STUDIA INFORMATICA 2012
Volume 33 Number 2A (105)

Krzysztof CZAJKOWSKI, Damian PEZDA
Politechnika Krakowska, Instytut Teleinformatyki, Wydział Fizyki, Matematyki i Informatyki

TECHNOLOGIA IN-MEMORY DATA GRID I JEJ ZASTOSOWANIA

Streszczenie. Artykuł omawia problematykę In-Memory Data Grid, jako jednej z możliwości podejścia do kwestii zarządzania danymi. Przedstawiono różne możliwości konfiguracji tego typu rozwiązań. Zagadnienia zostały zaprezentowane na przykładzie aplikacji utworzonej z wykorzystaniem klastera Hazelcast.

Słowa kluczowe: In-Memory Data Grid, rozproszone składowanie danych, rozproszone obliczenia, klaster

IN-MEMORY DATA GRID TECHNLOGY AND ITS IMPLEMENTATION

Summary. The article discusses the issues of In-Memory Data Grid as one of the possible approaches to the data management. The various configurations of such solutions were presented. The issues were presented on the example of application implemented with the use of Hazelcast cluster.

Keywords: In-Memory Data Grid, distributed data storage, distributed calculations, cluster

1. Wstęp

In-Memory Data Grid jest jednym z przykładów podejścia do kwestii zarządzania danymi. Znajduje swoje zastosowanie głównie w przypadkach wymagających bardzo dużej wydajności i skalowalności i dla tych celów została stworzona. Z powodzeniem jednak nadaje się również do rozwiązań dużo mniejszych, nawet domowych. Aby móc lepiej zrozumieć, czym tak naprawdę jest IMDG, należy zacząć od rozszyfrowania samej nazwy. Data Grid w wolnym tłumaczeniu oznacza siatkę danych. W prezentowanym przypadku jest to system,

zbudowany z pewnej liczby serwerów współpracujących ze sobą. Ich zadaniem jest zarządzanie danymi oraz wykonywanie na nich pewnych zadań w rozproszonym środowisku. Po dodaniu „In-Memory" otrzymujemy siatkę danych, która przechowuje dane w pamięci w celu uzyskania najwyższej możliwej wydajności. Takie rozwiązanie charakteryzuje nadmiarowość danych – przez przechowywanie kopii informacji zsynchronizowanych z poszczególnymi serwerami – stosowana w celu uzyskania większej elastyczności i zapewnienia dostępności danych w przypadku awarii któregoś z serwerów. Dane przechowywane wewnątrz tej siatki to obiekty wykorzystywane w aplikacji. Charakteryzuje je wiele czynników. Wymagają one między innymi: krótkiego czasu dostępu, dużej przepustowości, przewidywalnej skalowalności oraz stałej dostępności. Obiekty te zazwyczaj są tworzone w obiektowych językach programowania, takich jak: Java, C# czy Ruby. Obiekty te, często są zbudowane hierarchicznie i zawierają w sobie informacje, które w tradycyjnym podejściu musiałyby być przechowywane w wielu relacyjnych tabelach [3].

W niniejszym artykule została podjęta próba zweryfikowania, czy w obecnych czasach, przy potrzebie coraz szybszego dostępu do informacji, przechowywanie danych w miejscu tak ulotnym jak pamięć operacyjna komputera może stanowić jakąkolwiek konkurencję dla „tradycyjnych" rozwiązań. Autorzy stawiają sobie również pytanie: czy zastosowanie technologii IMDG może wybiegać poza samo przechowywanie danych? W artykule omówiono wymienione rozwiązania oraz zaprezentowano aplikację wykorzystującą je w praktyczny sposób.

2. Technologia IMDG

Dane przechowywane w klastrze IMDG są współdzielone pomiędzy wiele serwerów. Umożliwia to wprowadzenie poziomej skalowalności, co w przypadku wielu aplikacji jest bardzo istotne. Dzięki temu instancja aplikacji uruchomiona na jednym z serwerów bez żadnych problemów może korzystać z danych przechowywanych w zupełnie innym miejscu. Oczywiście istnieje alternatywne rozwiązanie, polegające na wykorzystaniu jednego współdzielonego zasobu, takiego jak relacyjna baza danych. Jednak podejście takie w znacznym stopniu obniża wydajność rozwiązania w związku z koniecznością zarówno skoordynowanego dostępu do odległego zasobu jaki konieczności mapowania danych relacyjnych na obiekty. Ponieważ w klastrze IMDG przechowywane są gotowe obiekty wykorzystywane w wyższych warstwach aplikacji, nie ma potrzeby wprowadzania mapowania obiektowo-relacyjnego (Object/Relational Mapping). Dodatkowo ograniczona jest skalowalność rozwiązania z powodu wąskiego gardła, jakim staje się współdzielona baza danych. Nawet jeżeli sama aplikacja napisana jest w sposób umożliwiający wysoką skalowalność, przeważnie pozostaje znany problem – pojedynczy punkt przechowywania danych.

