidłowego adresu.

A.2. Błędy danych

Zaproponowano wiele wariantów architektury SDDS LH* tolerujących błędy danych. Część z nich inspirowana jest konstrukcją macierzy RAID [3, 5].

W wariancie architektury LH* oznaczonej jako SDDS LH^*_{M} zastosowano technikę odzwierciedleń (ang. *mirroring*), aby otrzymać następujące własności [87]:

- dostępność wszystkich danych po wystąpieniu pojedynczego błędu węzła-serwera,
- dostępność wszystkich danych po wielokrotnych błędach węzła-serwera,
- dostępność większości danych w przypadku wystąpienia błędów w dwóch węzłach-serwerach,
- równoważenie obciążenia po wystąpieniu błędów (ang. *load-balancing*),
- wydajne odzyskiwanie danych po wystąpieniu błędów.

W tej architekturze zbiór danych podzielony jest na dwa plik SDDS oznaczone F_1 i F_2, które stanowią swoje lustrzane odbicie, tzn. zawierają te same dane. W przypadku awarii, któregoś z serwerów pliku F_1 dane mogą być odzyskane ze stowarzyszonego z nim serwera pliku F_2. Odwrotny scenariusz również jest możliwy. Istnieją dwie odmiany SDDS LH^*_{M}:

- SA-mirrors, w której oba pliki mają ten sam rozmiar, stosują te same funkcje haszujące i politykę kontroli podziałów (Modyfikacje obydwu plików wykonywane są równolegle.),
- SD-mirrors, w której pliki mogą różnić się niektórymi z podanych wyżej parametrów (Stosowana jest leniwa modyfikacja plików.).

Wariantem architektury SDDS LH^*_{M} jest SDDS LH^*_s, w której zastosowano paskowanie (ang. *striping*) [2, 4] rekordów na poziomie bitów. Dzięki temu zwiększono bezpieczeństwo danych związane z ich poufnością oraz dodano informację nadmiarową w postaci bitów parzystości [114]. Podział na paski na poziomie bitów może negatywnie wpływać na efektywność operacji wyszukiwania rekordów w pliku SDDS. Jeżeli jest ona ważniejsza od bezpieczeństwa danych, to zamiast tworzenia pasków na poziomie bitów można ten podział przeprowadzić na poziomie umownie przyjętych atrybutów.

Architektura SDDS LH^*_g, podobnie jak wyżej opisane architektury gwarantuje odporność na błędy pojedynczego wiaderka, ale ma mniejsze wymagania w zakresie zapotrzebowania na dostępną rozproszoną RAM multikomputera [111,113]. W tej architekturze stosowane są dwa rodzaje pliku SDDS: plik danych i plik parzystości. Rekordy z pliku danych organizowane są w grupy, w ten sposób, że każdy z rekordów należących do tej samej grupy znajduje się w innym wiaderku niż pozostałe. Dodatkowo dla każdej z grup tworzony jest rekord parzystości, który przechowywany jest w wiaderku należącym do pliku parzystości. Dzięki własnościom operacji różnicy symetrycznej (XOR) możliwe jest odtworzenie zarówno rekordów danych, jak i rekordów parzystości w przypadku awarii pojedynczego wiaderka.

Im więcej wiaderek zawiera plik SDDS tym większe jest prawdopodobieństwo, że wystąpi błąd więcej niż jednego wiaderka, przed którego skutkami nie chroni żaden ze schematów opisanych

wyżej. Konieczne stało się więc opracowanie architektur, w których zapewnienie dostępności danych (ang. *availability*) byłoby skalowalne wraz z rozmiarem pliku SDDS.

Architektura SDDS LH$^*_{sa}$ zapewnia skalowalność dostępności danych bez konieczności reorganizacji pliku SDDS [116]. W tej architekturze również są stosowane pliki parzystości, ale ich liczba zwiększa się wraz z rozmiarem pliku SDDS. Początkowo istnieje tylko jeden taki plik. Każde wiaderko danych zawiera dodatkowy atrybut nazywany poziomem dostępności. Informuje on ile plików parzystości jest związanych z tym wiaderkiem.

Architektura SDDS LH$^*_{\mathrm{RS}}$ [112, 117] jest wariantem SDDS LH$^*_{sa}$. Podobnie jak ona zapewnia skalowalną dostępność danych, ale rekordy parzystości są w jej przypadku wyliczane za pomocą kodu detekcyjno-korekcyjnego Reeda-Solomona używającego rozszerzonego ciała skończonego $GF(2^8)$ lub $GF(2^{16})$ [77]. Zastosowanie kodu RS pozwala zmniejszyć zapotrzebowanie na ilość pamięci potrzebnej do zapamiętania bitów parzystości oraz przyspieszyć ich wyliczanie.

