Instytut Informatyki Politechniki Poznańskiej

Rozproszony System Zarządzania Odtwarzaniem Mediów w Oparciu o Pesymistyczną Pamięć Transakcyjną

Martin Witczak, Konrad Siek

Raport RB-4/15

Poznań, 2015

Streszczenie

Ewaluacja rozproszonej pesymistycznej pamięci transakcyjnej ma kluczowe znaczenia dla dalszego rozwoju badań i istniejących jej implementacji. Mechanizm ten nie został dotychczas wykorzystany do implementacji aplikacji, która spełniałaby rzeczywiste przypadki użycia. Celem niniejszej pracy jest zbadanie możliwości użycia rozproszonej pesymistycznej pamięci transakcyjnej do synchronizacji w złożonych systemach rozproszonych poprzez przeprowadzenie studium na przykładzie rozproszonego systemu multimedialnego. W ramach pracy przeanalizowano założenia jakie muszę być spełnione przez aplikację, zidentyfikowane potencjalne problemy synchronizacyjne i zaproponowane ich rozwiązania przy użyciu pesymistycznej pamięci transakcyjnej poparte ewaluacją eksperymentalną.

Rozdział 1

Wstęp

Rozwój popularności komputerów i Internetu wymaga od systemów komputerowych obsługi coraz większej liczby użytkowników. W wyniku tego muszą się one stawać coraz bardziej wydajne i niezawodne. Dzięki zastosowaniu systemów wieloprocesorowych możliwe jest rozwiązywanie złożonych problemów obliczeniowych, poprzez jednoczesne wykonywanie niezależnych poleceń na każdym z procesorów. Połączenie wielu takich systemów przy użyciu sieci komputerowej tworzy system rozproszony, gdzie każdy węzeł może wykonywać współbieżnie odrębne obliczenia. Wyniki zbierane ze wszystkich węzłów pozwalają na rozwiązanie problemów wymagających połączonej mocy tysięcy wieloprocesorowych serwerów.

Wykonywanie jednoczesnych operacji przez niezależne części systemu wymaga współdzielenia pomiędzy nimi zasobów. Współbieżne operacje na współdzielonych zasobach mogą prowadzić do uzyskania niepoprawnych wyników, na przykład w sytuacji, gdy jeden węzeł nadpisze wyniki uzyskane przez drugi węzeł. Z tego powodu dostęp do współdzielonych zasobów musi podlegać synchronizacji, aby zachować wzajemne wykluczanie procesów. Obecnie wykorzystywane są do synchronizacji niskopoziomowe mechanizmy takie jak zamki czy semafory. Są one jednak trudne w użyciu, ponieważ wymagają wzięcia pod uwagę oddziaływania na siebie pozornie niepołączonych części systemu. Wykorzystanie prostych rozwiązań, takich jak pojedynczy zamek globalny, pozwala na zachowanie spójności systemu kosztem ograniczonej współbieżności. Przeprowadzono badania [13] w celu analizy alternatywnych do zamków rozwiązań problemu synchronizacji, mające na celu stworzenie nowych mechanizmów charakteryzujących się większą wydajnością i prostotą użycia.

Zaproponowano pamięć transakcyjną [14] – rozwiązanie polegające na objęciu sekcji kodu transakcją, która spełnia konkretne gwarancje jak np. atomowość czy szeregowalność. Pamięć transakcyjna odpowiada za takie wykonanie transakcji, aby te gwarancje były spełnione. Algorytmy sterowania współbieżnością są wykonywane w tle i są transparentne dla programisty. Pamięć transakcyjna może być zastosowana także w systemach rozproszonych. Pozwala ona na tworzenie rozproszonych transakcji, które wykonują kod współbieżnie na wielu węzłach systemu. Rozproszona pamięć transakcyjna wiąże ze sobą jednak dodatkowe wyzwania jak konieczność obsłużenia asynchroniczności czy awarii wę-

złów systemu.

Dotychczasowe implementacje pamięci transakcyjnej wykorzystują optymistyczne sterowanie współbieżnością. Oznacza to, że transakcje wykonywane są niezależnie, a ewentualne konflikty, z powodu modyfikacji tych samych danych współdzielonych, wykrywane są podczas zatwierdzania transakcji. W przypadku ich wystąpienia transakcja jest wycofywana, czyli następuje przywrócenie obiektów do stanu sprzed wykonania transakcji. Transakcja jest następnie wykonywana ponownie, co może powodować wielokrotne wykonanie operacji nieodwracalnych, które obejmuje ta transakcja. Operacje nieodwracalne to operacje powodujące zmianę stanu systemu, której nie można wycofać, jak np. wywołanie systemowe czy komunikacja sieciowa. Efekty wykonania transakcji zawierającej takie operacje nie mogą być w całości odwrócone i są widoczne dla reszty systemu.

Wystąpiono z propozycją użycia w pamięci transakcyjnej pesymistycznego sterowania współbieżnością [1, 29]. W tym podejściu transakcje oczekują na dostęp do współdzielonych obiektów, co skutkuje brakiem konfliktów spowodowanych przez jednoczesny dostęp do zasobów. W efekcie transakcje nie muszą być wycofywane i nie występuje problem z operacjami nieodwracalnymi.

Ewaluacja rozproszonej pesymistycznej pamięci transakcyjnej ma kluczowe znaczenia dla dalszego rozwoju badań i istniejących jej implementacji. Mechanizm ten nie został dotychczas wykorzystany do implementacji aplikacji, która spełniałaby rzeczywiste przypadki użycia. Niniejszy raport zawiera analizę zastosowania rozproszonej pesymistycznej pamięci transakcyjnej do synchronizacji w systemach rozproszonych poprzez przeprowadzenie studium na przykładzie rzeczywistej aplikacji rozproszonej w postaci. rozproszonego systemu kolejnowania i odtwarzania multimediów – szafy grającej.

W ramach raportu przenalizowano założenia, które musi ta aplikacja spełniać. Zwrócono uwagę na analizę potencjalnych problemów synchronizacyjnych oraz zaproponowano ich rozwiązania z użyciem pesymistycznej pamięci transakcyjnej. Wykonana implementacja aplikacji posłużyła do ewaluacji przeprowadzonych rozważań. Dzięki temu raport w sposób praktyczny pokazuje kierunek rozwiązywania rzeczywistych problemów przy użyciu pesymistycznej pamięci transakcyjnej.

Raport składa się z następujących części. W rozdziale 2 omówione zostały zagadnienia synchronizacji w systemie współbieżnym i rozproszonym oraz metody ich rozwiązywania. Rodział 3 opisuje aktualny stan wiedzy na temat pamięci transakcyjnej oraz aplikacje podobne do implementowanej. Rozdział 4 przedstawia Atomic RMI – implementację rozproszonej pesymistycznej pamięci transakcyjnej. W rozdziale 5 zaproponowano aplikację rozproszoną służącą do ewaluacji rozproszonej pamięci transakcyjnej – rozproszoną szafę grającą oraz opisano występujące w niej problemy synchronizacyjne i ich możliwe rozwiązania. Rozdział 6 prezentuje ewaluację zaimplementowanej aplikacji w przypadkach testowych mających na celu ocenę użytej do synchronizacji pamięci transakcyjnej. Rozdział 7 zawiera podsumowanie.

Rozdział 2

Podstawy Teoretyczne

W niniejszym rozdziale omówione zostaną podstawowe pojęcia wymagane dla zrozumienia problemów rozważanych w dalszej części pracy. Zdefiniowany zostanie problem synchronizacji w systemie współbieżnym i rozproszonym oraz metody jego rozwiązywania – zamki oraz pamięć transakcyjna.

2.1 Problem synchronizacji w systemie współbieżnym

Procesy [12], czyli sekwencje operacji wykonywanych jedna po drugiej, nazywamy *współbieżnymi* jeżeli pierwsza operacja pierwszego procesu rozpoczęła się zanim zakończyła się ostatnia operacja drugiego procesu. System, w którym występują procesy współbieżne nazywamy *systemem współbieżnym*. System współbieżny może składać się z wielu procesorów, które mają bezpośredni dostęp do *współdzielonej pamięci* [19], która tworzy wspólną przestrzeń adresową. Komunikacja między procesami zachodzi poprzez odczyty i zapisy do współdzielonej pamięci. W związku z tym procesy współbieżne mogą wymagać dostępu do zasobów systemu w tym samym czasie, co prowadzi do *rywalizacji* o zasoby.

Synchronizacja to proces ustalania kolejności wykonywania operacji w czasie, tak aby uzyskać *wzajemne wykluczanie*. Wzajemne wykluczanie to taka kolejność przydziału zasobów dla każdego procesu aby miał on dostęp do zasobu na wyłączność. W przypadku operacji wykonywanych sekwencyjnie wzajemne wykluczanie jest zachowane ponieważ tylko jedna operacja jest wykonywana jednocześnie. W systemie współbieżnym nie jest możliwe przewidzenie efektów operacji, jeżeli procesy nie wykluczają się wzajemnie podczas operacji na współdzielonych zasobach. W konsekwencji system współbieżny wymaga synchronizacji, jeżeli ma zachować semantykę systemu sekwencyjnego.

2.1.1 Sekcja krytyczna

Sekcje krytyczne [12] są metodą zachowania wzajemnego wykluczenia procesów, czyli zachowania synchronizacji dostępu. Objęcie sekcji kodu, która modyfikuje współdzielone zmienne, sekcją krytyczną oznacza, że:

1. Co najwyżej jeden proces może być jednocześnie wewnątrz sekcji krytycznej – zapewnia to wzajemne wykluczanie procesów.
2. Proces, który potrzebuje wejść do sekcji krytycznej uzyska do niej dostęp w skończonym czasie.
3. Proces musi opuścić sekcję krytyczną w skończonym czasie, czyli każda operacja na współdzielonej zmiennej musi się zakończyć.

Sekcja krytyczna ogranicza dostęp procesów w następujący sposób:

1. Jeżeli żaden proces nie jest w sekcji krytycznej, to proces może wejść do sekcji krytycznej niezwłocznie.
2. Jeżeli proces znajduje się w sekcji krytycznej, to pozostałe procesy próbujące wejść do sekcji krytycznej muszą zaczekać.
3. Jeżeli proces opuszcza sekcję krytyczną na dostęp do której oczekują inne procesy, to jeden z nich uzyskuje dostęp do sekcji krytycznej.
4. Powyższe procesy muszą zostać wykonane w skończonym czasie.

2.1.2 Zamki

Zamek [12] jest konstrukcją pozwalającą na zachowanie gwarancji zapewnianych przez sekcje krytyczne. Spośród wielu procesów oczekujących na sekcję krytyczną tylko jeden z nich zajmuje zamek, który ogranicza dostęp do tej sekcji kodu. Po opuszczeniu sekcji proces zwalnia zamek i kolejny proces może go zająć, co zapewnia wzajemne wykluczenie procesów. W systemie, w którym procesy do poprawnego wykonania wymagają dostępu na wyłączność do więcej niż jednego zasobu współdzielonego może wystąpić *zakleszczenie [3].* Zakleszczenie to stan, w którym dwa lub więcej procesów oczekuje w nieskończoność na zasoby tak, że blokują się wzajemnie. Do powstania zakleszczenia wymagane są następujące warunki *[6]*:

1. Tylko jeden proces może korzystać z zasobu – wzajemne wykluczenie.
2. Procesy zachowują wyłączność do zasobów, które już posiadają, w oczekiwaniu na dostęp do pozostałych.
3. Proces nie może zostać wywłaszczony z zasobu dopóki nie zostanie on zakończony.
4. Istnieje cykliczne odwołanie pomiędzy procesami oczekującymi na zasoby.

Zakleszczenie to bardzo poważny problem w systemach współbieżnych, który prowadzi do zatrzymania przetwarzania i jest trudny do przewidzenia nim nie wystąpi. Możliwe rozwiązania problemu zakleszczenia procesów wykorzystują trzy strategie [19]:

1. Zapobieganie zakleszczeniom: osiągane jest poprzez przydzielenie wszystkich wymaganych przez proces zasobów przed jego rozpoczęciem lub poprzez zaniechanie procesu, który posiada wymagane zasoby.
2. Unikanie zakleszczeń: proces otrzymuje dostęp do współdzielonych zasobów tylko

jeżeli przewidywany stan systemu po jego zakończeniu jest bezpieczny.

3. Wykrywanie zakleszczeń: polega na badaniu stanu procesów i ich przydzielonych zasobów pod kątem zaistnienia zakleszczenia. W przypadku wystąpienia zakleszczenia proces zakleszczony jest zaniechany.

Użycie zamków niesie ze sobą także zagrożenie wystąpienia *inwersji priorytetów* [21]. Przykładem inwersji priorytetów jest sytuacja w której istnieją trzy procesy H, M, L z odpowiednio wysokim, średnim i niskim priorytetem. Proces L posiada dostęp do zasobu R, do którego dostęp po chwili próbuje dostać proces z wyższym priorytetem H. Proces L zostaje jednak wywłaszczony przez proces M o wyższym priorytecie, ale niższym niż oczekujący na zasób proces H. W tej sytuacji proces M otrzymał dostęp do zasobu mimo niższego priorytetu niż H.

Osobnym problemem jest też możliwość wystąpienia *konwojowania* [14], czyli sytuacji w której proces posiadający dostęp do zasobu zostaje zawieszony powodując brak możliwości kontynuowania pracy i blokując pozostałe procesy oczekujące na zasób.

Rozróżnia się dwa określenia względem granularności zastosowanych zamków. *Drobnoziarnistość* oznacza objęcie każdego obiektu współdzielonego osobnym zamkiem, dzięki temu możliwa jest większa współbieżność i czas oczekiwania na zwolnienie sekcji jest krótszy ale możliwe jest wystąpienie zakleszczenia. Zamki *gruboziarniste* oznaczają objęcie wielu obiektów współdzielonych jednym zamkiem. Są one łatwiejsze w użyciu ale powodują niepotrzebne oczekiwanie na wykonanie instrukcji, które niekoniecznie wymagają synchronizacji. Skrajnym przypadkiem gruboziarnistości jest zastosowanie *zamka globalnego* czyli zamka, który obejmuje wszystkie polecenia wykonywane przez procesy. Prowadzi to do eliminacji współbieżnego przetwarzania i procesy działają sekwencyjnie. Miarą efektywności zastosowanych zamków jest *współzawodnictwo,* czyli liczba procesów oczekujących na zwolnienie zamka przez proces znajdujący się w sekcji krytycznej. Procesy oczekujące są bezczynne, co oznacza, że przetwarzanie zachodzi nieefektywnie i prawdopodobnie możliwa jest optymalizacja, która polega na eliminacji czekania przez procesy.

2.1.3 Pamięć transakcyjna

Unikanie zakleszczeń oraz dążenie do drobnoziarnistości zastosowanych zamków jest znaczącą trudnością podczas programowania systemów współbieżnych. Inne podejście do problemu wzajemnego wykluczania polega na użyciu struktur nieblokujących (*lock-free*) oraz wolnych od oczekiwania (*wait-free*) [13]. Celem jest uzyskanie konstrukcji, w której nie występują problemy inwersji priorytetów, zakleszczenia oraz konwojowania przy zachowaniu wysokiej współbieżności procesów. *Optymistyczna pamięć transakcyjna* [14] została zaproponowana jako odpowiedź na te problemy. Pamięć transakcyjna opiera się na idei *transakcji*, która jest powszechna w bazach danych. Transakcja to skończona sekwencja operacji wykonywanych przez pojedynczy proces, która spełnia pewne własności spójności, np.:

1. *Szeregowalność* [26]: efekt wykonania współbieżnie grupy transakcji jest taki sam jak wykonanie ich w pewnym uszeregowanym porządku,

2. *Atomowość* [33]: modyfikacje współdzielonych zasobów dokonane poprzez transakcje są wykonywane w całości albo w całości wycofywane.

W optymistycznej pamięci transakcyjnej przedstawionej w [14] kod objęty transakcją wykonywany jest współbieżnie przez wszystkie procesy. Wszystkie modyfikacje współdzielonych obiektów przechowywane są przez lokalną pamięć transakcji i są niewidoczne dla innych procesów do czasu jej zatwierdzenia. W momencie zatwierdzania transakcji sprawdza się czy pozostałe procesy wykonały konfliktujący zapis do tych samych zasobów współdzielonych. Konflikt może być rozwiązany na wiele sposobów. Podstawową metodą jest wycofanie konfliktującej transakcji i powtarzanie jej tak długo, jak długo występują konflikty. Efektem jest zwiększenie współbieżności poprzez jednoczesne wykonywanie sekcji krytycznej kosztem potencjalnej konieczności powtarzania transakcji.

Powtarzanie tych samych operacji wielokrotnie powoduje, że konieczne jest wzięcie pod uwagę *operacji nieodwracalnych* [31]. Operacje nieodwracalne to operacje, których wielokrotne wykonanie powoduje zmianę stanu systemu i nie jest możliwe przywrócenie jego stanu sprzed wykonania operacji [34]. Przykładem takich operacji może być wywołanie systemowe, komunikacja sieciowa albo operacje zapisu na dysku. Rozwiązaniem tego problemu może być wykluczenie używania operacji nieodwracalnych w transakcjach, ograniczenie współbieżności do sekwencyjnego wykonania transakcji. Operacje nieodwracalne mogą być też buforowane do czasu aż transakcja nie zostanie zatwierdzona.

Pamięć transakcyjna zwalnia programistę z konieczności przypisywania zamków dla konkretnych obiektów oraz koordynowania operacji ich zajmowania i zwalniania. Zapewnia przy tym maksymalną współbieżność procesów wykonujących sekcję krytyczną tak długo jak modyfikacje danych nie konfliktują ze sobą. Należy zwrócić jednak uwagę na narzut wydajnościowy implementacji pamięci transakcyjnych względem zamków. Implementacje pamięci transakcyjnej wymagają trzymania w ramach każdej transakcji logów zmian i wykonywania kosztownych operacji w pamięci podczas, gdy w przypadku zamków jedynym kosztem jest operacja zajęcia i zwolnienia zamka. W celu określenia czy dla danej sekcji krytycznej wydajniejsze będzie użycie zamków czy pamięci transakcyjnej zdefiniowano następujący wzór [32]:

$$r * k \geq c \tag{2.1}$$

r – średnia liczba powtórzeń transakcji
k – narzut wydajnościowy
c – średnia liczba współzawodniczących procesów o zajęcie zamka

Sekcja krytyczna wykona się szybciej przy użyciu transakcji niż zamka jeżeli liczba powtórzeń transakcji pomnożona przez narzut wydajnościowy pamięci transakcyjnej względem zamków będzie mniejsza niż liczba wątków oczekujących na zwolnienie zamka.