Przy zastosowaniu rozwiązania In-Memory Data Grid możliwe jest osiągnięcie krótkiego czasu dostępu do danych dzięki wspomnianemu przechowywaniu informacji w pamięci w postaci obiektów i dzielenie tych obiektów przez wiele współpracujących serwerów. Dzięki temu aplikacja jest w stanie uzyskać dostęp do potrzebnych jej danych bez jakiejkolwiek ich transformacji z postaci relacyjnej na obiektową. Dodatkowo sytuacja, w której konieczne jest przesyłanie danych przez sieć, zachodzi tylko wtedy, kiedy chcemy odwołać się do danych zarządzanych przez inny serwer IMDG, co z kolei eliminuje problem wąskiego gardła [3].

Jedną z wielu możliwości technologii In-Memory Data Grid jest kolejkowanie transakcji występujących na danych znajdujących się w pamięci i asynchroniczne zapisywanie ich wyników do bazy danych. Jest to szczególnie ważne w systemach, które mają bardzo wysoki wskaźnik zmian ze względu na przetwarzanie wielu małych transakcji, zwłaszcza gdy jedynie wynik końcowy powinien być zapisywany na stałe. Ciągła dostępność danych jest uzyskiwana w implementacjach In-Memory Data Grid przez kilku elementów.

Po pierwsze, grupa protokołów wykorzystywana przez poszczególne implementacje IMDG może szybko wykryć awarię jednego z serwerów i powiadomić o tym pozostałe działające węzły klastra. Po drugie, dane mogą być synchronicznie powielane na wielu serwerach, dzięki czemu wyeliminowany zostaje pojedynczy punkt awarii. Po trzecie, każdy serwer wie, gdzie dane są replikowane, dzięki czemu możliwe jest ich automatyczne odzyskanie i uaktualnienie. Po czwarte, wszelkie operacje wykonywane są raz i tylko raz, aby operacje, które są realizowane podczas awarii jednego z serwerów, nie uszkodziły danych w trakcie operacji naprawczych (*failover*).

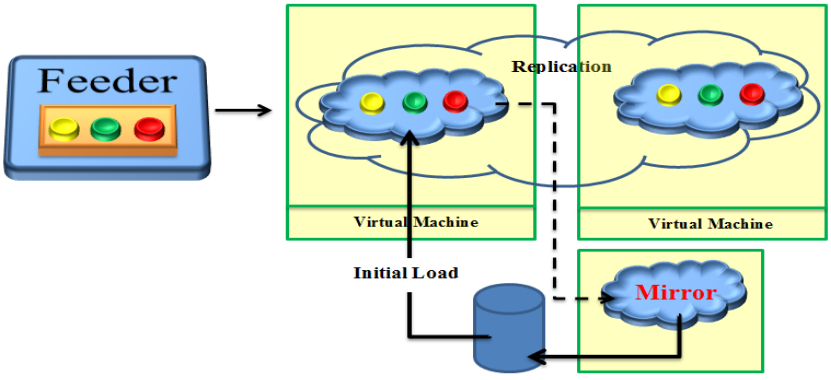
Jak zostało wspomniane wcześniej, IMDG przechowuje dane z wykorzystaniem pewnej liczby serwerów połączonych w klaster. Takie rozwiązanie daje nam wiele możliwości, ale jednocześnie stawia przed użytkownikiem spore wyzwanie. Przy tworzeniu klastra możliwe jest określenie, w jaki sposób dane będę przechowywane pomiędzy poszczególnymi węzłami. Do dyspozycji są następujące topologie IMDG:

- Całkowity backup – dane umieszczone w klastrze są automatycznie propagowane na wszystkie węzły, dzięki czemu każdy pojedynczy węzeł ma swoją kopię danych. Sytuacja ta diametralnie ogranicza ilość danych możliwych do przetrzymywania wewnątrz klastra i równa się pojemności najmniejszego węzła. Jednak zyskujemy maksymalne bezpieczeństwo informacji. Przyjmując, że n jest liczbą węzłów w klastrze, nawet jednoczesna awaria n-1 węzłów nie powoduje utraty danych [9].
- Całkowite rozproszenie danych – polega na tym, że dane umieszone w klastrze, są automatycznie równomiernie rozrzucane po wszystkich węzłach klastra. Rozwiązanie takie daje nam największą możliwą pojemność, która równa się sumie pojemności wszystkich węzłów klastra. Niestety takie podejście ma swoją wadę. W związku z tym, że każdy

węzeł przechowuje jedynie część wspólnych danych całego klastra, nie istnieje żadna kopia zapasowa. Sprowadza się to do tego, że w razie awarii choćby jednego węzła nieunikniona jest utrata części danych [9].

- Częściowe rozproszenie danych – jest to sytuacja pośrednia pomiędzy topologiami przedstawionymi wcześniej. Topologia ta jest różnie realizowana w różnych implementacjach IMDG, jednak główne założenie jest identyczne: zyskać jak największą pojemność klastra przy jednoczesnym zapewnieniu względnego bezpieczeństwa danych. W obecnych czasach niebezpieczeństwo awarii wielu serwerów na raz jest na tyle niewielkie, że często opłaca się poświęcić poczucie bezpieczeństwa na rzecz większej wydajności i pojemności rozwiązania [9].