Bibliografia

[1] Cameron Purdy. *Getting Coherence: Introduction to Oracle Coherence Data Grid*. http://www.youtube.com/watch?v=4Sq45B8wAXc, 2007.

[2] Andrew Stanley Tanenbaum. *Systemy operacyjne*. Helion, Gliwice, 2010.

[3] William Stallings. *Systemy operacyjne - Struktura i zasady budowy*. Wydawnictwa Naukowe PWN SA, Warszawa, 2006.

[4] Abraham Silberschatz, Peter B. Galvin, Greg Gagne. *Podstawy systemów operacyjnych*. Wydawnictwa Naukowo-Techniczne, Warszawa, wydanie siódme, 2005.

[5] Andrew Stanley Tanenbaum, Martin van Steen. *Systemy rozproszone. Zasady i paradygmaty*. WNT, Warszawa, 2006.

[6] Erik Hagersten, Anders Landin, Seif Haridi. DDM - A Cache-Only Memory Architecture. *IEEE Computer*, 25:44–54, 1992.

[7] Luca Benini, Giovanni De Micheli. Networks on Chip: A New Paradigm for Systems on Chip Design. *In Proceedings of Conference on Design, Automation and Test in Europe*, strony 418–419. IEEE Computer Society, 2002.

[8] William J. Dally, Brian Towles. Route Packets, Not Wires: On-Chip Interconnection Networks, 2001.

[9] Bill Nitzberg, Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, 1991.

[10] InfiniBand Trade Association. http://www.infinibandta.org, 2010.

[11] Myrinet overview. http://www.myricom.com/myrinet/overview/, 2009.

[12] George Coulouris, Jean Dollimore, Tim Kindberg. *Systemy rozproszone: podstawy i projektowanie*. WNT, Warszawa, 1998.

[13] J. Protić, M. Tomašević, V. Milutinović. A survey of distributed shared memory systems. *Hawaii International Conference on System Sciences*, strony 74–84, 1995.

[14] W. Litwin, M-A Neimat, D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. *Proceedings of the Twentieth International Conference on Very Large Databases*, strony 342–353, Santiago, Chile, 1994.

[15] *Memory Virtualization Primer*. http://www.rnanetworks.com/cache-architecture, 2009.

[16] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, T. Ruwart. The Global File System: A File System for Shared Disk Storage. *Submitted to the IEEE Transactions on Parallel and Distributed Systems*, 1997.

[17] Dan Kusnetzky. *Sorting out the different layers of virtualization.* http://blogs.zdnet.com/virtualization/?p=170, 2007.

[18] Xen — The Xen virtual machine monitor. http://www.cl.cam.ac.uk/research/srg/netos/xen/, 2009.

[19] Kerrighed — Linux clusters made easy. http://www.kerrighed.org, 2010.

[20] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, strony 190–201, 2000.

[21] GlusterFS. http://www.gluster.org, 2010.

[22] Clive Cook. *Memory Virtualization, the Third Wave of Virtualization.* http://vmblog.com/archive/2008/12/14/memory-virtualization-the-third-wave-of-virtualization.aspx, 2009.

[23] Tom Matson. *RNA Networks - Virtual Memory.* http://www.motionbox.com/videos/0096d1b61d12efc58f, 2009.

[24] Dan Kusnetzky. *RNA Networks and memory virtualization.* http://blogs.zdnet.com/virtualization/?p=655, 2009.

[25] Eric A. Anderson, Jeanna M. Neefe. An Exploration of Network RAM. Raport instytutowy, Computer Science Division, University of California at Berkeley, 1994.

[26] Michail D. Flouris, Evangelos P. Markatos. The Network RamDisk : Using Remote Memory on Heterogeneous NOWs. *Cluster Computing*, 2:281–293, 1999.

[27] Witold Litwin, Marie-Anna Neimat, Donovan A. Schneider. LH* — a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[28] Witold Litwin. Linear hashing: a new tool for file and table addressing. *VLDB '1980: Proceedings of the sixth international conference on Very Large Data Bases*, strony 212–223. VLDB Endowment, 1980.

[29] Cédric du Mouza, Witold Litwin, Philippe Rigaux. SD-Rtree: A scalable distributed rtree. *ICDE*, strony 296–305, 2007.

[30] Robert Love. *Kernel Linux - Przewodnik programisty.* Wydawnictwo Helion, Gliwice, 2004.