Problem operacji nieodwracalnych oraz wielokrotnego powtarzania transakcji był motywacją do wprowadzenia *pesymistycznej pamięci transakcyjnej* [1]. Pesymistyczna pamięć transakcyjna jest nową techniką implementacji pamięci transakcyjnej w której transakcje są wykonywane dokładnie jeden raz i nie są wycofywane. Możliwe jest powtórzenie transakcji lub jej wycofanie jednak tylko na bezpośrednie polecenie programisty. W tym

podejściu transakcje również wykonywane są współbieżnie, jednak w przypadku dostępu do tych samych obiektów wykonują one operacje na nim sekwencyjnie powodując zlikwidowanie konfliktów pomiędzy transakcjami. Pesymistyczna pamięć transakcyjna zostanie szerzej omówiona w ujęciu rozproszonym oraz w rozdziale 4 omawiającym jej konkretną implementację – Atomic RMI.

2.2 Problem synchronizacji w systemie rozproszonym

System rozproszony to zbiór niezależnych węzłów, komunikujący się i koordynujący wykonywane procesy tylko poprzez przekazywanie wiadomości przez sieć [19, 8]. System spełniający tą definicję charakteryzuje:

1. Brak wspólnego zegara fizycznego: Ponieważ komunikacja zachodzi wyłącznie poprzez przekazywanie wiadomości przez sieć, to zawsze istnieje granica dokładności do której da się zsynchronizować zegary. W związku z tym nie można zakładać, że czas lokalny jest wszędzie taki sam i opierać na tym działania systemu.
2. Brak współdzielonej pamięci: System rozproszony jest zbiorem różnych systemów posiadających pamięć lokalną i komunikacja pomiędzy nimi zachodzi tylko przez sieć.
3. Współbieżność i autonomiczność: Każdy węzeł systemu wykonuje swoją pracę niezależnie i posiada własne zasoby. Wymagana jest więc koordynacja ich pracy, która jest wykonywana poprzez przesyłane wiadomości. Zwiększenie wydajności systemu można uzyskać poprzez dodanie kolejnych węzłów.
4. Odporność na awarie: Każdy element systemu może ulec awarii. Przy projektowaniu systemu rozproszonego należy także wziąć pod uwagę problemy sieciowe – komunikaty mogą dotrzeć opóźnione, w innej kolejności lub zaginąć.

System rozproszony pozwala na łączenie wielu zasobów, które nie są dostępne w jednym miejscu, w logiczną całość za pomocą sieci. Umożliwa to wspólne korzystanie z zasobów przez wiele węzłów, z czego wynika konieczność zachowania synchronizacji dostępu do współdzielonych zasobów, aby osiągnąć wzajemne wykluczanie procesów należących do wielu węzłów.

2.2.1 Zamki

Wzajemne wykluczanie w systemie rozproszonym [19] oznacza, że tylko jeden proces może wykonywać sekcje krytyczną w danym czasie. Nie jest możliwe jednak użycie lokalnych zmiennych współdzielonych do implementacji zamków czy semaforów do implementacji wzajemnego wykluczania. Uzyskanie wzajemnego wykluczania wymaga użycia algorytmów korzystających wyłącznie z przekazywania wiadomości do komunikacji pomiędzy procesami. Zaprojektowanie algorytmu wzajemnego wykluczania jest złożone, ponieważ dodatkowo należy wziąć pod uwagę nieprzewidywalne opóźnienia podczas przesyłania wiadomości i brak znajomości stanu globalnego systemu. Dzięki powstaniu algorytmów takich

jak algorytm Lamporta [20], Maekawy [23] czy Ricarta-Agrawali [27] możliwe jest uzyskanie wzajemnego wykluczenia w systemie rozproszonym co pozwala na stworzenie *zamka rozproszonego,* który jest rozproszonym odpowiednikiem zamka w systemie współbieżnym.

2.2.2 Pamięć transakcyjna

Rozproszona pamięć transakcyjna to metoda zachowania synchronizacji dostępu do współdzielonych obiektów w systemie rozproszonym. Podobnie jak pamięć transakcyjna w systemie współbieżnym, polecenia są objęte atomowymi transakcjami. Transakcje w systemie rozproszonym mogą obejmować jednak wiele węzłów jednocześnie.

Pierwszym podejściem pozwalającym na użycie pamięci transakcyjnej w systemie rozproszonym jest jej replikacja [7, 15, 18]. W tym podejściu zbiór zasobów współdzielonych aplikacji jest replikowany na wszystkie węzły. *Replikacja maszyny stanów* [18] jest pesymistycznym podejściem, w którym pierwszym krokiem jest uszeregowanie transakcji na wszystkich węzłach systemu a następnie niezależne ich wykonanie na każdej z replik. W systemie deterministycznym ustalenie takiej samej kolejności transakcji dla każdej z replik pozwala na zachowanie spójności systemu. *Replikacja transakcyjna* [18] polega zaś na optymistycznym wykonaniu atomowej transakcji na jednej replice, bez koordynacji pomiędzy replikami przed lub w trakcie trwania transakcji. W przypadku konfliktu taka transakcja jest wycofywana i powtarzana. Wyniki przetwarzania są przesyłane w atomowych modyfikacjach do wszystkich replik.

Pamięci transakcyjne, które nie wykorzystują pełnej replikacji współdzielonych zbiorów danych, można podzielić względem miejsca wykonania transakcji i przemieszczania danych [28]:

1. W modelu *data-flow* transakcja wykonywana jest w całości na węźle, które ją zlecił. Obiekty modyfikowane przez transakcje są dynamicznie migrowane na ten węzeł i transakcja wykonuje operacje na tych lokalnych kopiach. Podejście to nie sprawdza się jeżeli obiekty nie mogą być migrowane z powodu ich rozmiaru lub restrykcji bezpieczeństwa.
2. W modelu *control-flow* obiekty nie są przemieszczane. Transakcje wywołują metody tych obiektów poprzez zdalne wywołania. W konsekwencji kod wykonywany przez tranakcje jest rozproszony pomiędzy węzłami.

Rozdział 3

Przegląd Systemów i Zastosowań Pamięci Transakcyjnej

Rozdział opisuje aktualny stan wiedzy na temat pamięci transakcyjnej. Omówione zostały testy wydajności pamięci transakcyjnej oraz zastosowania pamięci transakcyjnej w aplikacjach i systemach.

3.1 Testy wydajności pamięci transakcyjnej

Podstawowym narzędziem służącym do oceny implementacji pamięci transakcyjnej są testy wydajności. W niniejszej podsekcji omówione zostaną istniejące testy pamięci transakcyjnej, które różnią się złożonością oraz stopniem odzwierciedlenia realnych zastosowań pamięci transakcyjnej. Oceniają one także różne aspekty pamięci transakcyjnej co pozwala uzyskać szeroki pogląd na istniejące problemy pamięci transakcyjnej i kierunek dalszych badań.

3.1.1 STMBench7

STMBench7 [11] to propozycja testu wydajności oceniającego implementacje pamięci transakcyjnej w wersji współbieżnej. Motywacją do jego powstania był brak testów, które oceniałyby realistyczne przypadki użycia pamięci transakcyjnej w zastosowaniach współbieżnych. Istniejące wcześniej testy polegały na operacjach na prostych strukturach (np. drzewa czerwono-czarne) lub w warunkach ograniczonej współbieżności, gdzie pamięć transakcyjna nie mogła pokazać swoich zalet. Dodatkowo testy były wykonywane w różnych środowiskach i różnych językach programowania co utrudniało porównanie ich wyników. Wyciągnięcie niepoprawnych wniosków na podstawie niemiarodajnych testów wydajnościowych mogłoby skłonić badaczy do skupienia się na nieistotnych aspektach implementacji.

Zaproponowany został test wydajności na podstawie kompleksowej, realistycznej i obiektowej aplikacji wielowątkowej, która wykonuje operacje współbieżne na zbiorze grafów i

indeksów. STMBench7 opiera się na teście *OO7* [5], który testuje wydajność transakcji w obiektowych bazach danych. W odróżnieniu od OO7, STMBench7 ocenia wydajność dla rzeczywistych wzorów obciążenia systemu. Nie zależy on także od konkretnych implementacji pamięci transakcyjnej, jest łatwy w użyciu i pozwala na porównanie wyników. Autorzy podkreślają, że STMBench7 to dopiero pierwszy krok do stworzenia pełnowartościowego testu wydajności pamięci rozproszonej w aplikacji wielowątkowej.

3.1.2 STAMP

Stanford Transactional Application for Multi-Processing (STAMP) [24] to zbiór testów służących ewaluacji systemów pamięci transakcyjnej w systemach współbieżnych. Motywacją do jego powstania, podobnie jak w przypadku STMBench7, był brak testów pamięci transakcyjnej, które przedstawiałyby realistyczne przypadki jej wykorzystania.

STAMP to zbiór 8 aplikacji i 30 zestawów danych wejściowych mających na celu pokrycie problemu testowania pamięci transakcyjnej:

- wszerz – przetestowanie różnych domen aplikacji i algorytmów, ze szczególnym uwzględnieniem tych których nie można łatwo zrównoleglić bez mechanizmów synchronizacji, ponieważ one najmocniej mogą skorzystać z użycia pamięci transakcyjnej,
- wgłąb – szeroka gama rozważonych wariantów wykonania transakcji:
  - różne rozmiary zbiorów danych,
  - różne długość transakcji,
  - zmienny stopień współzawodnictwa,
- przenośność – ocena różnych klas pamięci rozproszonych: programowych, sprzętowych i hybrydowych.

W ramach badań przetestowane zostało 6 implementacji pamięci transakcyjnej poprzez zmierzenie wydajności 8 aplikacji testowych w 20 wariantach. Pozwoliło to porównać ich wydajność i wykryć charakterystyki pamięci transakcyjnych, które nie były wcześniej rozważane i wymagają dalszych badań.

3.1.3 Eigenbench

Eigenbench [16] to test wydajności pamięci transakcyjnej mający na celu ocenę wydajności pamięci transakcyjnej. W ramach projektu zaproponowane zostały przez autorów *Eigen-charakterystyki* – zbiór ośmiu ortogonalnych charakterystyk opisujących aplikacje wykorzystujące pamięć transakcyjną:

1. Współbieżność – liczba współbieżnych wątków,
2. Rozmiar zbioru roboczego – rozmiar często używanej pamięci,
3. Długość transakcji – liczba współdzielonych odczytów na transakcję,

4. Zanieczyszczenie – stosunek współdzielonych zapisów do współdzielonych odczytów,
5. Lokalność czasowa – prawdopodobieństwo powtórzenia adresu przy współdzielonym odczycie,
6. Współzawodnictwo – prawdopodobieństwo konfliktu w transakcji,
7. Przewaga – stosunek współdzielonych odczytów do wszystkich cykli wykonania,
8. Gęstość – stosunek niewspółdzielonych cykli wykonanych poza transakcją do wszystkich niewspółdzielonych cykli.

Połączenie tych charakterystyk opisuje w pełni własności aplikacji w kontekście pamięci transakcyjnej, a ich ortogonalność gwarantuje, że nie wpływają one na siebie wzajemnie i nie są od siebie zależne. Zaimplementowany test wydajności wykorzystuje te charakterystyki do oceny aplikacji i wykrywania patologii pamięci transakcyjnej. Został on z powodzeniem użyty do odtworzenia charakterystyk prezentowanych przez aplikacje wchodzące w ramy pakietu STAMP.

3.1.4 HyFlow

HyFlow [28] to pierwsza implementacja rozproszonej pamięci transakcyjnej. Implementacja umożliwia wykonywanie transakcji w modelu control-flow, data-flow albo hybrydowym, gdzie wybór dokonywany jest heurystycznie na podstawie takich danych jak lokalność, rozmiar obiektu czy charakterystyka dostępów. W ramach implementacji udostępnione są różne polityki zarządzania współzawodnictwem, różne algorytmy kontroli transakcji oraz protokoły wyszukiwania zdalnych obiektów. Zaimplementowano także proste testy w celu ewaluacji rozproszonych pamięci transakcyjnych:

1. Pożyczka – prosta aplikacja do przesyłania pieniędzy – ma na celu ewaluacje pamięci transakcyjnych w modelu control-flow, ponieważ transakcja uzyskująca pożyczkę obejmuje znaczącą liczbę odwołań do zdalnych obiektów.
2. Bank – aplikacja udostępniająca zbiór kont bankowych przydzielonych do różnych banków – połączenie wydajności transakcji pobierającej stan konta i wykonującej przelew pozwala na zmierzenie wydajności modeli sterujących współbieżnością w systemie rozproszonym.
3. Wakacje – rozproszony odpowiednik testu wakacje z pakietu STAMP, udostępnia system rezerwacji lotów – pozwala na ocenę wydajności transakcji wykonujących odczyt i zapis.
4. Mikrotesty – Rozproszone drzewo binarne i rozproszona lista – służy ewaluacji wydajności transakcji drobnoziarnistych.

Testy przeprowadzone przez autorów pokazują, że wydajność rozproszonej pamięci transakcyjnej jest porównywalna do innych rozproszonych sposobów kontroli współbieżności, przy zachowaniu prostszego interfejsu programistycznego.

3.1.5 Lee-TM

Lee-TM [2] to nietrywialny zestaw testów wydajności pamięci transakcyjnej bazujący na algorytmie Lee wykorzystywanym do trasowania obwodów elektronicznych. Algorytm Lee ma wiele cech pożądanych w testach wydajności pamięci transakcyjnej takich jak: wysoka współbieżność, złożona charakterystyka współzawodnictwa oraz szeroki zakres długości transakcji. Zaimplementowano pięć implementacji algorytmu: sekwencyjną, gruboziarniste zamki, średniogruboziarniste zamki, transakcyjną i zoptymalizowaną transakcyjną. Przeprowadzone zostały testy wykonania algorytmu Lee dla złożonych układów elektronicznych, gdzie potencjalną liczbę współbieżnych operacji można liczyć w tysiącach. Zoptymalizowana wersja transakcyjna osiągała w najlepszym wypadku wydajność porównywalną z gruboziarnistą implementacją zamków i dużo mniejszą niż średniogruboziarniste zamki. W przypadku niezoptymalizowanej implementacji z użyciem pamięci transakcyjnej zwrócono uwagę na problem powtarzania transakcji, który powodował wielokrotne wykonywanie tej samej pracy. Wnioskiem z pracy jest konieczność analizy zarządzania współzawodnictwem, ponieważ może to prowadzić do lepszej wydajności spowodowanej powtarzaniem mniejszej liczby transakcji.

3.2 Zastosowania pamięci transakcyjnej

Testy wydajnościowe są w swojej naturze syntetyczne, podczas gdy testowanie wydajności wymaga implementacji i analizy w realnych przypadkach użycia. Ważnym krokiem dla popularyzacji pamięci transakcyjnej jest wykorzystanie jej do stworzenia rzeczywistych aplikacji oraz dodanie niskopoziomowych implementacji pamięci transakcyjnej do powszechnie używanych rozwiązań.

3.2.1 Rzeczywiste aplikacje

Istnieje mało udokumentowanych przypadków wykorzystania pamięci transakcyjnej w rzeczywistych aplikacjach.

QuakeTM [10] to próba implementacji mechanizmów pamięci transakcyjnej w aplikacji będącej rzeczywistym serwerem gry Quake w wersji dla wielu graczy sieciowych. Aplikacja serwerowa będąca punktem wyjścia była wykonywana sekwencyjnie. Zrównoleglono jej wykonywanie w dwóch wersjach – gruboziarnistej, składającej się z 8 sekcji krytycznych oraz drobnoziarnistej – składającej się z 58 sekcji krytycznych oraz porównano jest ze sobą. Pierwszym wnioskiem wynikającym z badań jest znaczący narzut wydajnościowy pamięci transakcyjnej wobec drobnoziarnistych zamków. Sekcje krytyczne wykonywane są 3,5 razy wolniej w przypadku braku rywalizacji. Transakcje w badanych przypadkach często były wielokrotnie powtarzane. Zbadana wydajność pamięci transakcyjnej była gorsza od zamka globalnego.

3.2.2 Implementacje niskopoziomowe

Wraz z rozwojem badań nad pamięcią transakcyjną zainteresowali się nią twórcy popularnych rozwiązań niskopoziomowych. Dodanie obsługi pamięci transakcyjnej na niskim poziomie pozwala na zmniejszenie narzutu wydajnościowego tej metody synchronizacji.

Pierwszym przykładem niskopoziomowej implementacji jest eksperymentalna obsługa pamięci transakcyjnej [4] w kompilatorze powszechnie używanego języka C++ – GCC (od wersji 4.7). Wprowadzone do specyfikacji języka zostały dwa typy transakcji:

1. `transaction_relaxed` – Transakcja, która nie widzi efektów pracy innych transakcji oraz jej niezatwierdzony stan nie jest widoczny dla reszty transakcji. Możliwa jest jednak współpraca z innymi metodami synchronizacji.
2. `transaction_atomic` – Transakcja, która jest w całości wyizolowana od reszty systemu. Do momentu zatwierdzenia nie udostępnia swojego stanu pozostałym wątkom oraz nie widzi zmian przez nie dokonanych. Polecenia w atomowej transakcji są wykonywane w całości albo wycofywane.

Mimo, że implementacja jest wczesną niezoptymalizowaną wersją, z pewnością przyczyni się do zwiększenia zainteresowania pamięcią transakcyjną jako alternatywą dla zamków.

Wprowadzone zostały także rozwiązania sprzętowe wspierające wykonanie pamięci transakcyjnej poprzez udostępnianie niskopoziomowych funkcji na poziomie architektury procesora. W procesorze IBM Power8 zaimplementowane zostało sprzętowe wsparcie dla pamięci transakcyjnej w celu zwiększenia korzyści wydajnościowych z użycia tego procesora [22]. Zawiera ono sprzętowe polecenia rozpoczęcia i zakończenia transakcji, jej zaniechania oraz sprawdzenia stanu. Tak zdefiniowana transakcja nie powiedzie się jeżeli w czasie trwania transakcji inny wątek uzyska dostęp do tych samych miejsc w pamięci.

Kolejnym przykładem niskopoziomowej implementacji jest wsparcie dla pamięci transakcyjnej w procesorach Intela o nazwie kodowej Haswell [9]. Podobnie jak w poprzednim przykładzie udostępniane są sprzętowe polecenia rozpoczęcia, zakończenia i zaniechania transakcji. Udostępniony został też mechanizm pozwalający na zdefiniowanie zastępczego mechanizmu synchronizacji w przypadku uruchamiania aplikacji na procesorze nie posiadającym wsparcia dla pamięci transakcyjnej. Niestety odkryte zostały krytyczne błędy we wprowadzonym mechanizmie powodujące nieprzewidywalne zachowanie systemu [17]. Intel zaleca wyłączenie tego mechanizmu aby zachować stabilność systemu, co niestety negatywnie wpłynie na rozwój aplikacji wykorzystujących sprzętowe wsparcie dla pamięci transakcyjnej.

3.2.3 Podsumowanie

Podsumowując stan wiedzy na temat pamięci transakcyjnej stwierdzono, że nie jest to dziedzina dokładnie zbadana. Brakuje przede wszystkim realnych zastosowań pamięci transakcyjnej, które udowodniłyby możliwość jej użycia. Przyczyniłoby się to do zwiększenia zainteresowania tę alternatywną metodą synchronizacji i w konsekwencji uprościło programowanie systemów współbieżnych i rozproszonych.