W przypadku zastosowania dowolnej z wybranych topologii możliwe jest wykorzystanie dodatkowego rozwiązania, pozwalającego na zwiększenie bezpieczeństwa danych, bez jednoczesnej utraty elastyczności i pojemności klastra. Można tego dokonać przez asynchroniczne zapisywanie danych wykorzystywanych w klastrze na trwałym nośniku, którym w szczególności może być relacyjna baza danych. To, jakiej bazy danych użyjemy do trwałego zapisu danych, praktycznie nie ma żadnego wpływu na wydajność IMDG. Stanowi ona jedynie zabezpieczenie na wypadek awarii takiej liczby serwerów, podczas której możliwa jest utrata części danych. Wyobraźmy sobie sytuację, gdy jeden (lub więcej) serwer ulega awarii, na skutek czego w pamięci klastra zaczyna brakować danych. W takim wypadku podczas pierwszej próby dostępu do tych danych następuje ponowne wczytanie danych do klastra. Przy kolejnej próbie dostępu dane ponownie będą się znajdowały w pamięci klastra. Takie zestawienie IMDG wraz z zapisem danych na dysk daje nam praktycznie stuprocentowe zabezpieczenie przed utratą danych (rys. 1).



Rys. 1. Asynchroniczne kopiowanie danych z klastra
Fig. 1. Asynchronous persistence

Ważną cechą, o której należy wspomnieć, przedstawiając technologię In-Memory Data Grid, jest *near cache*. Jest to hybrydowe, dwupoziomowe podejście do dostępu do danych znajdujących się w klastrze. Wykorzystując w danej konfiguracji *near cache*, decydujemy się na wydzielenie w każdym węźle specjalnej przestrzeni, w której będą przechowywane naj-

częściej wykorzystywane dane. Dzięki temu praktycznie do zera spada opóźnienie przy odczycie najczęściej używanych danych, a jednocześnie nie traci się możliwości odpytywania klastra o dane fizycznie przechowywane w innym węźle [1].

3. Praca z In-Memory Data Grid

3.1. Istniejące implementacje In-Memory Data Grid

Jak zostało wspomniane we wcześniejszym rozdziale, In-Memory Data Grid to przede wszystkim zbiór założeń i pomysłów na to, w jaki sposób przechowywać dane i nimi zarządzać. Dzięki temu możliwe jest istnienie wielu odmiennych implementacji IMDG, których twórcy mogą w różny sposób interpretować ogólne założenia.

Wybór konkretnego rozwiązania nie jest łatwy z powodu istnienia na rynku wielu produktów, różniących się w mniej lub bardziej znaczący sposób.

Do najbardziej popularnych implementacji In-Memory Data Grid należą:

- Oracle Coherence – korporacyjna implementacja IMDG, stworzona przez firmę Tangosol. Od 23 marca 2007 roku Coherence jest wydawany i rozpowszechniany przez firmę Oracle [4].
- Hazelcast – jedna z najbardziej popularnych wersji IMDG, dostępna zarówno w wersji open source, jak i w wersji płatnej. W całości napisana w języku Java i w czasie pisania tego artykułu nie wspierała innych języków [8].
- Infinispan – całkowicie darmowa implementacja IMDG, rozwijana i wpierana w ramach społeczności JBoss [6].

Niezależnie od tego, czy wybór padnie na rozwiązanie komercyjne czy darmowe, najbardziej interesujące jest dostępne API, z którego będziemy mogli korzystać, przygotowując własną aplikację. Kolejne podrozdziały zawierają bardziej techniczne wprowadzenie w technologię In-Memory Data Grid. Zaprezentowane zostaną podstawowe możliwości tej technologii wraz z przykładami implementacji.

3.2. Miejsce przechowywania danych

Większość istniejących przykładów IMDG została zaimplementowana w podobny sposób: na podstawie rozproszonej wersji klasy java.util.Map. Wydaje się to bardzo rozsądnym podejściem, które pozwala na przechowywanie wewnątrz klastra danych dowolnego typu.

Po wstawieniu obiektów do mapy są one automatycznie rozrzucane pomiędzy węzły klastra. Zastosowanie mapy jest najbardziej intuicyjne, ponieważ stanowi naturalne zastępstwo

dla relacyjnego sposobu zarządzania danymi. W mapie możemy umieścić dowolny obiekt danego typu, co odpowiada konkretnej encji w relacyjnej bazie danych wraz z relacjami. Możliwość zapisania takiego obiektu pod pewnym unikalnym kluczem może stanowić bezpośrednie nawiązanie do klucza głównego. Przedstawiony kod prezentuje sposób korzystania z rozproszonej mapy:

```
HazelcastInstance hazelcastInstance = Hazelcast.getDefaultInstance();
IMap<String, String> distributedMap = hazelcastInstance.getMap("MAP_NAME");
distributedMap.put("Key", "Value");
String valueFromDistributedMap = distributedMap.get("Key");
```

Kod ten pokazuje, jak proste i intuicyjne jest korzystanie z IMDG na przykładzie implementacji Hazelcast.