[31] Abraham Silberschatz, James L. Peterson, Peter B. Galvin. *Podstawy systemów operacyjnych.* Wydawnictwa Naukowo-Techniczne, Warszawa, wydanie drugie, 1993.

[32] Douglas E. Comer, J Griffioen. A New Design for Distributed Systems: The Remote Memory Model. *Proceedings of the Summer 1990 Usenix Conference*, 1990.

[33] Donald E. Knuth. *Sortowanie i wyszukiwanie*, wolumen 3 serii *Sztuka programowania*. WNT, Warszawa, 2002.

[34] Liviu Iftode, Kai Li, Karin Petersen. Memory Servers for Multicomputers. *Compcon Spring '93, Digest of Papers*, strony 538–547, 1993.

[35] Evangelos P. Markatos, George Dramitinos. Implementation of a Reliable Remote Memory Pager. *In USENIX Annual Technical Conference*, strony 177–190, 1996.

[36] Tia Newhall, Sean Finney, Kuzman Ganchev, Michael Spiegel. Nswap: A network swap module for linux clusters. *Proceedings of the International Conference on Parallel and Distributed Computing*, wolumen 2790, strony 1160–1169, 2003.

[37] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[38] Michael R. Hines, Jian Wang, Kartik Gopalan. Distributed Anemone: Transparent Low-Latency Access to Remote Memory. *HiPC*, strony 509–521, 2006.

[39] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[40] Jonathan Corbet. *A simple block driver*. http://lwn.net/Articles/58719/, 2003.

[41] Jonathan Corbet. *Driver Porting: block layer overview*. http://lwn.net/Articles/24990/, 2003.

[42] Michael R. Hines, Kartik Gopalan. MemX: supporting large memory workloads in Xen virtual machines. *VTDC '07: Proceedings of the 3rd international workshop on Virtualization technology in distributed computing*, strony 1–8, New York, NY, USA, 2007.

[43] Shuang Liang, R. Noronha, D.K. Panda. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. *Cluster Computing, IEEE International Conference on*, 0:1–10, 2005.

[44] Ming-Chit Tam, Jonathan M. Smith, David J. Farber. A taxonomy-based comparison of several distributed shared memory systems. *SIGOPS Oper. Syst. Rev.*, 24(3):40–67, 1990.

[45] Ulrich Drepper. *What Every Programmer Should Know About Memory*. http://people.redhat.com/drepper/cpumemory.pdf, 2007.

[46] A. Judge, P.A. Nixon, V.J. Cahill, B. Tangney, S. Weber. Overview of Distributed Shared Memory. Raport instytutowy, Trinity College Dublin, 1998.

[47] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25:63–79, 1992.

[48] D.B. Gustavson, Qiang Li. The Scalable Coherent Interface (SCI). *Communications Magazine, IEEE*, 34(8):52–63, 1996.

[49] David V. James, Anthony T. Laundrie, Stein Gjessing, Gurindar Sohi. Scalable coherent interface. *Computer*, 23(6):74–77, 1990.

[50] Oliver Pugh, Jeff Cowhig, Jonathan Crowe, Eoin Blacklock. *Scalable Coherent Interface (SCI)*. http://ntrg.cs.tcd.ie/undergrad/4ba2.05/group12/index.html, 2010.

[51] David H. D. Warren, Seif Haridi. Data Diffusion Machine - A Scalable Shared Virtual Memory Multiprocessor. *FGCS*, strony 943–952, 1988.

[52] S. Lucci, I. Gertner, A. Gupta, U. Hegde. Reflective-memory multiprocessor. *Hawaii International Conference on System Sciences*, 0:85, 1995.

[53] Creve Maples, Larry Witrie. Merlin: A Superglue for Multicomputer Systems. *In Proc. of the 35th Compcon 90 Conf*, strony 73–81, 1990.

[54] B. Fleisch, G. Popek. Mirage: a coherent distributed shared memory design. *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, strony 211–223, New York, NY, USA, 1989. ACM.

[55] Partha Dasgupta, Jr. Richard J. LeBlanc, Mustaque Ahamad, Umakishore Ramachandran. The Clouds Distributed Operating System. *Computer*, 24(11):34–44, 1991.

[56] Raymond C. Chen, Partha Dasgupta. Implementing consistency control mechanisms in the Clouds distributed operating system. *ICDCS*, strony 10–17, 1991.

[57] Jelica Protić, Milo Tomašević, Veljko Milutinović. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel Distrib. Technol.*, 4(2):63–79, 1996.

[58] J. K. Bennett, J. B. Carter, W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, strony 168–176, New York, NY, USA, 1990. ACM.