Rozdział 4

Atomic RMI

Stworzenie projektu rozproszonego systemu zarządzania odtwarzaniem mediów rozpoczęto od wyboru konkretnej implementacji pamięci transakcyjnej. Dostępne implementacje rozproszonej pamięci transakcyjnej to HyFlow [28], z optymistycznym sterowaniem współbieżnością oraz *Atomic RMI* [30] z pesymistycznym sterowaniem współbieżnością. Podjęto decyzję o użyciu Atomic RMI ponieważ pesymistyczna kontrola współbieżności nie została jeszcze dobrze zbadana w praktyce. Dodatkowo pozwala ona na wykonywanie operacji nieodwracalnych oraz wywoływanie operacji zaniechania i powtórzenia transakcji na żądanie programisty. Możliwości te mogą być przydatne podczas programowania funkcjonalności badanego systemu. Atomic RMI działa w modelu control-flow, czyli transakcje są wykonywane na węzłach, które zawierają współdzielone dane.

Atomic RMI bazuje na mechanizmie Java Remote Method Invocation [25]. Java RMI jest modelem rozproszonych obliczeń uruchamianym przez wirtualną maszynę Javy i opiera się o mechanizm Remote Procedure Call. W odróżnieniu od RPC jest zorientowany obiektowo co oznacza, że wywoływane są metody zdalnych obiektów zamiast zwykłych procedur. Możliwe jest też przekazywanie całych obiektów jako argumenty lub zwracane wartości bez dodatkowego kodu. Podejście obiektowe pozwala na programowanie z użyciem wzorów projektowych, w przejrzysty i łatwy do zrozumienia sposób.

Atomic RMI rozszerza ten mechanizm o możliwość opakowania wywołań metod w transakcje obejmujące wiele obiektów znajdujących się na różnych węzłach. Transakcje pozwalają na zachowanie atomowości i szeregowalności oraz wspierają operacje wycofania oraz powtórzenia. Pesymistyczne blokowanie obiektów pozwala na wykonywanie operacji nieodwracalnych w ramach transakcji ponieważ nie zostaną one powtórzone jeżeli programista nie wymusi wycofania. Dzięki użyciu obiektowego mechanizmu wywoływania metod oraz definiowania transakcji kod nie jest przeplatany poleceniami sterującymi synchronizacją np. komunikatami sieciowymi.

4.1 Supremum Versioning Algorithm

Za kontrolą dostępu do danych stoi algorytm pesymistycznej kontroli współbieżności *Supremum Versioning Algorithm* [30, 36, 37]. Wspierany przez niego mechanizm *wczesnego zwalniania obiektów* pozwala na wyłączny dostęp do obiektów tak długo, jak transakcja je potrzebuje i udostępnienie ich pozostałym transakcjom tak szybko, jak tylko możliwe. Realizacja tego algorytmu wymaga zdefiniowania przez programistę *supremum* odwołań do obiektów w ramach transakcji, czyli górnej granicy wywołań na tym zdalnym obiekcie. Algorytm używa czterech zmiennych dla każdego obiektu wywoływanego w transakcjach tworząc system wersjonowania obiektów. Mają one na celu ustalenie czy polecenie w transakcji powinno być dozwolone, opóźnione lub wycofane:

- `o.gv` – global-version – zlicza ile transakcji, które będą potencjalnie używać obiektu o, rozpoczęło wykonywanie,
- `o.pv[k]` – private-version – zawiera informacje jaką wersję obiektu o może wywoływać transakcja `k`, jest równy wartości `o.gv` w momencie rozpoczęcia transakcji `k`,
- `o.lv` – local-version – oznacza numer wersji obiektu o w ostatniej transakcji, która zwolniła ten obiekt – mówi, że transakcja która może wywołać metodę obiektu o, to taka której wersja jest następna po bieżących `o.lv`,
- `o.ltv` – local-terminated-version – oznacza numer wersji o w ostatniej transakcji, która zwolniła obiekt i została zatwierdzona lub wycofana.

Używane są także dwie zmienne powiązane z ilością wywołań:

- `k.sup[o]` – mapa przechowująca informację o zdefiniowanym supremum wywołań obiektów dla transakcji k,
- `k.cc[o]` – licznik wykonanych wywołań na obiekcie o przez transakcję k.

W momencie rozpoczęcia transakcji k dla każdego obiektu o używanego w transakcji inkrementowany jest `o.gv` co oznacza, że pojawiła się nowa transakcja, która będzie używać tego obiektu. Następnie dla każdego obiektu o prywatne liczniki `o.pv[k]` dla transakcji k są inicjalizowane wartościami odpowiadającymi wersji globalnej `o.gv` (patrz rys. 4.1).

Wywołanie współdzielonego obiektu o przez transakcję `k` jest wykonywane jeżeli prywatna wersja obiektu `o.pv[k]` jest o jeden mniejsza niż lokalna wersja `o.lv`. Oznacza to, że transakcja, która poprzednio zajęła obiekt już go zwolniła. Jeżeli warunek ten jest spełniony transakcja sprawdza jeszcze czy obiekt jest poprawny, tzn. czy transakcja, która zwolniła go poprzednio nie została zaniechana. Gdyby tak się stało to transakcja jest wycofywana (patrz rys. 4.2).

Kolejnym krokiem jest właściwe wywołanie metody na zdalnym obiekcie po którym następuje inkrementacja licznika wywołań obiektu `k.cc[o]`, który zawiera informację o liczbie wywołań metod obiektu. Jeżeli licznik ten jest równy zdefiniowanemu supremum `k.sup[o]` to obiekt jest zwalniany i licznik lokalnej wersji `o.lv` otrzymuje wartość równą prywatnej wersji w transakcji `k` – `o.pv[k]`. Jeżeli w systemie uruchomiona jest inna trans-

akcja l posiadająca `o.pv[l]` o jeden większe od `o.lv` to uzyskuje ona dostęp do tego obiektu.

Po wykonaniu wszystkich operacji transakcja `k` nie zostaje zatwierdzona dopóki dla każdego obiektu o używanego przez nią licznik ostatecznej wersji `o.ltv` nie będzie o jeden mniejszy od `o.pv[k]`. Spełnienie tego warunku oznacza, że wszystkie transakcje, które poprzednio korzystały z obiektu, zostały zatwierdzone (patrz rys. 4.3) lub wycofane (patrz rys. 4.4). Następnie dla każdego obiektu `o` do `o.lv` przypisywana jest wartość o.pv[k] w przypadku jeśli nie osiągnięto supremum wywołań obiektu. Ostatecznie transakcja `k` przypisuje `o.pv[k]` do `o.ltv` co oznacza, że przetwarzanie zostało zakończone.

Jeżeli programista postanowi zwolnić obiekt przed osiągnięciem supremum (opisanym w sekcji 4.2) to transakcja `k` po uzyskaniu dostępu do obiektu `o` przypisuje `o.lv` równe `o.pv[k]` co oznacza że kolejne transakcje mogą korzystać z obiektu o (patrz rys. 4.5).

```
1  for(o : sort(k.sup))
2    o.lock.acquire();
3  for(o : k.sup) {
4    o.gv += 1;
5    o.pv[k] = o.gv;
6  }
7  for(o : sort(k.sup))
8    o.lock.release();
```

Rysunek 4.1: Inicjalizacja

```
1  k.waitUntil(o.pv[k]-1 == o.lv);
2  if (o.isInvalid())   {
3    k.rollback(); return;
4  }
5  o.m(...); // method call
6
7  if (k.cc[o] == k.sup[o])
8    o.lv = o.pv[k];
```

Rysunek 4.2: Wywołanie metody

```
1  for (o : k.sup) {
2    k.waitUntil(o.pv[k]-1 == o.ltv);
3    if (o.isInvalid()){
4    k.rollback(); return; }
5  }
6
7  for (o : k.sup) {
8    if (k.cc[o] < k.sup[o])
9      o.lv = o.pv[k];
10   o.ltv = o.pv[k];
11 }
```

Rysunek 4.3: Zatwierdzenie

```
1  for (o : k.sup) {
2    k.waitUntil(o.pv[k]-1 == o.ltv);
3    o.restore(k);
4    if (k.cc[o] < k.sup[o])
5      o.lv = o.pv[k];
6    o.ltv = o.pv[k];
7  }
```

Rysunek 4.4: Wycofanie

```
1  k.waitUntil(o.pv[k]-1 == o.lv);
2  o.lv = o.pv[k];
```

Rysunek 4.5: Ręczne zwolnienie

4.2 Wczesne zwalnianie obiektów

Kluczowym czynnikiem optymalizującym wykonanie transakcji i wyróżniającym pesymistyczną pamięć transakcyjną na tle innych metod synchronizacji jest wczesne zwalnianie obiektów. Mechanizm ten polega na zwalnianiu obiektu podczas wykonania transakcji w momencie w którym wiadomo, że nie zostanie on więcej użyty w dalszej części kodu. Obiekt jest zwalniany pomimo, że transakcja nie została zatwierdzona i pozostały do wykonania wywołania na innych obiektach. Konieczne do tego jest zdefiniowanie liczby wywołań metod zdalnego obiektu po której wiadomo, że obiekt nie będzie już używany i może zostać zwolniony. W przypadku podania liczby większej niż rzeczywista liczba wywołań obiekt nie zostanie zwolniony do momentu zatwierdzenia lub wycofania transakcji. W takim wypadku ograniczona zostaje współbieżność, ponieważ obiekt mógłby być używany w tym czasie przez inne transakcje i w efekcie uzyskujemy wydajność podobną do zamków. Z kolei podanie mniejszej liczby skutkuje wystąpieniem wyjątku i wycofaniem transakcji. Mogłoby to doprowadzić do jednoczesnego dostępu do danych na skutek zwolnienia obiektu przed jego ostatnim użyciem przez co nie byłoby zachowane wzajemne wykluczanie.

Suprema mogą być wyznaczane przez programistę lub automatycznie za pomocą narzędzi do analizy statycznej kodu. Istnieje możliwość ręcznego zwolnienia obiektu przez programistę jako część logiki programu [30]. Mechanizm ten należy stosować jeżeli nie jest możliwe wyznaczenie liczby wywołań przed rozpoczęciem transakcji.

4.3 Opis techniczny

Kontrola dostępu do zdalnych obiektów Java RMI biorących udział w transakcji odbywa się to poprzez stworzenie proxy dla każdego zdalnego obiektu biorącego udział w transakcji. Proxy udostępnia metody oryginalnego obiektu, ale opakowuje je w instrukcje sprawdzające czy spełnione są warunki dostępu do danych definiowane przez algorytm SVA. W zależności od wyniku algorytmu proxy oczekuje na zwolnienie, wywołuje metodę oryginalnego obiektu lub zaprzestaje dalszego wykonania. Dodatkowo przechowuje on migawkę obiektu na moment rozpoczęcia transakcji.

W systemach rozproszonych należy brać pod uwagę możliwość awarii dowolnego z węzłów dlatego w Atomic RMI obsługiwane są dwa rodzaje awarii – z punktu widzenia transakcji oraz zdalnego obiektu. Podczas wykonania transakcji wykonywane jest standardowe wywołanie zdalnego obiektu poprzez Java RMI, jeżeli zdalny obiekt nie jest dostępny to wykrycie awarii opiera się na wyjątku wyrzucanym poprzez Java RMI i dalsza obsługa jest zależna od programisty, możliwe jest wycofanie całej transakcji lub pominięcie tego wywołania jeżeli nie jest ono krytyczne dla działania systemu. Z drugiej strony węzeł inicjujący transakcję też może ulec awarii. Aby to wykryć zdalne obiekty używają mechanizmu pulsu polegającego na okresowym wysyłaniu zapytania monitorującego stan węzła nadzorującego transakcję. W przypadku braku odpowiedzi obiekt przywraca swój stan do stanu przed rozpoczęciem transakcji, aby nie pozostać w stanie niespójnym z resztą

systemu. Jeżeli wykryta awaria była fałszywie dodatnia może się zdarzyć, że transakcja ponownie skomunikuje się z obiektami, ale spowoduje to konieczność wycofania się transakcji ponieważ `o.lv > o.pv[k] + 1`.

4.4 Użycie

Biblioteka Atomic RMI kładzie nacisk na prostotę użycia oraz jak największą przezroczystość kodu odpowiedzialnego za transakcje, aby nie zaciemniać kodu logiki biznesowej. Na listingu 4.6 zaprezentowano kod wykonujący prostą transakcję symulującą rezerwację stolika w restauracji i przypisanie do niego konkretnej osoby i sprawdzenie czy jest ona z tego stolika zadowolona.

Pierwszym krokiem jest zdefiniowanie nowego obiektu transakcji, na którym wykonywane są dalsze operacje związane z daną transakcją (linia 1). Kolejnym krokiem jest pobranie referencji do zdalnych obiektów poprzez mechanizm rejestru Java RMI (linie 4-5). Następnie należy zdefiniować suprema dostępów do zdalnych obiektów RMI biorących udział w tej transakcji (linie 8-9) oraz pobranie transakcyjnych proxy do tych obiektów – tRestaurant oraz tPerson. Obiekt *restaurant* zostanie wywołany jednokrotnie a obiekt *person* dwukrotnie. Po ustaleniu supremum wywołań możliwe jest rozpoczęcie transakcji (linia 12). Następnym krokiem jest zarezerwowanie stolika poprzez wywołanie metody na proxy obieku restaurant (linia 13) oraz przypisanie osoby do stolika poprez wywołanie metody na proxy obiektu person (linia 15). W zależności od wyniku wywołania istnieje możliwość zatwierdzenia lub wycofania transakcji (linie 19-23). Na przykładzie obiektu restaurant widać jak łatwo w praktyce wykorzystać mechanizm wczesnego zwalniania obiektów. Jeszcze zanim wykonane zostaną metody dla obiektu person, restaurant jest dostępny i umożliwia rezerwacje stolików innym transakcjom.

```
Transaction t = new Transaction();

// Pobranie z rejestru RMI referencji do zdalnego obiektu
Restaurant restaurant = registry.lookup("restaurant");
Customer customer = registry.lookup("customer");

// Pobranie referencji do obiektu proxy
Restaurant tRestaurant = t.accesses(restaurant,1);
Person tPerson = t.accesses(person,2);

t.start(); // Start transakcji
Table table = tRestaurant.reserveTable();
// Obiekt restaurant jest zwolniony
tPerson.sitAtTable(table);
// Pozostalo jeszcze jedno wywolanie obiektu person

if (person.isSatisfied()) {
  transaction.commit(); // Zatwierdzenie transakcji
} else {
  transaction.rollback(); // Wycofanie transakcji
}
```

Rysunek 4.6: Przykładowa transakcja Atomic RMI

Rozdział 5

Projekt Aplikacji Rozproszonej

Ewaluacja zastosowania pamięci transakcyjnej w systemie rozproszonym posiadającym rzeczywiste zastosowanie wymaga zaprojektowania odpowiednio złożonego systemu spełniającego pewne warunki. System rozproszony pozwalający na analizę pamięci transakcyjnej ad minimum powinien charakteryzować się:

1. Występowaniem zasobów współdzielonych przez wielu użytkowników jednocześnie.
2. Koniecznością zachowania spójności systemu przy współbieżnym dostępie do zasobów.

W poprzednim rozdziale omówiona została konkretna implementacja rozproszonej pesymistycznej pamięci transakcyjnej – Atomic RMI. Przy projektowaniu systemu należy wziąć pod uwagę następujące wymagania Atomic RMI:

1. Obiekty współdzielone muszą być obiektami zdalnymi Java RMI,
2. Przed wywołaniem transakcji konieczna jest znajomość wykorzystywanych obiektów oraz liczba wywołań ich metod,
3. System spełnia model control-flow.

5.1 Aplikacja rozproszona – omówienie problemu

Zaproponowano *Atomic Café – rozproszoną szafę grającą* jako przykład aplikacji będącej systememem rozproszonym spełniającym powyższe warunki. Główne założenia rozproszonej szafy grającej w odniesieniu do powyższych wymagań są następujące:

1. System umożliwia tworzenie rozproszonej kolekcji muzyki i jej odtwarzanie.
2. Każdy węzeł systemu posiada zasoby współdzielone i wykorzystywane przez pozostałe węzły.
3. Z systemu może korzystać wielu klientów jednocześnie, którzy wykonują operacje współbieżnie i oczekują, że system zachowa spójność.

Szczegółowe zdefiniowanie założeń systemu wymaga rozpatrzenia możliwych przykładów użycia rozproszonej szafy grającej:

1. Grupa n studentów wynajmująca mieszkanie posiadające wspólny salon, w którym znajduje się wysokiej klasy sprzęt grający podłączony do komputera stacjonarnego. Każdy z nich ma swój własny komputer przenośny z dostępem do lokalnej sieci oraz zbiór muzyki w postaci plików znajdujących się na jego dysku. Studenci siedząc razem w salonie chcieliby mieć możliwość odtwarzania utworów z dowolnego z urządzeń.
2. Właściciel kawiarni, w której odtwarzana jest muzyka korzysta ze stworzonej przez siebie bazy utworów. Chciałby jednak pozwolić swoim klientom mieć wpływ na odtwarzaną muzykę. Na każdym z kilku pięter kawiarni znajduje się osobny komputer sterujący sprzętem grającym, właściciel chciałby, aby każdy z nich odtwarzał to samo. Kawiarnia mieści wielu gości, posiada dostęp do internetu i udostępnia bezpłatny hotspot dla klientów, którzy często pracują w niej przy użyciu swoich komputerów przenośnych.

Rozważony zostanie system rozproszony odpowiadający potrzebom obu tych przypadków z czego wynikają następujące wymagania:

1. Aplikacja uruchamiana jest na komputerach wszystkich zainteresowanych odtwarzaniem muzyki oraz na komputerach podłączonych pod dedykowane urządzenia do odtwarzania audio.
2. Aplikacja udostępnia wspólną dla wszystkich użytkowników bazę muzyki:
   (a) każdy użytkownik ma równe prawo do przeglądania, dodawania i usuwania utworów do bazy muzyki,
   (b) możliwy jest jednoczesny dostęp wszystkich użytkowników do bazy muzyki i współbieżne wykonywanie operacji na niej.
3. Aplikacja udostępnia możliwość odtwarzania utworów z bazy muzyki:
   (a) utwory powinny być odtwarzane według zdefiniowanej przez listę odtwarzania kolejności,
   (b) każdy użytkownik ma równe prawo do dodawania swoich utworów do listy odtwarzania,
   (c) operacje dodawania, usuwania i przeglądania listy są wykonywane współbieżnie przez użytkowników.

W konsekwencji wymagań system składa się z dwóch typów aplikacji – serwerów oraz klientów. Architektura poszczególnych części została przedstawiona w sekcji 5.2

- Serwer kontroluje odtwarzacz muzyczny, jego kolejkę odtwarzania oraz zbiera utwory muzyczne od klientów.
- Klient to użytkownik systemu, którego celem jest możliwość odtwarzania ulubionej muzyki na współdzielonym odtwarzaczu muzycznym.