3.3. Rozproszone przechowywanie danych

Gdy pewne dane zostaną umieszone w klastrze, automatycznie są one rozprzestrzeniane wewnątrz klastra w sposób zależny od tego, jaka polityka została przyjęta. Zgodnie z przykładami umieszonymi w poprzednim rozdziale, istnieją praktycznie trzy możliwości. Możemy zdecydować się na maksymalne wykorzystanie miejsca, jakie daje nam klaster, jednocześnie ryzykując utratę danych (żadna kopia zapasowa nie będzie przechowywana). Jednak łącznie, można postawić na praktycznie stuprocentowe bezpieczeństwo danych, uznając, że poszczególne węzły są jedynie kopią bezpieczeństwa w razie awarii któregoś z nich. Poza tymi dwoma skrajnymi przypadkami pozostaje jeszcze wariant pośredni, tzn. sytuacja, w której ograniczamy nieznacznie ilość możliwego do wykorzystania miejsca na rzecz bezpieczeństwa. Możliwe jest na przykład takie skonfigurowanie IMDG, dzięki któremu pojedynczy węzeł klastra oprócz przechowywania swoich danych będzie również przechowywał kopie zapasowe danych z dwóch innych węzłów. Dodatkowo można w tym miejscu ustalić, czy przy odczytywaniu danych z klastra dane będą zawsze pobierane od właściciela obiektów, czy również jest możliwe odczytywanie danych z kopii zapasowych, jeżeli w danym momencie jesteśmy połączeni z klastrem, który je akurat przechowuje. Włączenie możliwości czytania z kopii zapasowych zwiększa wydajność i nie grozi odczytywaniem nieaktualnych danych, gdyż obiekty są zapisywane synchronicznie w węźle będącym ich właścicielem oraz w węzłach przechowujących kopie [1].

3.4. Tworzenie klastra

Mówiąc o klastrze, ważną informacją jest sposób, w jaki możemy zmusić poszczególne węzły do połączenia się we wspólną całość. W przypadku implementacji wymienionych w tym artykule istnieją dwie możliwości. Każdy węzeł w klastrze jest rozpoznawalny po swo-

im unikalnym adresie sieciowym, składającym się z adresu IP oraz numeru portu. Numer portu jest bardzo ważny, ponieważ istnieje możliwość stworzenia wielu instancji węzła wewnątrz jednej fizycznej maszyny (mającej jedno IP). W pierwszym przypadku wystarczy stworzenie instancji węzła IMDG. W tym czasie w sieci, w której znajduje się uruchamiany węzeł, przeszukiwane są standardowe adresy oraz porty. Jeżeli instancja natrafi na inny węzeł, automatycznie łączy się z nim w klaster. Jeżeli poszukiwania okażą się bezowocne, węzeł będzie działał jako osobny klaster i czekał na połączenie innych węzłów. Możliwość taka wymusza jednak, aby wszystkie potencjalne węzły znajdowały się w tej samej podsieci, w przeciwnym razie ich połączenie nie będzie możliwe. Druga możliwość to ręczna konfiguracją węzła w taki sposób, aby szukał innych instancji pod niestandardowym adresem i/lub numerem portu. Istnieje możliwość połączenia węzłów znajdujących się w różnych sieciach lokalnych, jednak w takim wypadku przesyłanie danych jest dużo wolniejsze. Podstawowym zastosowaniem połączenia węzłów z różnych sieci jest synchronizacja pomiędzy sobą dwóch lub więcej klastrów lokalnych. Ze względów bezpieczeństwa w obydwóch przypadkach możliwe jest zdefiniowanie hasła, które będzie używane podczas połączenia z klastrem. Ogranicza to możliwości przyłączenia niechcianego węzła [8].

3.5. Zdarzenia zachodzące w klastrze

Bardzo ważną funkcjonalnością, udostępnianą przez różne implementacje, jest możliwość wysyłania wiadomości przez węzły klastra w modelu opublikuj/subskrybuj. Wiadomości przesyłane są wewnątrz całego klastra. W przypadku Hazelcasta subskrypcję wiadomości można osiągnąć przez prostą implementację interfejsu MessageListener. Po rozpoczęciu subskrypcji dany obiekt będzie otrzymywał wiadomości od każdego obiektu publikującego wiadomości, niezależnie od tego, czy już znajdował się w klastrze, czy dopiero do niego dołączył. Wszystkie wiadomości są przekazywane w kolejności, w jakiej zostały opublikowane. W celu publikowania wysłania wiadomości przez klaster wystarczy stworzyć obiekt klasy Topic, a następnie opublikować wiadomość. Listing pokazuje, w jaki sposób sprawić, aby klasa Simple miała możliwość zarówno publikowania, jak i odbierania wiadomości.

```
import com.hazelcast.core.Topic;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.MessageListener;
public class Sample implements MessageListener {
   public static void main(String[] args) {
       Sample sample = new Sample();
       Topic topic = Hazelcast.getTopic ("default");
       topic.addMessageListener(sample);
       topic.publish ("my-message-object");
   }
   public void onMessage(Object msg) {
       System.out.println("Message received = " + msg);
   }
}
```

Wiadomości wymieniane wewnątrz klastra nie muszą być wymuszane jedynie przez obiekty umieszczone w węzłach. Bardzo często zależy nam na informacjach, takich jak przyłączenie nowego węzła do klastra lub nagłe odłączenie jednego lub więcej istniejących węzłów. W wielu przypadkach takie zdarzenia nie powinny pozostać niezauważone lub wymagają specjalnej obsługi po stronie pisanej aplikacji.