[59] Kai Li, Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[60] Brian N. Bershad, Matthew J. Zekauskas, Wayne A. Sawdon. The Midway Distributed Shared Memory System. Raport instytutowy, Pittsburgh, PA, USA, 1993.

[61] Roberto Bisiani, Alessandro Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Trans. Computers*, 37(8):930–945, 1988.

[62] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, Hya Dwarkadas, Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *In Proceedings of the 1994 Winter Usenix Conference*, strony 115–131, 1994.

[63] Roberto Bisiani, Mosur Ravishankar. PLUS: a distributed shared-memory system. *SIGARCH Comput. Archit. News*, 18(3a):115–124, 1990.

[64] Anant Agarwal, Beng-Hong Lim, David Kranz, John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. *IN PROCEEDINGS OF THE 17TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, strony 104–114, 1990.

[65] Donald Yeung, John Kubiatowicz, Anant Agarwal. Multigrain shared memory. *ACM Trans. Comput. Syst.*, 18(2):154–196, 2000.

[66] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, John Hennessy. The performance impact of flexibility in

the Stanford FLASH multiprocessor. *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, strony 274–285, New York, NY, USA, 1994. ACM.

[67] David Cheriton, Hendrik A. Goosen, Patrick D. Boyle. ParaDiGM: A Highly Scalable Shared-Memory Multi-Computer Architecture. *IEEE Computer*, 24:33–46, 1991.

[68] Leonidas Kontothanassis, Robert Stets, Galen Hunt, Umit Rencuzogullari, Gautam Altekar, Sandhya Dwarkadas, Michael L. Scott. Shared memory computing on clusters with symmetric multiprocessors and system area networks. *ACM Trans. Comput. Syst.*, 23(3):301–335, 2005.

[69] Sandhya Dwarkadas, Robert Stets, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil. Cashmere-VLM: Remote Memory Paging for Software Distributed Shared Memory. *Parallel Processing Symposium, International*, 0:153, 1999.

[70] Chistian Osendorfer, Jie Tao, Carsten Trinitis, Martin Mairandres. ViSMI: Software Distributed Shared Memory for InfiniBand Clusters. *Network Computing and Applications, IEEE International Symposium on*, 0:185–191, 2004.

[71] J.B. Carter, D. Khandekar, L. Kamb. Distributed shared memory: where we are and where we should be headed. *Hot Topics in Operating Systems, Workshop on*, strony 119–122, 1995.

[72] Oracle Coherence Knowledge Base Home. http://coherence.oracle.com, 2009.

[73] Robert Devine. Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. *4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, strony 101–114, 1993.

[74] Y. Ndiaye, A. Diene, W. Litwin, T. Risch. AMOS-SDDS: A Scalable Distributed Data Manager for Windows Multicomputers. *14th Intl. Conference on Parallel and Distributed Computing Systems – PDCS 2001* ., 2001.

[75] Witold Litwin. $\text{LH}^*_{\text{RS}}{}^{\text{P2P}}$: A Scalable Distributed Data Structure for P2P Environment. http://video.google.com/videoplay?docid=-7096662377647111009#, 2007.

[76] Witold Litwin, Hanafi Yakouben, Thomas Schwarz. $\text{LH}^*_{\text{RS}}{}^{\text{P2P}}$: A Scalable Distributed Data Structure for P2P Environment. *NOTERE '08: Proceedings of the 8th international conference on New technologies in distributed systems*, strony 1–6, New York, NY, USA, 2008. ACM.

[77] Władysław Mochnacki. *Kody korekcyjne i kryptografia.* Wydawnictwa Politechniki Wrocławskiej, Wrocław, 1997.

[78] S. Gribble, E. Brewer, J. Hellerstein, D. Culler. Scalable, distributed data structures for Internet service construction. *Proceedings of OSDI*, 2000.

[79] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. *Proceedings of the 2001 ACM SIGCOMM Conference*, strony 149–160, 2001.

[80] Gurmeet Singh Manku, Mayank Bawa, Prabhakar Raghavan, Verity Inc. Symphony: Distributed Hashing in a Small World. *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, strony 127–140, 2003.

[81] Sylvia Ratnasamy, Paul Francis, Scott Shenker, Richard Karp, Mark Handley. A Scalable Content-Addressable Network. *In Proceedings of ACM SIGCOMM*, strony 161–172, 2001.

[82] Antony Rowstron, Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[83] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Wprowadzenie do algorytmów*. Wydawnictwa Naukowo-Techniczne, Warszawa, 1998.

[84] TOP500 Supercomputing Sites. http://www.top500.org, 2010.

[85] Linux Kernel. http://kernel.org/, 2010.