Założeniem było uzyskanie oprogramowania rozwiązującego przykładowe problemy studentów oraz właściciela kawiarni w następujący sposób:

1. Grupa studentów – 1 serwer, n klientów

   - Jeden ze studentów instaluje aplikację serwerową na komputerze stacjonarnym podłączonym do sprzętu grającego.
   - Każdy ze studentów po przyjściu z zajęć uruchamia aplikację kliencką na swoim laptopie. Następnie dodaje do bazy muzyki utwory, które ma ochotę tego dnia posłuchać. Po przejrzeniu dostępnych utworów i listy odtwarzania kolejkuje swoje utwory.

2. Kawiarnia – n homogenicznych serwerów, m klientów

   - Właściciel kawiarni instaluje aplikacje serwerowe na wszystkich urządzeniach, które są podłączone do głośników w kawiarni. Dodaje do ich bazy muzyki utwory wpływające na określony nastrój kawiarni i układa z nich kolejkę odtwarzania.
   - Klient przychodzi do kawiarni, a następnie łączy się z dostępnym hotspotem WiFi używając swojego laptopa. Uruchamia aplikację kliencką, co pozwala mu przejrzeć listę aktualnie odtwarzanych utworów i zmodyfikować ją wedle swojego uznania dodając np. nowe utwory do bazy muzyki dostępnej w kawiarni. Właściciel kawiarni nie musi dzięki temu być na bieżąco z trendami muzycznymi, a klienci mają pełen wpływ muzykę odtwarzaną w kawiarni.

Rozwiązanie – uruchamiają oni aplikację kliencką na swoich komputerach oraz aplikację serwerową na sprzęcie grającym. Otrzymują dzięki temu możliwość kolejkowania utworów pochodzących z osobistej biblioteki muzyki i odtwarzania ich na wspólnym sprzęcie grającym.

Aplikacja użytkowana przez wielu użykowników, którzy korzystają ze wspólnych zasobów jednocześnie i nieprzewidywalnie, wymaga użycia mechanizmów synchronizacji. W kolejnych sekcjach opisany zostanie system oraz problemy w nim występujące. Analiza przeprowadzona jest pod kątem wykorzystania pesymistycznej pamięci transakcyjnej, a dokładnie omawianego w rozdziale 4 Atomic RMI.

5.2 Idea systemu

Biorąc pod uwagę wymagania i ograniczenia opisane w sekcji 5.1 zaproponowano następujących aktorów i ich funkcjonalności:

1. jako serwer chcę:

   (a) połączyć się z innymi węzłami serwerowymi,
   (b) udostępniać klientom możliwość podłączenia do systemu,
   (c) przechowywać utwory dodane do bazy muzyki oraz kolejkę odtwarzania,

(d) widzieć listę podłączonych klientów, listę utworów w bazie muzyki, kolejkę odtwarzania i aktualnie odtwarzany utwór,

(e) odtwarzać muzykę na podstawie ustawionej listy odtwarzania,

2. jako klient chcę:

(a) podłączyć się do systemu poprzez dowolny z serwerów,

(b) widzieć listę podłączonych klientów, listę utworów w bazie muzyki, kolejkę odtwarzania i aktualnie odtwarzany utwór,

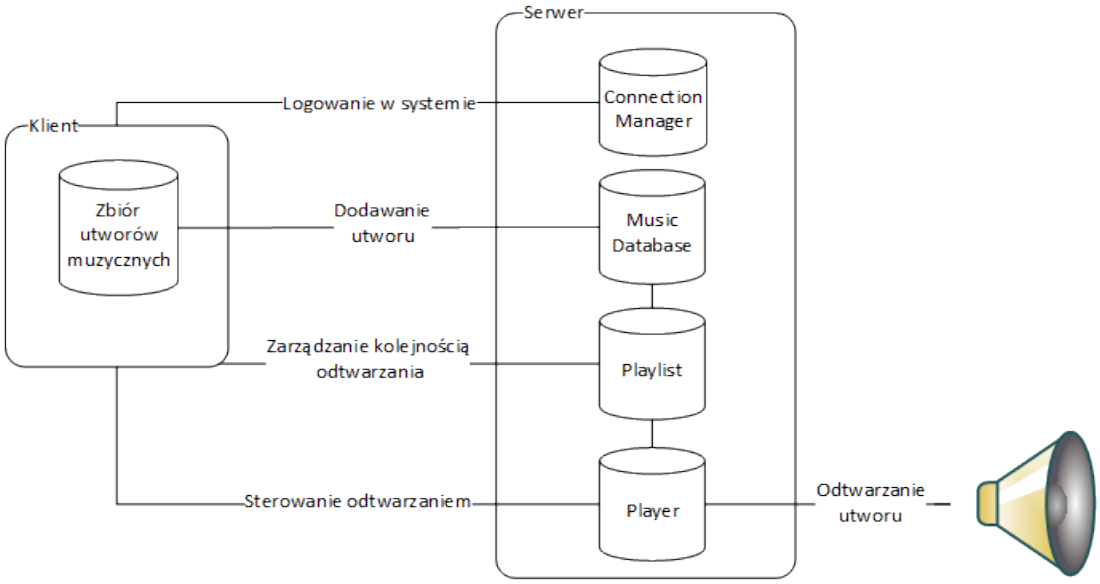
(c) dodawać utwory ze swojego dysku do bazy muzyki,

(d) usuwać utwory z bazy muzyki,

(e) dodawać i usuwać utwory do kolejki odtwarzania oraz zmieniać ich kolejność,

(f) sterować odtwarzaniem muzyki na weźle z funkcją odtwarzania,

System dzieli się na aplikacje serwerowe oraz klienckie opisane w kolejnych sekcjach. Każdy węzeł systemu zawiera w sobie węzeł rozproszonego rejestru obiektów opisanego w sekcji 5.4. Rys. 5.1 przedstawia zarys architektury systemu, który dokładniej zostanie omówiony w kolejnych sekcjach.



Rysunek 5.1: Architektura rozproszonej szafy grającej

5.2.1 Aplikacja serwerowa

Podstawowym zadaniem aplikacji serwerowej jest przechowywanie i rozsyłanie muzyki oraz jej odtwarzanie. Dodatkowo służą one klientom jako interfejs udostępniający możliwość wykonywania operacji w systemie. Aplikacje serwerowe uruchamiane są na komputerach, które spełniają następujące wymagania:

1. dostępne zasoby dyskowe pozwalające na przechowywanie dużej liczby utworów,

2. są podłączone do sprzętu grającego,
3. są połączone do sieci lokalnej.

W systemie może istnieć dowolna liczba węzłów serwerowych, które powinny ze sobą współpracować tworząc jeden spójny system rozproszony. Pierwszy uruchomiony serwer służy za punkt dostępu dla kolejnych, każdy nowy serwer podłącza się do jednego z już działających serwerów.

Aby zrealizować zdefiniowane wymagania odnośnie funkcjonalności systemu każdy z węzłów serwerowych przechowuje następujące informacje i synchronizuje je z pozostałymi:

1. listę aktywnych węzłów serwerowych,
2. listę aktywnych użytkowników w systemie,
3. utwory dostępne w lokalnej bazie muzyki,
4. stan odtwarzacza i aktualnie ustawiony utwór,
5. kolejkę odtwarzania i przypisane do niej odtwarzacze.

Serwer udostępnia następujące obiekty zdalne:

- `ConnectionManager` – obiekt umożliwiający rejestrowanie się użytkowników w systemie, spełnia następujące funkcje:
  1. przechowuje informacje o zalogowanych użytkowniach w postaci obiektów typu Client,
  2. udostępnia metody zdalne:
     (a) `connect(Client client)` – rejestruje konkretnego użytkownika w systemie,
     (b) `disconnect(Client client)` – wylogowuje konkretnego użytkownika z systemu.

1. `MusicDatabase` – obiekt będący częścią bazy muzyki (patrz 5.3.1), spełnia następujące funkcje:
   (a) zawiera utwory muzyczne w postaci obiektów implementujących interfejs `Playable`, czyli posiadających metody umożliwiające np. odtwarzanie,
   (b) udostępnia metody zdalne:
      i. `addTrack(Playable track)` – dodaje do bazy muzyki informacje o konkretnym utworze (nie wysyła samej treści utworu),
      ii. `removeTrack(Playable track)` – usuwa z bazy muzyki konkretny utwór,
      iii. `checkIfTrackExists(Playable track)` – zwraca informację czy dany utwór znajduje się w bazie,
      iv. `sendFile(RemoteInputStream ristream,String fileName)` – wysyła do zbioru utworów bazy muzyki konkretny plik.

- `Player` – odtwarzacz zarządzający odtwarzaniem, spełnia następujące funkcje:
  – przechowuje informacje o stanie odtwarzacza (stałe: `PLAY`, `STOP`, `PAUSE`, `NEXT`)

oraz aktualnie ustawiony utwór typu `Playable`,
  - udostępnia metody zdalne:
    * `setState(PLAYER_STATES state)` – ustawia status odtwarzacza,
    * `setPlaylist(Playlist playlist)` – ustawia kolejkę utworów, z której będzie pobierać odtwarzacz,
    * `setTrack(Playable track)` – ustawia konkretny utwór na odtwarzaczu.

- `Playlist` – obiekt przechowujący kolejkę utworów do odtworzenia, spełnia następujące funkcje:
  - `queueTrack(Playable track)` – dodaje utwór do kolejki odtwarzania,
  - `removeTrack(Playable track)` – usuwa utwór z kolejki odtwarzania,
  - `moveTrack(int fromTrack, int index)` – przesuwa utwór w kolejce odtwarzania,
  - `getTrack()` – zwraca pierwszy utwór z kolejki odtwarzania,
  - `getTracklist()` – zwraca listę wszystkich utworów w kolejce odtwarzania,
  - `addPlayer(IPlayer player)` – dodaje nowy odtwarzacz do grupy (patrz 5.3.3),
  - `playTrack()` – odtwarza pierwszy utwór z kolejki na wszystkich odtwarzaczach z grupy,

Wykorzystanie tych obiektów w praktyce zostało opisane szczegółowo w sekcji 5.3.

5.2.2 Aplikacja kliencka

Aplikacja kliencka jest uruchamiana na komputerach, które posiadają bibliotekę utworów muzycznych używanych lokalnie przez użytkownika. Aplikacja ta służy do sterowania muzyką odtwarzaną przez węzły serwerowe oraz do zarządzania bazą utworów. Łączy się ona z dowolnym z węzłów serwerowych dostępnych w systemie, a następnie otrzymuje informację o utworach dostępnych w bazie muzyki oraz listę węzłów posiadających kolejkę odtwarzania, czyli umozliwiających sterowanie odtwarzaniem muzyki. Klient może wybrać muzykę ze swojej biblioteki i dodać ją do bazy muzyki co jest równoznacze z poinformowaniem wszystkich klientów i serwerów o nowym dostępnym utworze. Posiada on też możliwość ściągnięcia do swojej prywatnej biblioteki plików muzycznych pochodzących z bazy muzyki. Dzięki temu system może służyć jako platforma wymiany utworów muzycznych, nawet jeżeli użytkownicy nie chcą w tym momencie odtwarzać żadnej muzyki.

Klient wykorzystuje udostępnione przez serwery obiekty zdalne zdefiniowane w 5.2.1. Ponieważ w systemie może istnieć wielu klientów jednocześnie, którzy wykonują operacje współbieżnie wymagana jest analiza możliwych problemów implementacyjnych i synchronizacyjnych.

5.3 Zdefiniowanie problemów

Poniżej przedstawiono problemy związane z synchronizacją i efektywością przetwarzania jakie występują w proponowanej aplikacji:

1. Dodawanie i odtwarzanie utworów pochodzących z wielu węzłów,
2. Przesyłanie plików muzycznych,
3. Zarządzanie odtwarzaniem i kolejką odtwarzania.

5.3.1 Dodawanie i odtwarzanie utworów pochodzących z wielu węzłów

Problematyka związana ze współbieżną obsługą utworów muzycznych jest kluczową częścią systemu, a jej źródłem są użytkownicy dołączający się do systemu i dodający utwory poprzez wysłanie do bazy muzyki. Po przesłaniu utworu do serwera, z którym połączony jest klient każdy węzeł musi mieć możliwość odwołania się do tego utworu. Rozważono tutaj trzy możliwe rozwiązania odnośnie przechowywania utworów w bazie muzyki. W każdym z nich początkowo klient wysyła utwór do swojego serwera.

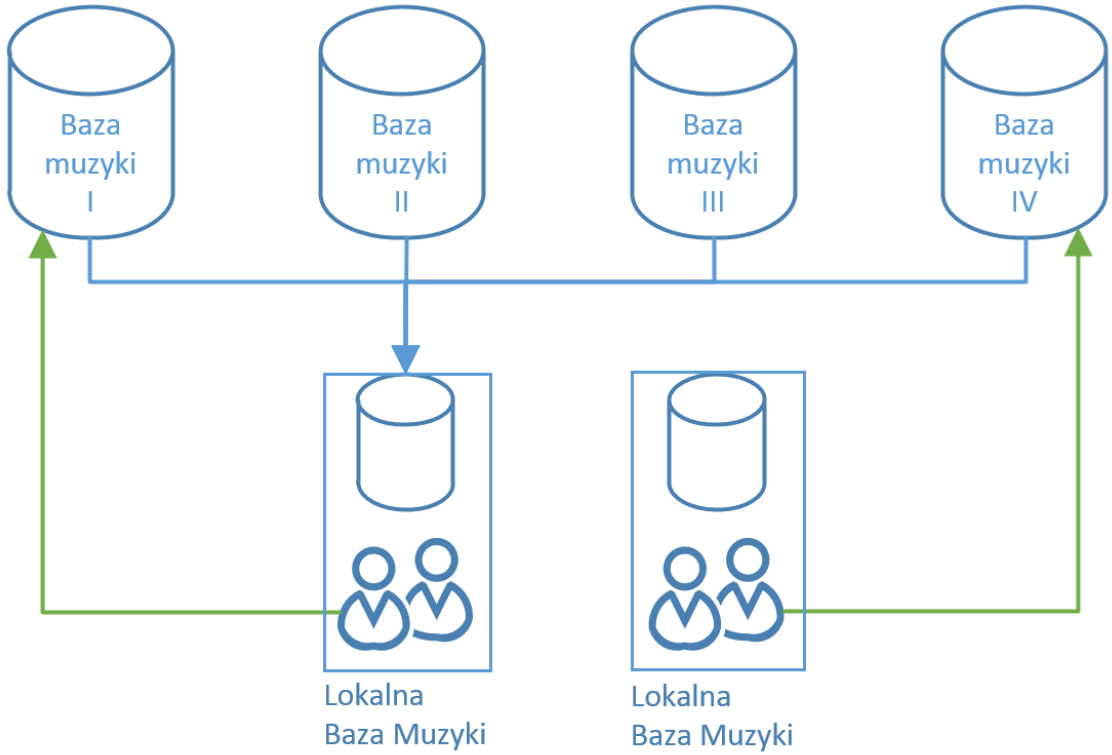
1. Po otrzymaniu utworu serwer rozsyła go do swoich pozostałych klientów oraz innych serwerów, które również przesyłają to do swoich klientów. Dzięki temu każdy węzeł systemu posiada całkowity zbiór utworów w nim dostępnych. Zaletą tego rozwiązania jest natychmiastowy dostęp do dowolnego utworu, ponieważ nie ma potrzeby ściągania plików przez sieć. Do wad należy zaliczyć wysokie zużycie miejsca na dysku wszystkich węzłów oraz niepotrzebny ruch sieciowy, ponieważ część węzłów może nie używać większości plików, które otrzymała przez sieć.
2. Po otrzymaniu utworu serwer rozsyła go do pozostałych serwerów. Przypadek ten jest zoptymalizowaną wersją rozwiązania 1, z wyjątkiem tego, iż serwery posiadają wszystkie pliki muzyczne. Ponieważ każdy z nich może mieć wielu klientów, którzy pobierają z niego pliki, to uzasadnione jest przechowywanie ich pomimo zużycia miejsca.
3. Dodany utwór jest wysyłany do serwera i tylko tam przechowywany. Przy pierwszym żądaniu odtworzenia danego pliku jest on ściągnięty na serwer i zachowywany w celu późniejszego wykorzystania. Dzięki temu obciążenie sieci oraz dysku jest optymalne lecz odbywa się to kosztem zwiększonego czasu potrzebnego na odtworzenie pliku. Oczekiwanie to wynika z konieczności wcześniejszego pobrania go.

Wybrano rozwiązanie 3 ponieważ zużywa ono najmniej zasobów systemu oraz odzwierciedla powszechne użycie strumieniowania przy przesyłaniu muzyki. Spowodowane przez to problemy mogą zostać rozwiązane przez dobre przemyślenie użycia pamięci transakcyjnej. Wynika z tego następująca architektura struktur przechowujących muzykę i odpowiednie procedury ich obsługi. Zdefiniowano elementy systemu przechowywania muzyki:

- Utwór – utwór muzyczny dodany do systemu przez użytkownika,

- Tracklista – całkowita lista utworów dostępnych w systemie,
- Zdalna baza muzyki – informacje o utworach przechowywanych na danym węźle serwerowym,
- Lokalna baza muzyki – informacje o wszystkich utworach przechowanych w systemie zebrane przez węzeł kliencki.

Dodanie utworu do systemu jest wykonywane przez węzeł kliencki połączony z jednym z węzłów serwerowych. Plik z muzyką jest wysyłany do zdalnej bazy muzyki znajdującej się na tym węźle, a po zakończeniu przesyłu informacje o utworze są wpisywane do bazy muzyki. Od tego czasu utwór jest dostępny w systemie dla wszystkich jego uczestników. Klient wysyła zawsze utwory do tej samej bazy muzyki. Nie zmienia się to w trakcie działania programu. Możliwe byłoby wprowadzenie mechanizmów równoważenia obciążenia i przełączanie się użytkowników pomiędzy najwydajniejszymi i najzasobniejszymi bazami danych, ale jest to poza zakresem prezentowanej pracy.



Rysunek 5.2: Pobieranie listy dostępnych utworów ze zdalnych baz muzyki

Efektem takiego rozwiązania jest rozproszenie utworów w systemie pomiędzy wiele węzłów. Aby uzyskać informację o globalnym stanie zawartości bazy muzyki należy odpytać każdą z nich. Uzyskanie spójnego stanu globalnego baz muzycznych wymaga zastosowania synchronizacji. Zdefiniowano następujące rozwiązania problemu synchronizacji poprzez użycie transakcji rozproszonych:

1. Odpytanie każdej z baz muzyki w osobnej transakcji,

2. Odpytanie wszystkich baz muzyki w jednej transakcji.

Rozważono działanie obu rozwiązań na przykładzie dwóch węzłów, które jednocześnie chcą pobrać listę utworów z trzech baz muzyki. Dla uproszczenia analizy przyjęto system homogeniczny, gdzie węzły wykonują operacje z tą samą szybkością, odpytanie każdej z bazy muzyki zajmuje jedną jednostkę czasu i odpytywanie baz następuje zawsze w tej samej kolejności.