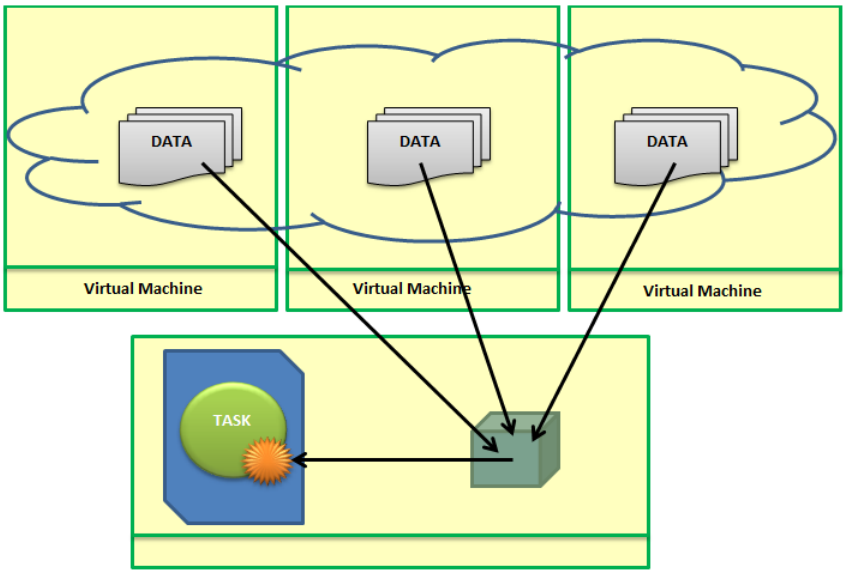
Dodatkowo możliwe jest zdobycie szczegółowych informacji o członkach klastra z poziomu kodu Java. Informacje można uzyskać między innymi o adresie oraz porcie, na jakich działają pozostałe węzły, oraz czy któryś z nich jest tzw. superklientem. Jeżeli dany węzeł jest superklientem oznacza to, że jest podłączony do klastra, ale nie jest właścicielem żadnych danych. Wprawdzie istnienie superklienta nie wnosi do klastra żadnej dodatkowej pamięci, w której mogłyby być przechowywane dane lub ich kopie zapasowe, jednak jego istnienie jest bardzo istotne ze względu na fakt, że ma on możliwość bardzo szybkiego odczytu danych z klastra, tak jakby był jednym z jego węzłów przechowujących dane.

3.6. Rozproszone obliczenia

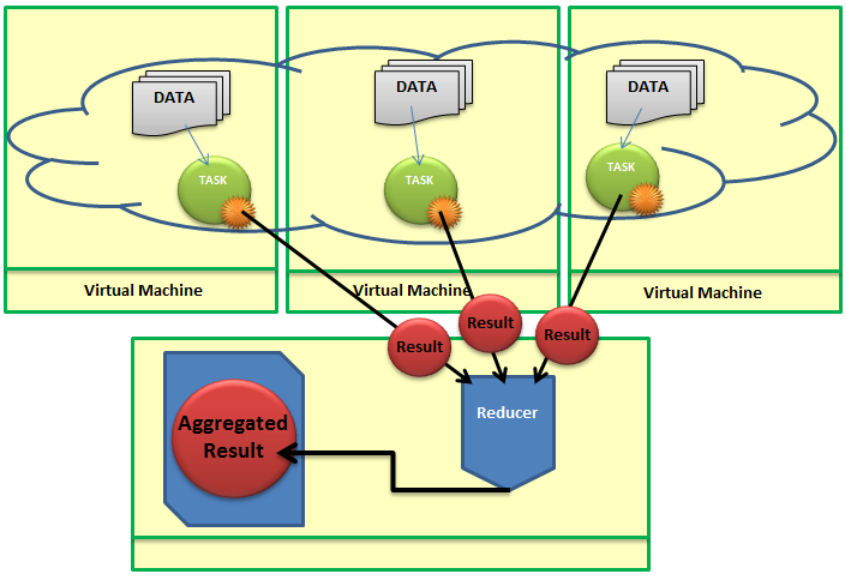
Oczywiste jest, że przechowywanie danych nie jest celem samym w sobie. Każda, nawet najprostsza aplikacja wymaga pewnych operacji wykonywanych na przechowywanych danych. Im więcej danych przechowujemy, tym dłużej trwają obliczenia na nich wykonywane. Sytuacja, która od razu się narzuca, to taka, w której wszystkie dane potrzebne do wykonania obliczeń zostają pobrane do jednego miejsca, a następnie wykonywane są wszystkie obliczenia. Niestety takie podejście ma bardzo istotną wadę, a mianowicie dużą czasochłonność. W pierwszej kolejności tracimy czas na pobieranie danych. W drugiej kolejności czas potrzebny na przetworzenie wszystkich informacji jest zdecydowanie dłuższy niż na przeanalizowanie pewnej ich części. Dlatego też twórcy różnych implementacji IMDG starają się zniwelować te straty, umożliwiając wykonywanie rozproszonych operacji na danych znajdujących się wewnątrz klastra. Zasada jest bardzo prosta. Określamy, jakie działanie zamierzamy wykonać, a następnie każdy z węzłów wykonuje odpowiednie operacje na danych znajdujących się pod jego opieką. Na końcu wyniki uzyskane przez pojedyncze węzły są przekazywane do węzła potrzebującego określonych informacji, gdzie następuje ostateczne przetworzenie danych [2].