[86] Andy D. Pimentel, Louis O. Hertzberger. Evaluation of LH^*_{LH} for a Multicomputer Architecture. *In Proc. of the EuroPar Conference*, strony 217–229, 1999.

[87] W. Litwin, M. a. Neimat. High-Availability LH* Schemes with Mirroring, 1996.

[88] Filesystem in Userspace (FUSE). http://fuse.sourceforge.net/, 2010.

[89] Andrew Stanley Tanenbaum, Albert S. Woodhull. *Operating systems. Design and implementation*. Pearson Education International, Upper Saddle River, 2009.

[90] Arkadiusz Chrobot, Grzegorz Łukawski, Krzysztof Sapiecha. Scalable Distributed Data Structures for Linux-based Multicomputer. *International Symposium on Parallel and Distributed Computing in Kraków, IEEE Computer Society*, strony 424–428, 2008.

[91] W. Richard Stevens. *Programowanie zastosowań sieciowych w systemie Unix*. Wydawnictwa Naukowo-Techniczne, Warszawa, 1995.

[92] Rob Gerth. *Concise Promela Reference*. http://spinroot.com/spin/Man/Quick.html, 1997.

[93] P. T. Ares. *The Network Block Device*. http://www2.linuxjournal.com/article/3778, 2005.

[94] Global Network Block Device. http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/4.8/pdf/Global_Network_Block_Device.pdf, 2010.

[95] Mark Mitchell, Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, Thousand Oaks, CA, USA, 2001.

[96] Serial ATA International Organization. http://www.serialata.org/, 2010.

[97] INCITS Technical Committee T13. http://www.t13.org/, 2010.

[98] SCSI Trade Association. http://www.scsita.org/, 2010.

[99] Lustre a Network Clustering FS. http://www.lustre.org, 2010.

[100] Daniel Robbins. *Common threads: Advanced filesystem implementor's guide, Part 7 — Introducing ext3*. http://www.ibm.com/developerworks/linux/library/l-fs7.html, 2001.

[101] PostgreSQL. http://postgresql.org, 2010.

[102] pgbench (Postgres Benchmark). http://wiki.postgresql.org/wiki/Pgbench, 2010.

[103] Transaction Processing Performance Council. http://www.tpc.org/, 2010.

[104] W. Hsu, A. J. Smith. The performance impact of I/O optimizations and disk improvements. *IBM J. Res. Dev.*, 48(2):255–289, 2004.

[105] John D. Shakshober. *Choosing an I/O Scheduler for Red Hat® Enterprise Linux® 4 and the 2.6 Kernel*. http://www.redhat.com/magazine/008jun05/features/schedulers/, 2010.

[106] Linux: Fair Queuing Disk Schedulers. http://kerneltrap.org/node/580, 2010.

[107] Jens Axboe. *CFQ IO Scheduler*. http://www.linux.org.au/conf/2007/talk/123.html, 2007.

[108] SDDS-2005 version 1.1. http://ceria.dauphine.fr/SDDS-2005/SDDS-2005.HTM, 2010.

[109] Jonathan Corbet. *API changes in the 2.6 kernel series*. http://lwn.net/Articles/2.6-kernel-api/, 2010.

[110] Krzysztof Sapiecha, Grzegorz Łukawski. Fault–tolerant Protocols for Scalable Distributed Data Structures. *Springer-Verlag LNCS 3911*, 2005.

[111] Witold Litwin, Jai Menon, Tore Risch, Thomas J. E. Schwarz. Design Issues For Scalable Availability LH* Schemes with Record Grouping. *Carleton Scientific*, strony 200–211, 1999.

[112] Witold Litwin, Rim Moussa, Thomas Schwarz. LH^*_{RS}—a highly-available scalable distributed data structure. *ACM Trans. Database Syst.*, 30(3):769–811, 2005.

[113] Witold Litwin, Tore Risch. LH^*_g: A High-Availability Scalable Distributed Data Structure By Record Grouping. *IEEE Transactions on Knowledge and Data Engineering*, 14:923–927, 2002.

[114] W. Litwin, M-A Neimat, G. Levy, S. Ndiaye, T. Seck. LH^*_s: a High-availability and High-security Scalable Distributed Data Structure, 1997.

[115] Janusz Sosnowski. *Testowanie i niezawodność systemów komputerowych*. WNT, Warszawa, 2005.

[116] W. Litwin, J. Menon, T. Risch. LH* Schemes with Scalable Availability, 1998.

[117] Witold Litwin, Thomas Schwarz. LH^*_{RS} : A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. *SIGMOD Conference*, strony 237–248, 2000.