Na rys. 5.3 przedstawiono pierwsze rozwiązanie. Gdy pierwszy węzeł rozpoczyna odpytywanie pierwszej bazy muzyki drugi musi poczekać na zwolnienie jej obiektu i wstrzymuje przetwarzanie. Po zakończeniu T_1 przez W_1, W_2 uzyskuje dostęp do zwolnionego obiektu i dalej przetwarzanie obu węzłów następuje bez oczekiwania na zwolnienie obiektów, czyli w sposób optymalny. Minusem tego rozwiązania jest brak uzyskania spójnego stanu systemu. Pobranie listy utworów z każdej z baz zachodzi w osobnej transakcji i w związku z tym cała procedura może być przeplatana operacjami dodania lub usunięcia utworu wykonywanymi przez inne węzły. Z tego powodu żytkownik mógłby otrzymać informacje o stanie systemu, który nie zawiera np. pełnej dyskografii artysty mimo, że w rzeczywistości operacje nie spowodowały niekompletności kolekcji.

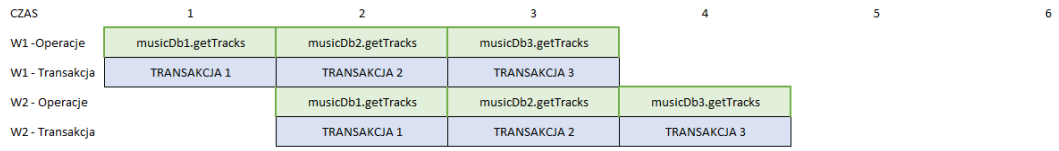
Rys. 5.4 prezentuje drugie rozwiązanie. Pobranie listy utworów zachodzi na każdym węźle w jednej transakcji co powoduje założenie blokady na wszystkich obiektach i pobranie spójnej listy utworów. Blokada wszystkich obiektów powoduje jednak wykonanie operacji sekwencyjnie.

Zwiększenie współbieżności jest możliwe przy wykorzystaniu mechanizmu wczesnego zwalniania obiektów udostępnianego przez Atomic RMI. Rozwiązanie z użyciem tego mechanizmu jest zaprezentowane na rys. 5.5. Transakcja wykonywana przez W_1 po odpytaniu każdego z obiektów zwalnia go i udostępnia transakcji wykonywanej na W_2. Dzięki wczesnemu zwalnianiu obiektów objętych transakcją uzyskano efektywne przetwarzanie współbieżne przy zachowaniu pełnej spójności pobranych informacji.

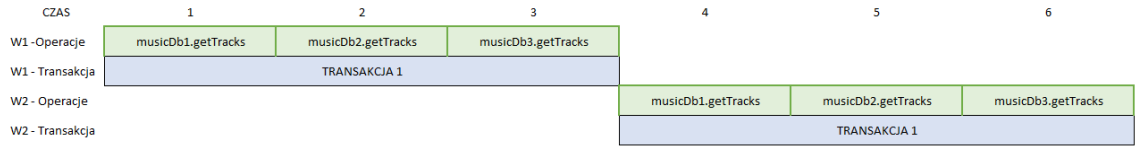
Na rys. 5.6 rozważono sytuację, w której węzły nie wywołują baz muzyki w tej samej kolejności. W pesymistycznym przypadku efektem tego może być zdecydowanie zmniejszenie efektywności przetwarzania. Rozważono skrajny przykład, gdzie dwa węzły odpytują bazy muzyki w odwrotnej kolejności. W rezultacie otrzymane rozwiązanie nie wykorzystuje w żaden sposób mechanizmu wczesnego zwalniania obiektów. Uzyskanie stałego porządku odpytywania baz muzyki na wszystkich węzłach jest proste do wykonania, ponieważ każda baza muzyki posiada własną liczbę porządkową znaną wszystkim węzłom. Dzięki wykorzystaniu zachowania porządku odpytywania baz muzyki uzyskano znaczący wzrost współbieżności.

5.3.2 Przesyłanie plików przez sieć

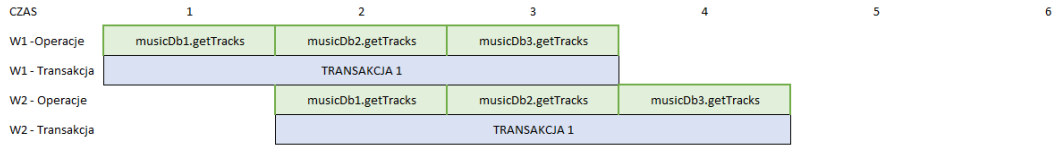
Aplikacja nie posiada stałej, predefiniowanej bazy muzyki dostępnej w systemie. Każdy z utworów musi zostać dodany przez któregoś z klientów poprzez funkcję wysyłania plików muzycznych. W świecie muzyki często zdarza się, że pewien utwór czy album uzyskuje znaczącą popularność. Istnieje możliwość, że więcej niż jeden użytkownik w danym mo-



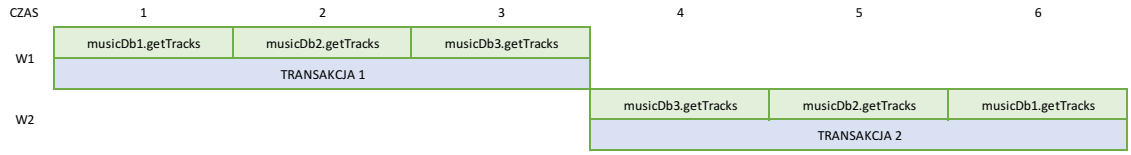
Rysunek 5.3: Przebieg czasowy odpytania baz muzyki w osobnych transakcjach



Rysunek 5.4: Przebieg czasowy odpytania baz muzyki w pojedynczych transakcjach bez wczesnego zwalniania obiektów



Rysunek 5.5: Przebieg czasowy odpytania baz muzyki w pojedynczych transakcjach z użyciem wczesnego zwalniania obiektów oraz z zachowaniem porządku odpytywania baz muzyki



Rysunek 5.6: Przebieg czasowy pesymistycznego przypadku odpytania baz muzyki w pojedynczych transakcjach z użyciem wczesnego zwalniania obiektów bez zachowania porządku odpytywania baz muzyki

mencie postanowi dodać do bazy ten sam utwór.

Procedura dodania utworu jest następująca:

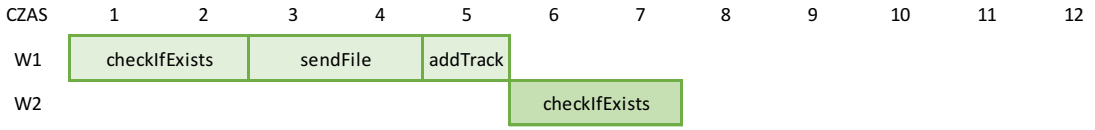
1. Sprawdzić czy w bazie muzyki nie ma już takiego utworu – `musicDatabase.checkIfTrackExists(track)`,
2. Wykonać upload pliku – `musicDatabase.sendFile(file)`,
3. Dodać utwór do bazy muzyki jeżeli wysyłanie się powiodło – `musicDatabase.addTrack(track)`.

Wszystkie te operacje są wykonywane na zdalnym obiekcie `musicDatabase` reprezentującym bazę muzyki, z którą połączony jest klient.

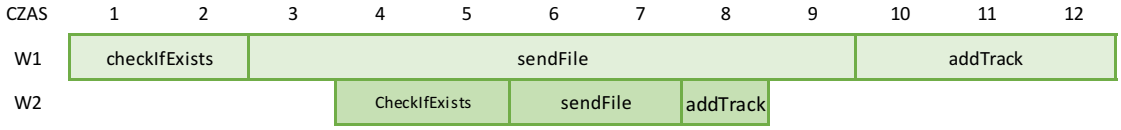
Pliki muzyczne są plikami o różnym rozmiarze, pojedyncze utwory mają wielkość rzędu kilka megabajtów, podczas gdy nagrania całych koncertów mogą zajmować setki megabajtów. Czas przesyłania pliku z utworem przez sieć może być liczony w godzinach. Istnieje duża szansa, że przesyłanie zostanie przerwane na skutek awarii komunikacji sieciowej i należy to wziąć pod uwagę, aby zachować niezawodność i spójność systemu. Podczas projektowania aplikacji rozważono różne przeploty operacji, w przypadku wysyłania plików przez wiele węzłów jednocześnie. Przedstawiono je na poniższych rysunkach. W_1 i W_2 to węzły klienckie wykonujące odpowiednie operacje na tych samych obiektach zdalnych pojedynczej bazy muzyki i dodające ten sam utwór.

1. Sytuacja przedstawiona na rys. 5.7: W_1 wykonuje po kolei wszystkie kroki wymagane do dodania pliku. W_2 wykonuje sprawdzenie, czy taki plik już istnieje dopiero po zapisie utworu w bazie muzycznej przez W_1 i otrzymuje odpowiedź twierdzącą. Sytuacja ta nie wymaga synchronizacji, operacje nie są wykonywane współbieżnie i stan systemu będzie spójny.
2. Sytuacja przedstawiona na rys. 5.8: W_1 rozpoczyna przesyłanie pliku do bazy muzyki, w tym czasie W_2 wykonuje sprawdzenie czy utwór istnieje w bazie. W_1 nie dokonał jeszcze zapisu, więc W_2 uzyskuje informację, że takiego utworu nie ma i również rozpoczyna przesyłanie. Przesyłanie wykonywane przez W_2 kończy się dużo wcześniej niż W_1 i dokonywany jest zapis. W_1 po zakończeniu przesyłania próbuje również wykonać zapis już istniejącego pliku. Nieprawidłowości, które wystąpią z powodu tego zdarzenia są zależne od szczegółów implementacji. Sytuacja taka może zdarzyć się, gdy jeden z węzłów posiada dużo szybsze połączenie sieciowe od drugiego i efektem jest podwojenie ruchu sieciowego oraz operacji zapisu.
3. Sytuacja przedstawiona na rys. 5.9: W_1 oraz W_2 wykonują współbieżne sprawdzenie czy plik istnieje w bazie po czym przystępują do jednoczesnego uploadu. Po zakończonym przesyłaniu plików oba węzły zlecają jednoczesny zapis, który powoduje powstanie dwóch jednakowych utworów w bazie muzyki.

Wyżej pokazane przeploty dowodzą potrzeby zastosowania synchronizacji. Synchronizacja w przypadku tego problemu zachodzi na jednym obiekcie zdalnym musicDB Jest on jedynym obiektem biorącym udział w tej transakcji, więc nie jest możliwe wykorzystanie



Rysunek 5.7: Przeplot operacji dodania utworu przez dwa węzły – pierwszy węzeł dodaje utwór



Rysunek 5.8: Przeplot operacji dodania utworu przez dwa węzły – drugi węzeł dodaje utwór

mechanizmu wczesnego zwalniania obiektów podczas wykonywania transakcji.

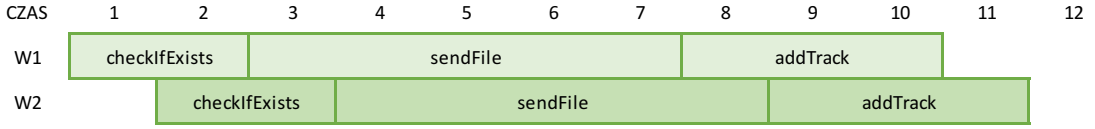
Po analizie potrzeb synchronizacyjnych wyłoniono dwa możliwe sposoby użycia transakcji do synchronizacji wysyłania plików:

1. Transakcja gruboziarnista – objęcie pojedynczą transakcją całej procedury wysłania pliku (patrz listing 5.10),
2. Transakcje drobnoziarniste – podzielenie procedury na dwie osobne transakcje (patrz listing 5.11).

Rozwiązanie wykorzystujące transakcje gruboziarnistą pokazane jest na rys. 5.12. W efekcie zastosowania pojedynczej transakcji osiągnięto poziom współbieżności porównywalny z zastosowaniem zamka na całej operacji. Negatywnym efektem tego rozwiązania jest brak możliwości jednoczesnego wysyłania do bazy muzyki przez więcej niż jeden węzeł. Zaletą jest zachowanie spójności i możliwość wycofywania transakcji w przypadku wystąpienia nieprawidłowości. Dodatkowo w tej sytuacji minimalizowany jest ruch sieciowy ponieważ nie dochodzi do sytuacji, w której utwór byłby wysyłany jednocześnie przez wielu klientów.

W opisywanym podejściu widać, że nie zawsze udaje się wykorzystać mechanizm wczesnego zwalniania obiektów pesymistycznej pamięci transakcyjnej aby zoptymalizować dany proces. Zaletą jest jednak możliwość wycofania transakcji w przypadku wystąpienia awarii przywrócenia wcześniejszego stanu.

Rozwiązanie wykorzystujące transakcje drobnoziarniste przedstawione na rys. 5.13, rys. 5.14 i rys. 5.15, jest pewnym kompromisem pomiędzy spójnością a wydajnością



Rysunek 5.9: Przeplot operacji dodania utworu przez dwa węzły – węzły dodają utwór jednocześnie

```
transaction.begin();

//Sprawdzenie czy utwór występuje
musicDb.checkIfExists(track);

//Wysłanie pliku do bazy muzyki:
musicDb.sendFile(track);

//Dodanie utworu:
musicDb.addTrack();

transaction.commit();
```

Rysunek 5.10: Przebieg wykonania transakcji gruboziarnistej dodającej utwór do bazy muzyki

```
transaction.begin();
//Sprawdzenie czy utwór występuje
musicDb.checkIfExists(track);
transaction.commit();

//Wysłanie pliku do bazy muzyki:
musicDb.sendFile(track);

transaction.begin();
//Dodanie utworu
musicDb.addTrack();
transaction.commit();
```

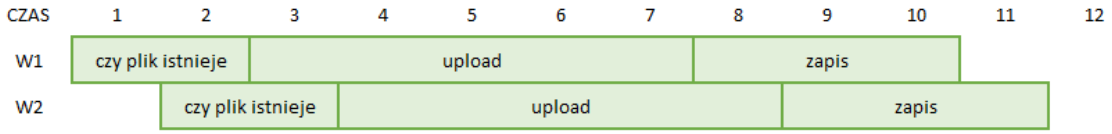
Rysunek 5.11: Przebieg wykonania transakcji drobnoziarnistych dodających utwór do bazy muzyki

podczas wysyłki plików. Znaczącą różnicą pomiędzy rozwiązaniem drobnoziarnistym a gruboziarnistym jest wyjęcie przesyłania pliku spod transakcji.

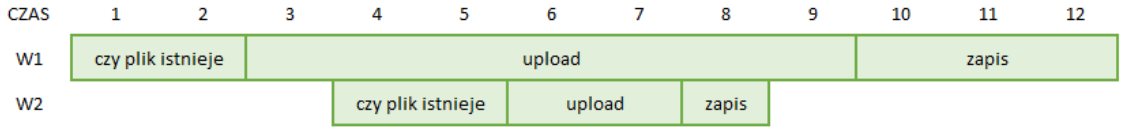
Przesyłanie pliku charakteryzuje się relatywnie długim czasem wykonywania, co zwiększa prawdopodobieństwo wystąpienia problemów sieciowych w trakcie jego trwania. Jednoczesne przesyłanie tego samego pliku do bazy muzyki za pomocą metody `sendFile` nie powoduje powstania niespójności. Celem operacji jest wyłącznie pojawienie się pliku w zasobach dyskowych bazy muzyki i jego źródło nie ma znaczenia. Utwór jest dodawany do bazy dopiero poprzez wykonanie na zdalnym obiekcie metody `addTrack`, która przypisuje do danego utworu konkretny zasób na dysku. Jeżeli metoda ta nie zostanie wykonana, utwór nie pojawia się w bazie muzyki mimo, że plik z utworem został przesłany. Mankamentem tego rozwiązania jest pozostawienie pliku w bazie muzyki w sytuacji, gdy wystąpiła awaria podczas operacji `addTrack`. Po wykryciu takiej awarii możliwe jest usunięcie dodawanego pliku podczas wycofywania transakcji.

Rys. 5.13 przedstawia sytuację w której procesy W_1 i W_2 chcą jednocześnie wysłać ten sam utwór do bazy muzyki. W_1 jako pierwszy uzyskuje dostęp do bazy muzyki. Otrzymuje następnie informacje, że ten utwór nie istnieje, po czym rozpoczyna wysyłanie pliku. W_2 również otrzymuje informacje, że utwór nie istnieje w bazie i rozpoczyna wysyłkę tego samego pliku. W_1 jako pierwszy kończy wysyłanie, uzyskuje w transakcji dostęp do bazy muzyki i dodaje do niej utwór. W_2 po zakończeniu wysyłania pliku uzyskuje dostęp do bazy muzyki i wykonuje na niej zdalną metodę dodającą utwór. Otrzymuje jednak informacje, że utwór ten został już dodany do bazy muzyki. Procedura została wykonana poprawnie ale kosztem jest dwukrotne przesyłanie tego samego pliku przez sieć.

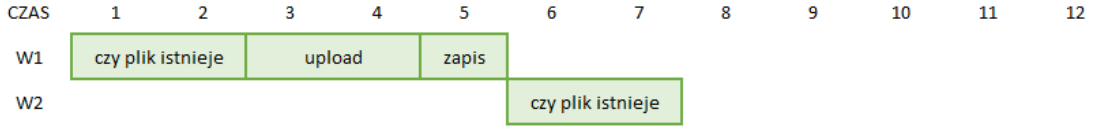
Rys. 5.14 również przedstawia sytuację w której W_1 i W_2 chcą jednocześnie wysłać ten sam utwór. Sprawdzają po kolei czy ten utwór już istnieje w bazie i rozpoczynają przesyłanie pliku. Prędkość, z którą wysyłają pliki różni sie znacząco i w tym przypadku W_2 kończy wysyłanie wcześniej mimo, że W_1 zaczął pierwszy. W_2 dodaje utwór do bazy muzyki, a W_1 otrzymuje podczas dodawania do bazy informację, że utwór już istnieje. W tej sytuacji również procedura została wykonana poprawnie i plik był dwukrotnie przesłany.



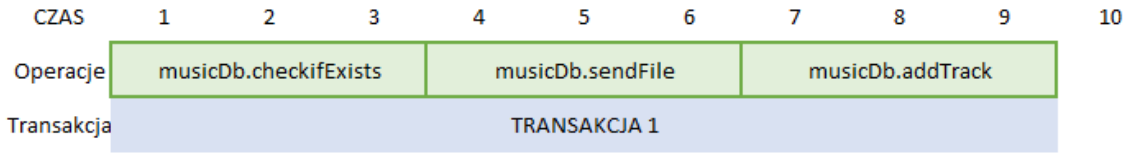
Rysunek 5.12: Operacja dodania utworu za pomocą transakcji gruboziarnistej



Rysunek 5.13: Operacja dodania utworu za pomocą transakcji drobnoziarnistych – W_1dodaje utwór



Rysunek 5.14: Operacja dodania utworu za pomocą transakcji drobnoziarnistych – W_2 dodaje utwór



Rysunek 5.15: Operacja dodania utworu za pomocą transakcji drobnoziarnistych – awaria W_2, W_1dodaje utwór

Rozważono także możliwość awarii podczas procesu wysyłania utworu. Przykład przedstawiony na rys. 5.15 pokazuje awarię W_2 podczas dodawania utworu do bazy. W tej sytuacji W_1 doda utwór do bazy muzyki.