Można wymienić bardzo dużo przykładów zastosowania takiej funkcjonalności. Na potrzeby przykładu przyjęto, że wewnątrz klastra przechowane są dane o użytkownikach serwera. Kluczem jest nazwa użytkownika, a wartością jego wiek. Zadanie polega na wyliczeniu średniego wieku użytkownika serwera. Naturalnym odruchem jest pobranie wieku każdego z użytkowników, a następnie obliczenie średniej. Przy korzystaniu z IMDG wywołanie obliczeń wygląda podobnie, jednak fizycznie są dane wstępnie analizowane wewnątrz danego

węzła, a następnie finalne obliczenia następują wewnątrz węzła wywołującego zapytanie. Porównanie opisanej sytuacji ilustrują rysunki 2 oraz 3.



Rys. 2. Obliczenia na podstawie zdobytych danych
Fig. 2. Calculations using collected data



Rys. 3. Postępowanie w przypadku operacji rozproszonych
Fig. 3. Distributed calculations

3.7. Bezpieczeństwo danych

Do tej pory wymienionych zostało wiele cech technologii In-Memory Data Grid, świadczących o tym, że IMDG potrafi zapewnić, poza oczywistą (jak się może wydawać) wydajnością, również bezpieczeństwo powierzonych jej danych. Niezależnie od tego, czy wybierzemy powielanie wszystkich danych wewnątrz klastra, czy zdamy się na niezawodność maszyn, do dyspozycji jest jeszcze jedna możliwość, a mianowicie asynchroniczny zapis danych znajdu-

jących się wewnątrz klastra na trwałym nośniku. Zmiany zachodzące wśród danych umieszczonych wewnątrz klastra są odzwierciedlane przez ich zapis w np. relacyjnej bazie danych, w innej rozproszonej implementacji klastra lub (w najprostszym przypadku) w pojedynczym pliku. Niezależnie od tego, która z wymienionych opcji zostanie wybrana, mamy pewność, że dane, które znajdą na dysku, nie zostaną utracone. Oczywiście nie wszystkie zmiany, którym ulegają dane, są odzwierciedlane na zapasowym nośniku. Zapis danych jest asynchroniczny, przez co zwiększa swoją wydajność. Zamiast najdrobniejsze zmiany wprowadzać od razu, IMDG zbiera o nich informacje i w momencie mniejszego obciążenia węzła zapisuje całą serię zmian za jednym razem. Z reguły czas zwłoki jest konfigurowalny w poszczególnych implementacjach IMDG oraz jest on liczony w sekundach. Jest to bardzo rozsądne podejście, ponieważ nietrudno wyobrazić sobie sytuację, w której jedna informacja może zmienić się wielokrotnie w niewielkim odstępie czasu. Dzięki temu, zamiast wprowadzania kilku mniej istotnych zmian pośrednich, zapisywany jest jedynie końcowy stan danego obiektu. Jeżeli w pewnym momencie dane z pamięci klastra zostaną utracone, przy próbie pierwszego dostępu zostaną one wczytane z dysku i ponownie umieszczone wewnątrz klastra. Najbardziej intuicyjnym rozwiązaniem wydaje się zastosowanie w tym celu innego rozproszonego klastra przechowującego swoje dane na dysku, a nie w pamięci. Umożliwia to między innymi ujednolicenie podejścia konfiguracyjnego w obydwu przypadku, np. dzięki przyporządkowaniu węzłów w stosunku jeden do jeden. Daje nam to możliwość stworzenia lustrzanego odbicia klastra IMDG przechowywanego na dysku twardym [5]. Jedną z bardziej znanych implementacji takiego klastra może być Cassandra, rozpowszechniana na licencji Apache, która z powodzeniem znajduje swoje zastosowanie w aplikacjach, takich jak Facebook [10] czy Twitter.

3.8. Near Cache

Dane umieszczone w klastrze, takim jak np. Hazelcast, są rozprowadzane pomiędzy wszystkie jego węzły. Bardzo często dochodzi do sytuacji, kiedy wielokrotnie odczytywane są te same dane. W przypadku gdy dane te znajdują się w naszym węźle, problem nie istnieje. Inaczej jest w przypadku, kiedy interesujące informacje muszą być pobierane z innego węzła za pośrednictwem sieci. W takiej sytuacji wydajność zdecydowanie spada. Aby utrzymać ją na odpowiednim poziomie, należy ustawić Near Cache dla mapy, która jest najczęściej wykorzystywana. Niestety korzyści uzyskane przez zastosowanie w mapie funkcjonalności Near Cache mają swoją cenę. Zastanawiając się nad zastosowaniem Near Cache'u, należy przede wszystkim rozważyć następujące problemy:

- wirtualna maszyna Java będzie zmuszona przechowywać dodatkowe dane na rzecz działającego Near Cache, co niewątpliwie zwiększy zużycie pamięci, która mogłaby zostać przeznaczona na nowe dane,
- Near Chache łamie zasadę silnej spójności danych. Niestety w pewnym momencie może się okazać, że dane odczytywane z Near Cache są już nieaktualne [7].

4. Aplikacja

Stworzona aplikacja jest przykładem sieciowej gry w Pokera, stworzonej na podstawie technologii In-Memory Data Grid. Aplikacja ma za zadanie zaprezentowanie możliwości wykorzystania opisywanej w artykule technologii w celach do tej pory nierealizowanych przy jej użyciu. Przed przystąpieniem do realizacji projektu przyjęto następujące założenia:

- aplikacja będzie umożliwiać grę maksymalnie 10 graczom (ograniczenia odmiany Texas Holdem),
- komunikacja wewnątrz aplikacji będzie oparta wyłącznie na rozwiązaniach z implementacji IMDG,
- aplikacja będzie wykorzystywała klaster Hazelcast jako implementację IMDG,
- wszystkie informacje wykorzystywane w grze, tj. całkowita pula, liczba żetonów graczy, karty znajdujące się na stole, karty na rękach graczy, będą przechowywane wewnątrz klastra IMDG,
- w trakcie rozgrywki dowolna liczba graczy będzie mogła odejść z gry oraz do niej dołączyć (udział w grze od nowego rozdania) bez przerywania trwającego rozdania lub restartu aplikacji,
- aplikacja będzie w całości napisana w języku Java.

Na potrzeby realizowanej aplikacji został wykorzystany klaster mający standardową konfigurację.

Gra Poker została wybrana nieprzypadkowo, w celu zaprezentowania możliwości technologii In-Memory Data Grid. Najważniejszym jej elementem jest bowiem konieczność uczestniczenia w grze dodatkowej osoby, zwanej krupierem. To ona decyduje o przebiegu rozgrywki. Korzystając z architektury klient-serwer, bardzo łatwo przydzielić odpowiednie role. Serwer pełni rolę krupiera dzięki możliwości komunikacji ze wszystkimi klientami (graczami). Teoretycznie w przypadku gry Poker architektura klient-serwer jest najlepszym możliwym rozwiązaniem. Przy pomocy aplikacji stworzonej na potrzeby niniejszego artykułu autorzy starają się pokazać możliwość alternatywnego wykorzystania technologii In-Memory Data Grid.

Najważniejszy problem, który należało rozwiązać, to pojedynczy punkt awarii. Dzięki wykorzystaniu klastra IMDG jest to absolutnie możliwe, jednak nasuwa się inny problem,

czyli: w jaki sposób zastąpić „serwer” w roli krupiera? Na szczęście wszystkie potrzebne informacje, z których korzysta krupier, przechowywane są wewnątrz klastra. Oznacza to, że w danym momencie jeden z węzłów pełni zarówno rolę gracza, jak i krupiera. W przypadku awarii takiego węzła rolę krupiera może przejąć kolejny gracz dzięki dostępowi do wszelkich potrzebnych danych (rys. 4).

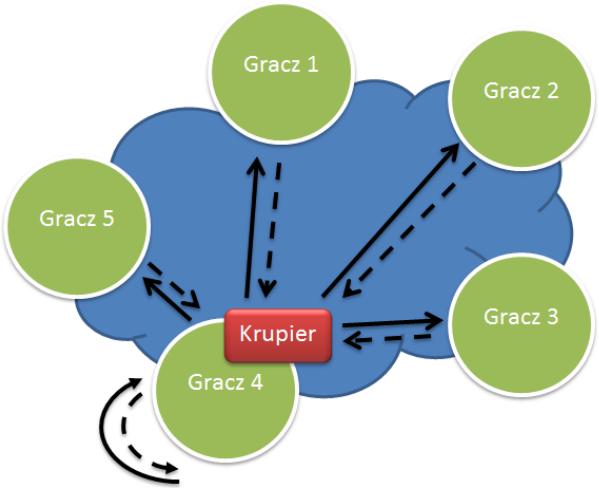
Rys. 4. Przejęcie przez kolejnego gracza roli krupiera
Fig. 4. Getting new dealer from all players.

Podczas rozpoczęcia gry pierwszy gracz, który zakłada klaster, staje się krupierem. Pozostałe węzły pełnią rolę zwykłych graczy. Po rozpoczęciu gry krupier wpisuje do klastra następujące informacje: listę graczy biorących udział w bieżącym rozdaniu, podstawową stawkę gry (tzw. duża w ciemno) oraz id gracza, który aktualnie powinien podjąć akcję, oraz gracza, który powinien zakończyć licytację. Dzięki nasłuchiwaniu na zdarzenia zachodzące w klastrze wszystkie węzły uaktualniają te informacje u siebie. Jeżeli konkretny węzeł wykryje, że to właśnie jego kolej na ruch, dokonuje odpowiedniej akcji i za pomocą klastra przekazuje sterowanie ponownie do dealera. Oczywiście gracz pełniący funkcję krupiera również bierze udział w grze, a więc musi sam do siebie wysłać sygnał o konieczności wykonania określonej akcji. Rola krupiera ogranicza się do przekazywania sterowania pomiędzy kolejnych graczy, rozdawania kart oraz wyłaniania zwycięzcy (rys. 5).