5.3.3 Zarządzanie odtwarzaczem i listą odtwarzania

Rozważono dwa podejścia do odtwarzania muzyki z punktu widzenia użytkownika systemu:

1. utwór z kolejki odtwarzany jest na wszystkich odtwarzaczach – odtwarzanie synchronizowane,
2. każdy odtwarzacz pobiera z kolejki utwór osobno i go odtwarza nie synchronizując się z pozostałymi – odtwarzanie autonomiczne.

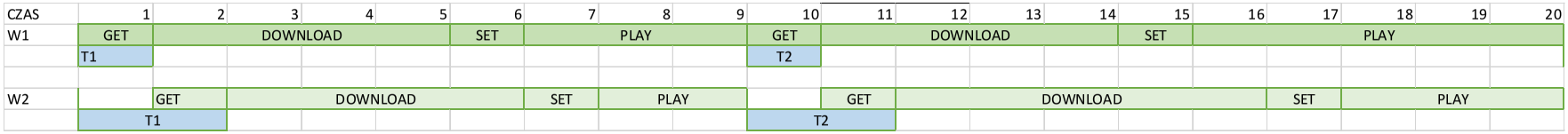
Podejście pierwsze wynika z faktu, że zgodnie z założeniami w systemie może być dostępny więcej niż jeden węzeł podłączony do sprzętu odtwarzającego muzykę. Odtwarzacze mogą być dowolnie rozproszone fizycznie w przestrzeni budynku lub pomieszczenia, w którym uruchomiony jest system. Użytkownik przemieszczając się lokalu np. po kawiarni, chciałby słuchać tego samego utworu w każdym miejscu. Może się zdarzyć, że użytkownik systemu będzie słyszał dźwięk z więcej niż jednego źródła jednocześnie. Aby zachować komfort i pozwolić użytkownikowi na skupienie się na muzyce powinien on słyszeć jedynie jeden utwór w tym samym momencie. Realizacja dokładnej synchronizacji dźwięku jest poza obszarem obejmowanym przez tą pracę, więc założono możliwe przesunięcia w odtwarzaniu lub ewentualną implementację mechanizmu synchronizującego czas w systemie na niższej warstwie abstrakcji.

W drugiej sytuacji każdy utwór odtwarzany jest przez inny odtwarzacz. Wtedy odtwarzacze mogą znajdować się w różnych pomieszczeniach i nie spowoduje to dyskomfortu u słuchacza poprzez nakładanie się na siebie utworów. Każde z tych podejść generuje różnice w potrzebie synchronizacji i zachowania spójności systemu, które rozważono poniżej. Na potrzeby implementacji obu rozwiązań zdefiniowano następująco obiekt bazy muzyki.

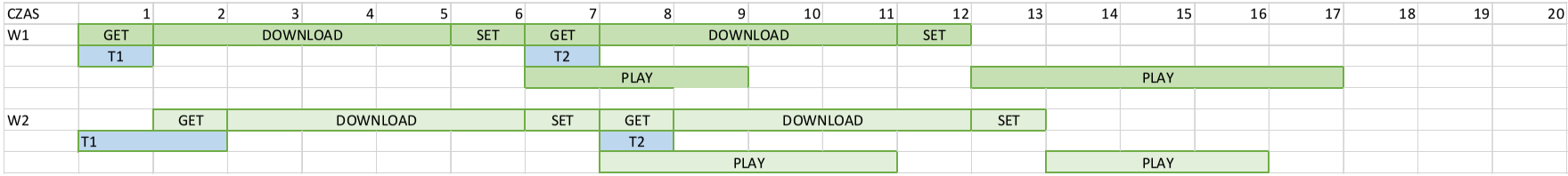
- `MusicQueue` – obiekt posiadający informację o kolejce odtwarzania utworów:
  - `getTrack()` – zwraca pierwszy utwór z kolejki i go z niej usuwa,
  - `addTrack(Track track)` – dodaje utwór na koniec kolejki,
  - `removeTrack(Track track)` – usuwa konkretny utwór z kolejki odtwarzania,
  - `moveTrack(Track track, Integer position)` – przesuwa konkretny utwór na dane miejsce w kolejce odtwarzania.

5.3.3.1 Odtwarzanie autonomiczne

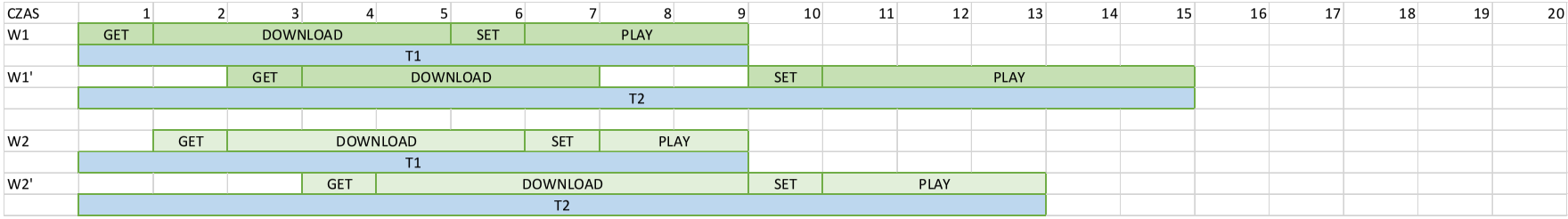
Każdy odtwarzacz posiada pełną autonomię, pobiera i odtwarza utwory z bazy muzyki nie synchronizując odtwarzania z pozostałymi. Na potrzeby tego rozwiązania zdefiniowano następująco obiekt odtwarzacza muzyki i jego metody.



Rysunek 5.16: Przebieg autonomicznego odtwarzania utworów przez dwa węzły



Rysunek 5.17: Przebieg autonomicznego odtwarzania utworów przez dwa węzły – optymalizacja z użyciem dodatkowego procesu odtwarzającego muzykę



Rysunek 5.18: Przebieg autonomicznego odtwarzania utworów przez dwa węzły – optymalizacja z użyciem na każdym węźle dwóch procesów wykonujących atomową transakcję

- `Player` – obiekt zarządzający odtwarzaniem muzyki na konkretnym węźle:

    1. `setTrack(Track track)` – ustawia dany utwór do odtwarzania,
    2. `downloadTrack(Track track)` – pobiera plik z utworem z bazy muzyki,
    3. `play()` – odtwarza aktualnie ustawiony utwór jeżeli jest już pobrany.

Zdefiniowano również następujący proces dodawania utworu do bazy muzyki:

1. Użytkownicy dodają do kolejki muzyki utwory – `MusicQueue.addTrack(track)`,
2. Każdy odtwarzacz wykonuje polecenia:

    (a) `GET: Track track = MusicQueue.getTrack()`,
    (b) `DOWNLOAD: MusicDatabase.downloadTrack(track)`,
    (c) `SET: Player.setTrack(track)`,
    (d) `PLAY: Player.play()`.

Rozważono możliwość zachowania spójności systemu i synchronizacji podczas operacji na współdzielonych obiektach na podstawie różnych sytuacji przedstawionych na diagramach.

Diagram 5.16 przedstawia przebieg tego procesu dla dwóch węzłów z uwzględnieniem propozycji transakcji. Jedynym obiektem współdzielonym przez węzły jest `MusicQueue`, metody obiektów `Player` są wywoływane lokalnie. Z tego powodu objęto transakcją polecenia `MusicQueue.getTrack()`. Uzyskano dzięki temu synchronizację dostępu do listy utworów przez odtwarzacze oraz inne obiekty, które mogłyby w tym czasie dodawać, usuwać czy zmieniać kolejność listy utworów. Transakcja wykonywana na jednym obiekcie nie pozwala jednak na wykorzystanie korzystnych aspektów pamięci transakcyjnej, w tym przypadku jest równorzędna z zastosowaniem zamków. Można też zauważyć, że odtwarzacze nie grają utworów w sposób ciągły. Czas potrzebny na odtworzenie utworów przez węzeł pierwszy to 20 jednostek, podczas gdy te utwory trwają 8 jednostek. Dla węzła drugiego jest to także 20 jednostek dla utworów o łącznej długości 5 jednostek. Dodatkowo czas pomiędzy odtworzeniami dla W_1 wynosi $16 - 9 = 7$ jednostek czasu, a dla W_2 jest to $18 - 9 = 9$ jednostek czasu. Spowodowane jest to brakiem współbieżności pobierania i odtwarzania oraz oczekiwaniem na zwolnienie obiektu `MusicQueue`, aby pobrać utwór.

Ze względu na fakt, iż płynne odtwarzanie muzyki jest kluczowym wymaganiem w opracowywanym systemie, przeanalizowano możliwe usprawnienia. Odtwarzanie utworu trwa relatywnie długo. W tym czasie mogłyby być wykonywane operacje dążące do pobrania kolejnego utworu. Po zakończeniu odtwarzania byłby on od razu dostępny, co zminimalizowałoby przerwy. Na diagramie 5.17 przedstawiono propozycje ulepszenia. Dla każdego węzła, odtwarzanie podzielono na dwa procesy działające współbieżnie – pobierający utwory oraz je odtwarzający. Dzięki temu w czasie odtwarzania utworu w tle pobierany jest kolejny. Minimalizuje to czas oczekiwania na odtworzenie np. w przypadku rywalizacji wielu węzłów o zasób kolejki muzyki. Odtworzenie tych samych utworów co w poprzedniej sytuacji zajęło tym razem W_1 – 17 jednostek czasu, a W_2 – 16 jednostek czasu co dowodzi, że ten sposób implementacji jest lepszy niż poprzedni. Zauważono, że utwory są pobierane sekwencyjnie, jeden po drugim, co prowadzi do niepotrzebnego oczekiwania odtwarzacza.

Zaproponowane powyżej oddzielenie procesu odtwarzającego od pobierającego nie zwiększyło współbieżności pobierania, które wciąż wykonywane jest w kontekście jednego węzła sekwencyjnie. Na listingu 5.19 pokazano transakcję wykonywaną atomowo w celu uzyskania współbieżności pobierania muzyki.

```
1 transaction.begin();
2
3 Track track = MusicQueue.getTrack();
4 MusicDatabase.downloadTrack(track);
5 Player.setTrack(track);
6 Player.play(),
7
8 transaction.commit()
```

Rysunek 5.19: Atomowa transakcja odtwarzająca jeden utwór z bazy muzyki

W przedstawionych rozwiązaniach synchronizacja dostępu wymagana była wyłącznie dla obiektu zdalnego `MusicQueue`. W tym przypadku potrzebne było objęcie synchronizacją również obiektu `Player`, który jest lokalny dla danego węzła. Zdefiniowany jest jako obiekt zdalny, więc jego metody mogą być wykonane poprzez intefejs zdalny. Pobranie pliku za pomocą `MusicDatabase.downloadTrack(track)` zachodzi w trakcie trwania transakcji ale w przypadku pobierania utworu nie jest wymagana synchronizacja dostępu. Zdefiniowano następującą liczbę wywołań obiektów dla tej transakcji, aby wykorzystać mechanizm wczesnego zwalniania obiektów:

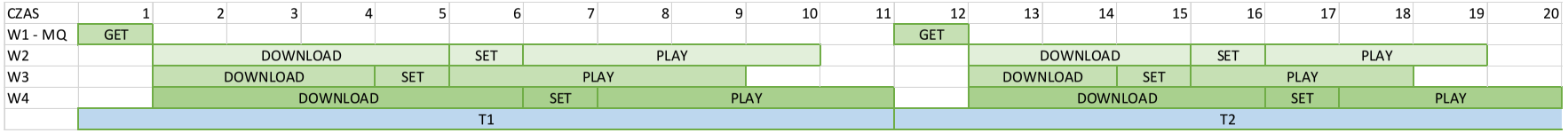
- `MusicQueue` – 1 wywołanie,
- `Player` – 2 wywołania.

Dzięki temu obiekt `MusicQueue` jest zwalniany od razu po użyciu i jest dostępny dla kolejnych transakcji.

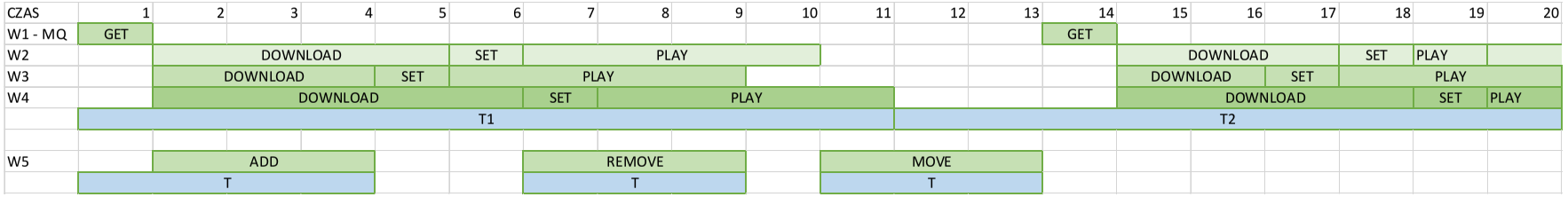
Rys. 5.18 przedstawia przeplot operacji przy użyciu powyższej implementacji. Widać na nim zastosowanie mechanizmu wczesnego zwalniania obiektów:

- wszystkie 4 wątki (W_{1_1}, W_{1_2}, W_{2_1}, W_{2_2}) rozpoczynają transakcję, W_{1_1} otrzymuje dostęp do obiektu zdalnego `MQ` i wykonuje na nim `GET`,
- po upływie 1 jednostki czasu W_1 wykonał polecenie i kontynuuje wykonywanie T_1, ale zwalnia już obiekt `MQ`,
- W_{2_1} otrzymuje dostęp do `MQ`,
- itd.

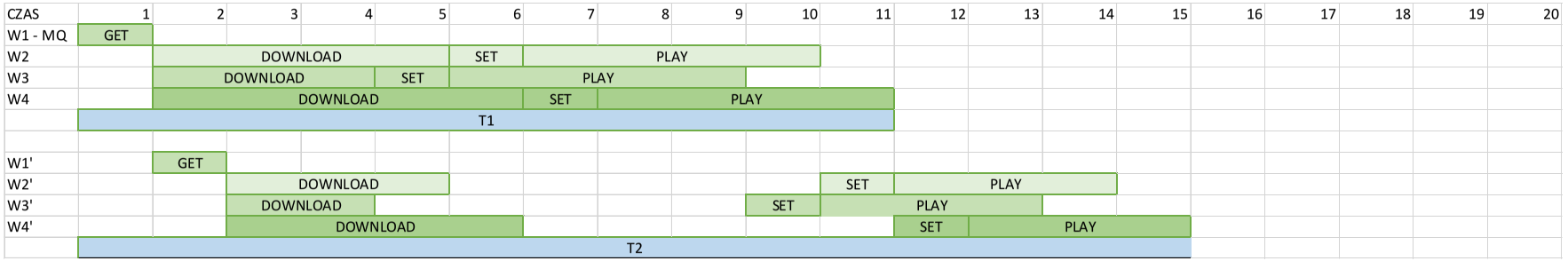
Zwiększona współbieżność pobierania utworów pozytywnie wpłynęła na czas wykonania poleceń – dla W_1 jest to 15 jednostek, dla W_2 – 13 jednostek.



Rysunek 5.20: Odtwarzanie utworu na wielu węzłach jednocześnie



Rysunek 5.21: Odtwarzanie utworu na wielu węzłach jednocześnie – rywalizacja o obiekt kolejki muzyki



Rysunek 5.22: Odtwarzanie utworu na wielu węzłach jednocześnie – optymalizacja z użyciem drugiego procesu wykonującego atomową transakcję

5.3.3.2 Odtwarzanie synchronizowane

Aby uzyskać jednoczesne odtwarzanie na wielu odtwarzaczach potrzebny jest obiekt, który będzie sterować całym procesem i kontrolować jego poprawność. Dodanie do obiektu kolejki muzyki (`MusicQueue`) informacji o wszystkich odtwarzaczach w systemie spowodowało, że posiada on wszystkie wymagane informacje, aby pełnić taką funkcję. Na potrzeby implementacji tego rozwiązania obiekt odtwarzacza zdefiniowano następująco:

- `Player` – obiekt zarządzający odtwarzaniem muzyki na konkretnym węźle:
  - `setTrack(Track track)` – ustawia dany utwór do odtwarzania,
  - `downloadTrack()` – pobiera aktualnie ustawiony plik z utworem z bazy muzyki,
  - `play()` – odtwarza aktualnie ustawiony utwór jeżeli jest już pobrany.

Oraz wstępnie zdefiniowano proces:

1. Użytkownicy dodają do kolejki muzyki utwory – `MusicQueue.addTrack(track)`,
2. Kolejka dla każdego przypisanego odtwarzacza ustawia pierwszy utwór jaki jest w niej zapisany:
   (a) `Track track = MusicQueue.getTrack()`,
   (b) dla każdego z $i = 1...N$ odtwarzaczy wywołaj `Player[i].setTrack(track)`,
   (c) dla każdego z $i = 1...N$ odtwarzaczy wywołaj `Player[i].downloadTrack()`,
   (d) dla każdego z $i = 1...N$ odtwarzaczy wywołaj `Player[i].playTrack()`,

Pierwsze podejście do rozwiązania problemu zostalo zaprezentowane na rys. 5.20. Węzeł sterujący rozpoczyna transakcję i pobiera utwór ze swojej kolejki. Następnie wysyła współbieżnie do wszystkich odtwarzaczy przypisanych do niego informację o tym utworze i poleca rozpoczęcie pobierania utworu. Po pobraniu utworu odtwarzacz otrzymuje polecenie, aby zaczął odtwarzać muzykę. Po zakończeniu odtwarzania transakcja jest zakończona i rozpoczyna się proces odtworzenia kolejnego utworu. W tej sytuacji nie występuje współbieżny dostęp do tych samych obiektów. Transakcje służą do zachowania spójności w przypadku awarii poprzez procedurę wycofania lub powtórzenia. Podobnie jak na rys. 5.16 odtwarzanie nie jest wykonywane optymalnie i pomiędzy odtworzeniami są znaczące przerwy, podczas których są pobierane utwory. Kolejnym problemem jest możliwość rywalizacji na obiekcie kolejki muzyki. Dostęp do niego mają też węzły klienckie, które mogą zarządzać kolejnością utworów oraz dodawać i usuwać nowe. Sytuacja przedstawiona na rys. 5.21 pokazuje, że klient manipulujący kolejką odtwarzania może zwiększać przerwę w odtwarzaniu utworów.

W celu zoptymalizowania rozwiązania rozważono możliwość stworzenia drugiego procesu sterowanego przez tę samą kolejkę muzyki, który równolegle wykonywałby identyczną transakcję. W przypadku odtwarzania autonomicznego zaprezentowanego na rys. 5.18 obiekt odtwarzacza wykonywał sam transakcję i mógł wywołać pobieranie lokalnie bez użycia interfejsu. Dzięki temu nie było potrzeby rywalizować o zdalny obiekt odtwarzacza w transakcji do czasu, aż utwór został pobrany. Sterowanie wieloma odtwarzaczami przez jedną kolejkę muzyki wymaga jednak wywoływania wszystkich poleceń

zdalnie po uprzednim uzyskaniu dostępu do zdalnego obiektu. Zauważono, że operacja `Player.downloadTrack()` nie wymaga synchronizacji mimo, że jest częścią współdzielonego obiektu, ponieważ pobieranie utworu nie zmienia stanu tego obiektu. Wywołanie tej metody bezpośrednio na obiekcie `Player` zamiast na transakcyjnym proxy pozwala na osiągnięcie wyższej współbieżność ponieważ pobieranie utworów może zachodzić jednocześnie przez oba procesy.

Rozwiązanie to zostało przedstawione na rys. 5.22. Pierwszy wątek realizuje odtwarzanie w transakcji tak samo jak w poprzednim przykładzie. Zajmuje on obiekt bazy muzyki, aby pobrać z niego utwór. Dzięki mechanizmowi wczesnego zwalniania obiektów jest on dostępny dla wątku klienta, który dodaje do kolejki nowy utwór i zwalnia obiekt. Wątek drugi pobiera kolejny utwór z bazy muzyki i rozpoczyna pobieranie jego pliku. Dzięki temu, że drugi proces ma dużo czasu na pobranie utworu z bazy muzyki zanim uzyska dostęp do odtwarzacza, wpływ rywalizacji o obiekt muzyki na wydajność został zminimalizowany. Rezultatem tego jest również zmniejszenie przerwy pomiędzy odtworzeniami.

5.4 Usługa lokalizacji zdalnych obiektów

Po analizie architektury Java RMI pod kątem projektowanej aplikacji rozproszonej zauważono, że rejestr jest pojedynczym punktem awarii i potencjalnym wąskim gardłem. W oryginalnej wersji Oracle rejestr nie jest zreplikowany i znajduje się na jednym węźle. Rejestr jest kluczową częścią systemu, pełni funkcję usługi udostępniającej informację o wszelkich obiektach w systemie. Aby zachować wysoką odporność na awarie zdecydowano o implementacji rozproszonego rejestru RMI rozszerzającego podstawowy rejestr Java RMI. Postawowym założeniem było zwielokrotnienie listy zdalnych obiektów przechowywanych przez rejestr, tak aby jego awaria nie powodowała awarii całego systemu. Kolejnym czynnikiem decydującym był aspekt wydajnościowy, rozłożenie odwołań do rejestru na wiele węzłów może decydować o wydajności całego systemu.

Rozważono przypadek, w którym system opiera się na pojedynczym rejestrze, który umieszczony jest jednym z węzłów serwerowych. Odwołania do rejestru występują w następujących sytuacjach. Serwer dodaje do niego informacje o wszystkich zdalnych obiektach, które są na nim dostępne oraz pobiera informacje o obiektach udostępnianych przez inne serwery. Dzięki temu posiada możliwość komunikacji z nimi, zlecania im zadań oraz pobierania wyników. Charakterystyka użycia systemu sugeruje, że takie odpytywanie rejestru zachodzi na początku podczas dołączania serwera, gdy buduje on migawkę całego systemu oraz po dołączeniu każdego nowego serwera, gdy należy uzyskać informacje o obiektach udostępnianych przez niego. Również każdy z klientów podczas dołączania powinien uzyskać informacje o obiektach dostępnych w systemie. Ponadto po każdym dołączeniu lub odłączeniu serwera te informacje powinny zostać zaktualizowane. Biorąc pod uwagę konieczność skalowania systemu do dowolnej liczby węzłów widać, że jest to pojedynczy punkt awarii dla całego systemu. Jego awaria sprawia, że pomimo działających pozostałych n-1 węzłów system nie jest sprawny. Nasuwającym się rozwiązaniem jest więc jego replikacja, opisana w [35].

Rozdział 6

Ewaluacja Eksperymentalna

W niniejszym rozdziale opisano ewaluację zaproponowanej aplikacji wykorzystującej pesymistyczną rozproszoną pamięć transakcyjną.

6.1 Implementacja

W ramach pracy zaimplementowano aplikację spełniającą rzeczywiste przypadki użycia omówione w rozdziale 5. Celem implementacji było sprawdzenie w praktyce zaprojektowanych wcześniej rozwiązań i ewaluacja realnego zastosowania pesymistycznej rozproszonej pamięci transakcyjnej.

Aplikacja zaimplementowano przy użyciu Javy w wersji 7. Dzieli się ona na serwer oraz klienta z interfejsem graficznym. Interfejs graficzny pozwala na użycie podstawowych funkcji systemu, czyli zarządzania utworami i ich odtwarzaniem. Umożliwa on także podstawową analizę zachowania systemu poprzez konsolę logującą wykonywane transakcje.

6.2 Opis techniczny sposobu pomiarów i badań

Środowisko testowe składało się z dwóch komputerów:

1. Komputer stacjonarny podłączony do głośników 5.1, wyposażony w procesor Intel Core i5-4670K 3.40GHz, 8GB pamięci DDR3 1600MHz oraz dysk SSD Plextor M5Pro 128GB. Aplikacja uruchamiana była w systemie Linux Mint zwirtualizowanym poprzez VirtualBox w wersji 5.0.0, któremu przydzielono dwa rdzenie procesora. System gospodarza, w którym uruchomiony był VirtualBox to Windows 10.
2. Laptop wyposażony w procesor Intel Core i7-4510U 2.00GHz, 8GB pamięci DDR3 1600MHz oraz dysk SSD Samsung PM851 256GB. Aplikacja uruchamiana była w systemie Linux Mint zwirtualizowanym poprzez VMware Player w wersji 7.1.2, któremu przydzielono dwa rdzenie procesora. System gospodarza, w którym uruchomiony był VMware Player to Windows 10.

Urządzenia połączone zostały siecią lokalną poprzez router Linksys WRT54GL. Komputer stacjonarny został podłączony do niego przewodowo za pomocą skrętki. Laptop podłączony został do routera poprzez sieć WiFi w standardzie 802.11g o przepustowości 54Mbit.

W celu wykonania testów na komputerze stacjonarnym uruchamiana była odpowiednia liczba homogenicznych serwerów aplikacji, a na laptopie odpowiednia liczba aplikacji klienckich. Zaimplementowane zostały aplikacje testowe wykonujące odpowiednie operacje na aplikacji klienckiej i serwerowej. Mierzony był czas od rozpoczęcia procedur przewidzianych w teście do ich zakończenia. Operacje przygotowujące środowisko testowe nie były wliczane w czas trwania testu. Każdy przypadek testowy był uruchamiany trzykrotnie, a ostateczny wynik uśredniony aby uzyskać miarodajne wyniki.

6.3 Testy

Przeprowadzono testy rozwiązań omówionych w rozdziale 5 w realnych przypadkach użycia oraz pod kątem wydajności i zasadności zastosowania pesymistycznej rozproszonej pamięci transakcyjnej Atomic RMI.

6.3.1 Dodawanie utworów do bazy muzyki

Celem testu było porównanie wydajnościowe dodawania utworów do bazy muzyki z użyciem dwóch podejść opisanych w sekcji 5.3.2:

1. Przesyłanie pliku objęte w całości pojedynczą transakcją (patrz rys. 5.12),
2. Przesyłanie pliku podzielone na dwie transakcje (patrz rys. 5.13).

Procedura dodawania utworów do bazy muzyki jest wykonywana za pomocą wywołań metod tylko jednego obiektu zdalnego. W tej sytuacji objęcie operacji pojedynczą transakcją, zaproponowane w pierwszym rozwiązaniu, pozwala na zachowanie pełnej spójności kosztem ograniczenia współbieżności. Drugie rozwiązanie polega na użyciu drobnoziarnistych transakcji zamiast pojedynczej. Jest ono pewnym kompromisem pomiędzy wydajnością, a spójnością.

Procedura testowa polegała na uruchomieniu jednego serwera z bazą muzyki i m klientów z nią połączonych. Każdy z klientów dodaje do bazy muzyki 10 unikalnych utworów. Utwory są tworzone poprzez klonowanie przykładowego pliku typu Wave o rozmiarze 3,5 MB. Zmierzono czas od rozpoczęcia wysyłania utworów do zakończenia ich wysyłania przez ostatniego klienta. Im krótszy zmierzony czas tym większa wydajność badanego rozwiązania. Przeprowadzono testy dla $m = 2, 4, 8, 16, 32, 48$.

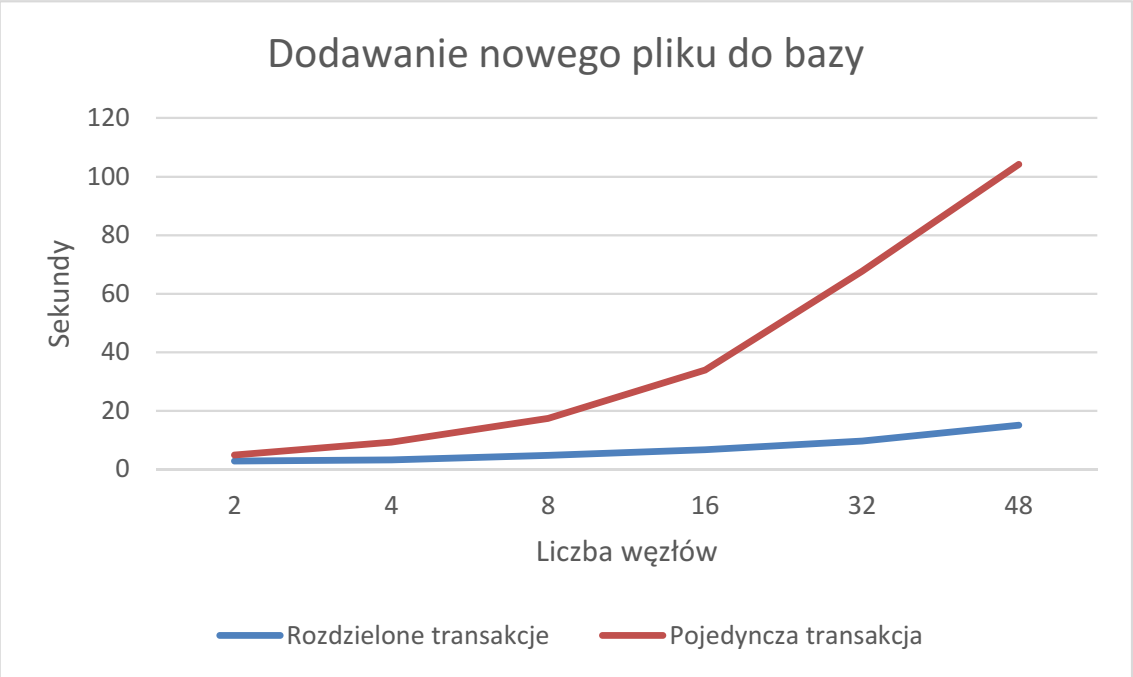
Wnioski

Wyniki testów przedstawione na rys. 6.2, zgodnie z rozważaniami teoretycznymi, pokazują przewagę wydajności rozwiązania w którym przesyłanie plików objęte jest drobnoziarnistymi transakcjami zamiast gruboziarnistą transakcją. Dla dwóch węzłów transakcje

drobnoziarniste uzyskują wydajność równą 173% wydajności pojedynczej transakcji. Wraz ze wzrostem liczby węzłów przewaga wydajności rośnie – dla 48 węzłów transakcje drobnoziarniste osiągają 689% wydajności pojedynczej transakcji.

Analiza tego przypadku pokazuje, ze pesymistyczna pamięć transakcyjna nie pozwala na zwiększenie współbieżności w przypadku transakcji gruboziarnistych obejmujących wiele operacji na pojedynczym obiekcie. Obiekt używany w transakcji jest zajmowany na jej początku i zwalniany na końcu co odpowiada zastosowaniu zamka gruboziarnistego na całej operacji. Zaletą jest zachowanie spójności danych – w przypadku wystąpienia błędu transakcja może zostać wycofana lub powtórzona. Zaimplementowanie takiej transakcji jest bardzo proste – dzięki użyciu pojedynczej transakcji możemy bardzo prosto uzyskać gwarancję spójności, jeżeli wydajność ma drugorzędne znaczenie.

W rozważanej sytuacji użycie pamięci transakcyjnej jest zbliżone do zastosowania zamków. Zaimplementowanie drobnoziarnistych transakcji, podobnie jak w przypadku drobnoziarnistych zamków, wymaga analizy problemu synchronizacyjnego. Uzyskano dzięki temu dużo wyższą współbieżność operacji. Zachowanie spójności wymaga jednak dodania do istniejących mechanizmów wycofywania transakcji dodatkowej logiki, która w przypadku pojedynczej transakcji nie była potrzebna.



Rysunek 6.1: Porównanie wydajności transakcji gruboziarnistych i drobnoziarnistych dodających nowe pliki do bazy muzyki

6.3.2 Pobieranie listy utworów z bazy muzyki

Przeprowadzono testy następujących rozwiązań opisanych w sekcji 5.3.1:

1. Odpytanie baz muzyki w jednej transakcji bez zwalniania obiektów (patrz rys. 5.4).

Wykonana jest pojedyncza transakcja, która zajmuje wszystkie zdalne obiekty baz muzyki i zwalnia je dopiero gdy zostanie zatwierdzona.

2. Odpytanie baz muzyki w jednej transakcji z wczesnym zwalnianiem obiektów (patrz rys. 5.5). Podobnie jak w pierwszym rozwiązaniu wykonywana jest pojedyncza transakcja obejmująca wszystkie bazy muzyki. Wykorzystywany jest jednak mechanizm wczesnego zwalniania obiektów.

Oba rozwiązania zakładają, że odpytywanie baz muzyki zachodzi na wszystkich węzłach w jednakowej kolejności.

Celem testów było sprawdzenie wydajności mechanizmu wczesnego zwalniania obiektów w pesymistycznej pamięci transakcyjnej. Procedura testowa zakładała n klientów połączonych z n serwerami w ten sposób, że każdy klient połączony jest z innym serwerem. Każdy serwer posiada własną bazę muzyki w której znajduje się 10 utworów. Każdy węzeł 100 razy pobiera listę utworów dostępnych w całym systemie. Węzły wykonują swoje operacje współbieżnie. Zmierzony został czas od rozpoczęcia pobierania listy utworów do momentu gdy ostatni węzeł skończy pobierać listę utworów. Wykonane zostały przypadki testowe dla $n = 2, 4, 8, 16, 32, 48$.

Wnioski

Wyniki testów zostały przedstawione na rys. 6.2. Rozwiązanie z wykorzystaniem wczesnego zwalniania obiektów uzyskało lepsze wyniki dla każdej liczby węzłów. Wzrost wydajności osiągnięty przy użyciu tego mechanizmu względem rozwiązania bez wczesnego zwalniania obiektów wyniósł od 21% dla 2 baz muzyki do 495% dla 48 baz muzyki. Wyniki testu pokazują, że procedura pobrania utworów ze wszystkich baz muzyki w jednej transakcji została znacząco zoptymalizowana poprzez użycie wczesnego zwalniania obiektów.

Pojedyncza transakcja wykonywana bez wczesnego zwalniania obiektów posiada wydajność odpowiadającą zastosowaniu zamka gruboziarnistego. Uruchomienie takiej transakcji wymaga zajęcia obiektów n baz muzyki, które w tej sytuacji są zwalniane dopiero po zatwierdzeniu transakcji. Oznacza to, że obiekt pierwszej odpytanej bazy muzyki będzie zablokowany do czasu aż odpytana zostanie ostatnia baza mimo, że nie są wykonywane na nim żadne operacje.

Liczba operacji w transakcji rośnie wprost proporcjonalnie do liczby węzłów serwerowych, czyli do liczby dostępnych baz muzyki. Wykonanie transakcji z użyciem wczesnego zwalniania obiektów również wymaga wstępnie zajęcia wszystkich baz muzyki. Różnica wydajności wynika z faktu, że obiekt każdej odpytanej bazy danych jest zwalniany od razu po wykonaniu operacji i udostępniany kolejnym klientom. Dzięki temu przetwarzanie zachodzi współbieżnie a przewaga wydajności transakcji wykorzystującej mechanizm wczesnego zwalniania obiektów rośnie wraz ze wzrostem liczby węzłów.

Użycie pesymistycznej pamięci transakcyjnej dla transakcji wykonującej metody wielu obiektów pozwala na osiągnięcie znaczącego wzrostu współbieżności i efektywności przetwarzania przy zachowaniu spójności wykonywanych operacji. Wymaganiem dla użycia

mechanizmu wczesnego zwalniania obiektów jest wyłącznie znajomość liczby wywołań metod każdego ze zdalnych obiektów biorących udział w transakcji.

Rysunek 6.2: Porównanie wydajności odpytywania baz muzyki z użyciem transakcji wykorzystującej mechanizm wczesnego zwalniania obiektów oraz transakcji niewykorzystującej tego mechanizmu

6.3.3 Odtwarzanie utworów

Przeprowadzono testy wydajnościowe implementacji rozwiązań omówionych w podsekcji 5.3.3, które opisują dwa podejścia do odtwarzania muzyki w wykonywanym systemie:

1. Odtwarzanie autonomiczne – każdy odtwarzacz działa autonomicznie, pobiera utwory z kolejki odtwarzania i nie synchronizuje odtwarzanych utworów,
2. Odtwarzanie synchronizowane – kolejka odtwarzania steruje odtwarzaniem tych samych utworów na wszystkich połączonych z nią odtwarzaczach.

6.3.3.1 Odtwarzanie autonomiczne

Przeprowadzono porównanie wydajności rozwiązań pokazanych na rys. 5.16 oraz 5.18. Celem testu było sprawdzenie wydajności omówionych rozwiązań w zależności od liczby jednocześnie odtwarzających węzłów. Każdy z węzłów odtwarzających pobiera utwory z tego samego węzła z bazą muzyki co oznacza, że przepustowość łącza sieciowego może mieć znaczący wpływ na wydajność.

Przeprowadzona procedura testowa polegała na odtworzeniu przez każdy z odtwarzaczy muzyki 10 utworów pobranych z kolejki odtwarzania. Kolejka odtwarzania zawiera dodane

wcześniej $10n$ utworów o jednakowej długości, gdzie n to liczba węzłów odtwarzających muzykę. Przeprowadzono testy obu rozwiązań dla $n = \{2, 4, 8, 16, 32, 48\}$.

Wnioski

Wyniki przedstawiono na rys. 6.3. Rozwiązanie wykorzystujące na każdym węźle dwa procesy wykonujące autonomiczne transakcje pozwala na osiągnięcie większej wydajności. Czas wymagany na pobranie utworów przez to rozwiązanie był krótszy o 38% dla 2 węzłów od rozwiązania z użyciem jednego procesu i różnica ta rosła do 45% dla 16 węzłów jednocześnie odtwarzających muzykę. Testy wykonane dla wyższej liczby węzłów tj. 32 oraz 48 pokazują zmniejszenie tej różnicy do odpowiednio 35% dla 32 węzłów i 25% dla 48 węzłów.

Użycie dwóch procesów jednocześnie pozwala na pobieranie kolejnego utworu podczas odtwarzania muzyki przez każdy z węzłów. Dzięki temu mogą one odtwarzać muzykę płynnie, bez przerw pomiędzy utworami. Wraz z rosnącą liczbą węzłów zwiększa się liczba procesów jednocześnie pobierających utwory z bazy muzyki. Przepustowość połączenia sieciowego jest ograniczona w związku z czym każdy proces pobiera utwór ze średnią prędkością równą $przepustowośćŁącza/liczbaWezłów$. Czas pobierania utworu jest więc dłuższy dla większej liczby węzłów.

Przepustowość łącza ma bezpośredni wpływ na wydajność dla każdej liczby węzłów w przypadku pojedynczego procesu. W przypadku dwóch procesów przepustowość nie wpływa na czas odtwarzania tak długo jak czas pobierania utworu jest krótszy niż czas odtwarzania. Dla przypadków testowych z liczbą węzłów mniejszą lub równą 16 czas pobierania jest krótszy niż odtwarzania, co powoduje optymalizację poprzez brak oczekiwania na pobranie kolejnego utworu. Dla przypadków z 32 i 48 węzłami czas ten jest już dłuższy, co prowadzi do czekania i powoduje zmniejszenie zysku z zastosowania dwóch procesów współbieżnych na każdy węzeł.

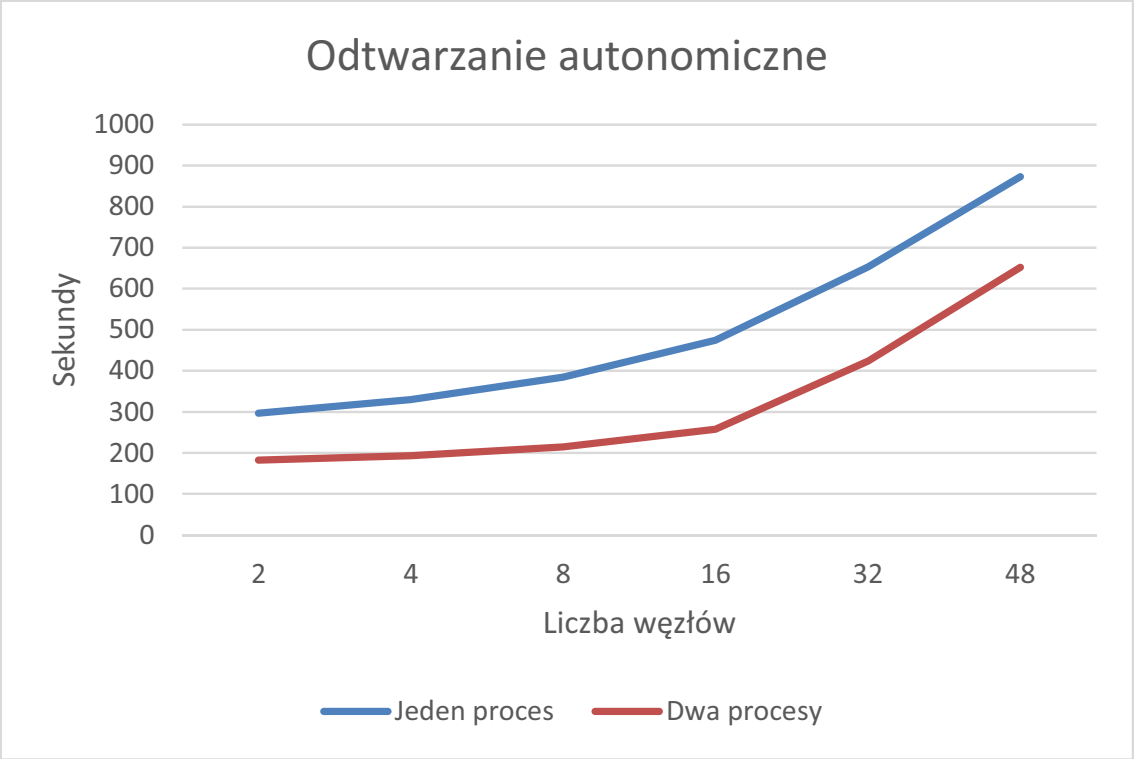
6.3.3.2 Odtwarzanie synchronizowane

Wykonano porównanie wydajności rozwiązań pokazanych na rys. 5.20 oraz rys. 5.22. Celem testu było sprawdzenie wydajności omówionych rozwiązań w zależności od stopnia rywalizacji o dostęp do obiektu kolejki odtwarzania.

Przeprowadzona procedura testowa polegała na odtworzeniu 10 utworów znajdujących się w kolejce odtwarzania na odtwarzaczu muzyki. Zaimplementowano proces, który wykonywał bez przerwy operacje na kolejce odtwarzania w celu zasymulowania wpływu rywalizacji na osiągniętą współbieżność. Przeprowadzono testy obu rozwiązań dla $n = \{0, 1, 2, 4, 8\}$, gdzie n to liczba uruchomionych współbieżnie wątków wykonujących operacje na kolejce odtwarzania.

Wnioski

Wyniki przedstawiono na rys. 6.4. Rozwiązanie wykorzystujące dwa współbieżne procesy uruchomione na węźle sterującym osiągnęło lepsze wyniki od rozwiązania z pojedynczym

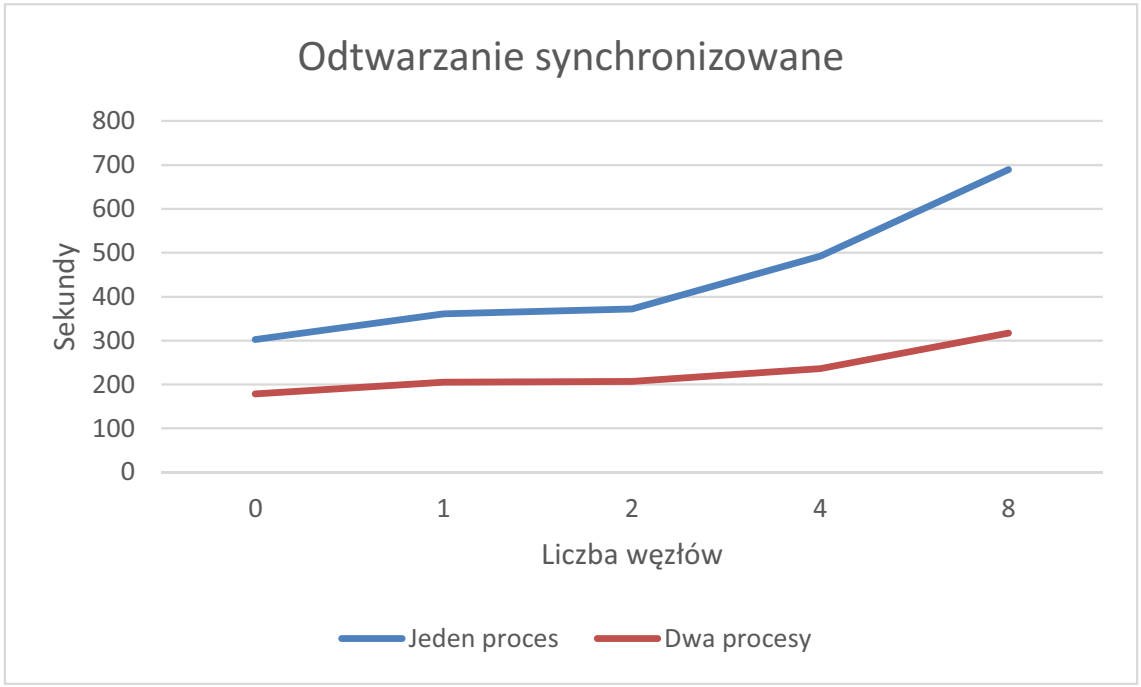


Rysunek 6.3: Porównanie wydajności autonomicznego odtwarzania muzyki z użyciem jednego oraz dwóch procesów współbieżnych

procesem. Czas odtwarzania z użyciem podwójnego procesu był krótszy o 40% przy braku rywalizacji o obiekt kolejki muzyki do 54% przy 8 węzłach wykonujących operacje na kolejce muzyki.

Różnica wydajności rośnie wraz ze stopniem rywalizacji o obiekt kolejki muzyki. Pomimo znaczącego obciążenia kolejki odtwarzania czas wykonywania zoptymalizowanej wersji rośnie wolniej niż wersji z pojedynczym procesem. Implementacja tej optymalizacji możliwa była dzięki mechanizmowi wczesnego zwalniania obiektów, który pozwala obu procesom na jednoczesne pobieranie muzyki co prowadzi do zmniejszenia przerw w odtwarzaniu.

Dodatkowo interesującą funkcjonalnością Atomic RMI okazała się możliwość objęcia transakcją zarówno obiektów zdalnych jak i obiektów znajdujących się na danym węźle. W tym przypadku obiekt kolejki muzyki znajduje się lokalnie na węźle wykonującym operacje, a odtwarzacze znajdują się na pozostałych węzłach. Z punktu widzenia programisty jest to korzystne uproszczenie, ponieważ synchronizacja obiektów lokalnych i zdalnych może odbywać się użyciem tego samego kodu i z zachowaniem wspólnych gwarancji.



Rysunek 6.4: Porównanie wydajności synchronizowanego odtwarzania muzyki z użyciem jednego oraz dwóch procesów współbieżnych

Rozdział 7

Podsumowanie

Przedmiotem raportu jest ewaluacja możliwości zastosowania technologii rozproszonej pamięci transakcyjnej w celu stworzenia rozproszonego systemu zarządzania odtwarzaniem mediów. W ramach raportu zapoznano się z literaturą dotyczącą zagadnienia synchronizacji dostępu do współdzielonych danych w systemach współbieżnych i rozproszonych. Przeanalizowano także badania z zakresu pamięci transakcyjnej i jej zastosowań w systemach współbieżnych i rozproszonych oraz dostępne testy wydajności pamięci transakcyjnej.

Raport definiuje wymagania stawiane przed aplikacją spełniającą rzeczywiste przypadki użycia, która pozwoliłaby na ewaluację rozproszonej pamięci transakcyjnej. W konsekwencji zaproponowano aplikację Atomic Café – rozproszoną szafę grającą, działającą w architekturze klient-serwer. W celu synchronizacji dostępu wykorzystano implementację pesymistycznej pamięci transakcyjnej – Atomic RMI. Rozważono i przeanalizowano występujące w aplikacji funkcjonalności, które wymagałyby synchronizacji współbieżnego dostępu. Zaprezentowano możliwe przeploty operacji oraz wykorzystanie pamięci transakcyjnej do synchronizacji dostępu do współdzielonych obiektów.

W celu sprawdzenia poprawności rozważań zaimplementowano serwer oraz klienta rozproszonej szafy grającej wraz z interfejsem użytkownika. Dodatkowo wykonano rozproszony rejestr obiektów Java RMI, który zwiększył odporność powyższego systemu rozproszonego na awarie.

Na podstawie rozważań teoretycznych oraz przeprowadzonych testów wydajnościowych wyciągnięto wnioski odnoszące się do problemu synchronizacji z użyciem rozproszonej pesymistycznej pamięci transakcyjnej. W przypadku konieczności zachowania synchronizacji wielu operacji wykonywanych na jednym obiekcie objęcie ich pojedynczą transakcją nie zwiększa współbieżności, ale pozwala na zachowanie spójności operacji. Zaimplementowanie transakcji drobnoziarnistych może pozwolić na zwiększenie współbieżności, ale należy wziąć pod uwagę możliwość utraty spójności systemu.

Wykorzystanie pesymistycznej pamięci transakcyjnej umożliwia uzyskanie wysokiej współbieżności transakcji wywołujących metody wielu obiektów dzięki mechanizmowi wczesnego zwalniania obiektów. Zastosowanie tego mechanizmu wymaga znajomości liczby wywołań wykonywanych na zdalnych obiektach albo zaprogramowania ręcznego zwolnienia obiektu, jeżeli nie będzie on więcej używany. Na przykładzie omawianej rozproszonej szafy grającej

można stwierdzić, że w przypadku operacji na wielu obiektach pesymistyczne podejście do sterowania współbieżnością prowadzi do znaczącego wzrostu wydajności.

Zastosowana w badaniach implementacja rozproszonej pesymistycznej pamięci transakcyjnej – Atomic RMI, sprawdziła się jako stabilna i łatwa do wykorzystania metoda synchronizacji w systemie rozproszonym. Pozwoliła ona na zastosowanie obiektowego podejścia do programowania przy tworzeniu aplikacji czego efektem jest możliwość prostego jej rozbudowania. Badania pokazały również, że można ją wykorzystać do zachowania spójności transakcji wykonujących operacje zarówno na obiektach zdalnych jak i lokalnych.

Przeprowadzone studium nie wyczerpuje tematu ewaluacji zastosowania rozproszonej pamięci transakcyjnej. Zaproponowaną aplikację można rozszerzyć o większy zestaw testów pamięci transakcyjnej. Złożona charakterystyka tej aplikacji pozwala na zaproponowanie zarówno nowych testów, dotyczących nieomawianych w tej pracy problemów, jak i rozszerzenia istniejących testów o przypadki pozwalające na dokładniejszą analizę pamięci transakcyjnej.

Kolejnym krokiem może być wykorzystanie do synchronizacji optymistycznej pamięci transakcyjnej. Pozwoliłoby to na porównanie wydajności oraz na analizę różnic występujących pomiędzy tymi podejściami.

Bibliografia

[1] Yehuda Afek, Alexander Matveev, and Nir Shavit. Pessimistic Software Lock-elision. In *Proceedings of DISC'12: the 26th International Conference on Distributed Computing*, pages 297–311, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] Mohammad Ansari, Christos Kotselidis, Ian Watson, and Kirkham. Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory. In *Algorithms and Architectures for Parallel Processing*, volume 5022 of *Lecture Notes in Computer Science*, pages 196–207. Springer Berlin Heidelberg, 2008.

[3] Dusan Z. Badal. The Distributed Deadlock Detection Algorithm. *ACM Trans. Comput. Syst.*, 4(4):320–337, September 1986.

[4] Hans Boehm, Justin Gottschlich, Victor Luchangco, Maged Michael, Mark Moir, Clark Nelson, Torvald Riegel, Tatiana Shpeisman, and Michael Wong. Transactional Language Constructs for C++. *ISO/IEC JTC1/SC22 (Programming languages and operating systems) WG21 (C++)*, 2012.

[5] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 Benchmark. In *Proceedings of SIGMOD '93: the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, New York, NY, USA, 1993. ACM.

[6] Edward G. Coffman, Michael Elphick, and Arie Shoshani. System Deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[7] Maria Couceiro, Paulo Romano, Nuno Carvalho, and Luis Rodrigues. D2STM: Dependable Distributed Software Transactional Memory. In *Proceedings of PRDC '09: the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 307–313, Nov 2009.

[8] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (3rd Edition)*. Addison Wesley, 2000.

[9] Florian Funke, Alfons Kemper, Tobias Mühlbauer, Thomas Neumann, and Viktor Leis. HyPer Beyond Software: Exploiting Modern Hardware for Main-Memory Database Systems. *Datenbank-Spektrum*, 14(3):173–181, 2014.

[10] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory. In *Proceedings of ICS '09: the 23rd International Conference on Supercomputing*, pages 126–135, New York, NY, USA, 2009. ACM.

[11] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of EuroSys '07: the Second European Systems Conference*, 2007.

[12] Per Brinch Hansen. *Operating System Principles (Prentice-Hall Series in Automatic Computation)*. Prentice Hall, 1973.

[13] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[14] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[15] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. HiperTM: High Performance, Fault-Tolerant Transactional Memory. In *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 181–196. Springer Berlin Heidelberg, 2014.

[16] Sungpack Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Proceedings of IISWC '10: IEEE International Symposium on Workload Characterization*, pages 1–11, Dec 2010.

[17] Intel. Intel Xeon Processor E3-1200 v3 Product Family Specification Update. http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf, 2015.

[18] Tadeusz Kobus, Maciej Kokocinski, and Paweł T. Wojciechowski. Hybrid Replication: State-Machine-Based and Deferred-Update Replication Schemes Combined. In *Proceedings of ICDCS '13: IEEE 33rd International Conference on Distributed Computing Systems*, pages 286–296, July 2013.

[19] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[20] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[21] Gertrude Levine. Priority Inversion with Fungible Resources. *Ada Lett.*, 31(2):9–14, February 2012.

[22] Wei Li. IBM XL compiler hardware transactional memory builtin functions for IBM AIX on IBM POWER8 processor-based systems. 2014.

[23] Mamoru Maekawa. A N Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, May 1985.

[24] Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of IISWC '08: IEEE International Symposium on Workload Characterization*, pages 35–46, Sept 2008.

[25] Oracle. Java RMI. http://www.oracle.com/technetwork/articles/javase/rmi-corba-136641.html.

[26] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, October 1979.

[27] Glenn Ricart and Ashok K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Commun. ACM*, 24(1):9–17, January 1981.

[28] Mohamed M. Saad. Hyflow: A high performance distributed software transactional memory framework. Master's thesis, 2011.

[29] Konrad Siek and Paweł T. Wojciechowski. Brief Announcement: Towards a Fully-articulated Pessimistic Distributed Transactional Memory. In *Proceedings of SPAA '13: the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 111–114, New York, NY, USA, 2013. ACM.

[30] Konrad Siek and Paweł T. Wojciechowski. Atomic RMI: A Distributed Transactional Memory Framework. *International Journal of Parallel Programming*, pages 1–22, 2015.

[31] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with Isolation and Cooperation. *SIGPLAN Not.*, 42(10):191–210, October 2007.

[32] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. General and Efficient Locking Without Blocking. In *Proceedings of MSPC '08: the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 1–5, New York, NY, USA, 2008. ACM.

[33] William Edward Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–282, April 1989.

[34] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable Transactions and Their Applications. In *Proceedings of the SPAA '08: the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, pages 285–296, New York, NY, USA, 2008. ACM.

[35] Martin Witczak. Atomic Café - Rozproszony system zarządzania odtwarzaniem mediów. Master's thesis, 2015.

[36] Paweł T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems.* Wydawnictwo Politechniki Poznańskiej, Pl. Marii Skłodowskiej-Curie 2, Poznań 60-965, Poland, 1st edition, 2007. 204pp.

[37] Paweł T. Wojciechowski, Olivier Rütti, and André Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium*, April 2004.