W każdym momencie zarówno krupier, jak i pozostali gracze uaktualniają dane w klastrze, dzięki czemu podczas odejścia któregokolwiek z nich dane będą w dalszym ciągu dostępne dla graczy uczestniczących w rozgrywce. Wyłonienie kolejnego gracza, który zastąpi odchodzącego krupiera, odbywa się na zasadzie „kto pierwszy, ten lepszy”. Brak krupiera komunikowany jest specjalnym zdarzeniem wewnątrz klastra. Węzeł, który jako pierwszy otrzyma wiadomość i wpisze do klastra swoje id, zostaje nowym krupierem.

Jeżeli w pewnym momencie rozgrywki grę opuści gracz niebędący krupierem, informacja o tym zdarzeniu również zostanie obsłużona. Dzięki temu, nawet w przypadku gdy z rozgrywki zrezygnuje gracz aktualnie podejmujący jakąś akcję, sterowanie zostanie ponownie

przekazane krupierowi i gra będzie mogła być kontynuowana. Zapobiega to sytuacji, w której wszystkie węzły są bezczynne i gra nie ma możliwości kontynuacji. W trakcie trwania rozdania możliwe jest przyłączenie się do gry nowych graczy, jednak wezmą oni udział w rozgrywce dopiero podczas nowego rozdania.



Rys. 5. Przepływ sterowania podczas gry
Fig. 5. Control flow of the game.

5. Podsumowanie

W dobie ciągłej walki z czasem oraz nastawieniu na jak najwyższą wydajność można stwierdzić, że technologia IMDG jest rozwiązaniem wartym rozważenia. Przechowywanie danych w pamięci, poza twardym dyskiem, zapewnia szybki dostęp do danych. Dodatkowo funkcjonalności dostarczane przez istniejące implementacje IMDG pozwalają na wygodne zarządzanie klastrem, bezpieczeństwo danych oraz wiele ułatwień sprawiających, że również praca z tą technologią jest bardzo wygodna. Dzięki temu, że na rynku dostępnych jest wiele różnych implementacji IMDG, począwszy od tych darmowych, a skończywszy na komercyjnych rozwiązaniach, każdy może wykorzystać inne podejście do problemu przechowywania danych we własnych warunkach.

Oczywiste jest, że technologia In-Memory Data Grid nie jest najlepszym rozwiązaniem do każdego rodzaju problemu. Z pewnością istnieje wiele przykładów aplikacji, wymagających wysokiej skalowalności, w których zastosowanie IMDG nie jest dobrym pomysłem. Jednak, jak można stwierdzić na przykładzie stworzonej aplikacji, nie należy ograniczać możliwości jej zastosowania tylko do składowania danych. Możliwe jest zupełnie nowatorskie spojrzenie na technologię In-Memory Data Grid, która wcale nie musi kojarzyć się tylko z alternatywą dla relacyjnej bazy danych.

BIBLIOGRAFIA

1. Seović A., Falco M., Peralta P.: Oracle Coherence 3.5. Birmingham 2010.
2. Kuczek K.: In-Memory Data Grid [dostęp 10 września 2011], http://ai.ia.agh.edu.pl/wiki/_media/pl:dydaktyka:ztb:2010:projekty:nosql:data_grid_-_dokumentacja_kacper_kuczek_.pdf.
3. Purdy C.: Defining a Data Grid [dostęp 10 września 2011], http://www.jroller.com/cpurdy/entry/defining_a_data_grid.
4. Oracle – Oracle Press Release. Oracle Buys In-Memory Data Grid Leader Tangosol [dostęp 3 września 2011], http://www.oracle.com/us/corporate/press/015758_EN.
5. Java Developer's Network – Skalowanie aplikacji cz. I – front-end. [dostęp 3 września 2011], http://jdn.pl/node/1524.
6. JBoss Community – Infinispan [dostęp 10 września 2011], http://www.jboss.org/infinispan.
7. Oracle Coherence Developer's Guide – Release 3.7.1 [dostęp 10 września 2011], http://download.oracle.com/docs/cd/E24290_01/coh.371/e22837/toc.htm.
8. Hazelcast Documentation – version 1.9.4, 6 września 2011 [dostęp 10 września 2011], http://hazelcast.com/docs/1.9.4/manual/single_html/.
9. Deploying In-Memory-Data-Grid on the Cloud Gigaspaces [dostęp 10 września 2011], http://www.gigaspaces.com/wiki/display/CCF/Deploying+In-Memory-Data-Grid+on+the+Cloud.
10. Avinash Lakshman. Cassandra – A structured storage system on a P2P Network [dostęp 12 września 2011], http://www.facebook.com/note.php?note_id=24413138919&id=9445547199&index=9.

Wpłynęło do Redakcji 16 stycznia 2012 r.

Abstract

In-Memory Data Grid is one of the examples of the approach to the data management. It finds its application mainly in cases requiring high performance as well as scalability and for these purposes it has been created. With success, however, it is also suitable for solutions much smaller, even personal. In-Memory Data Grid is a grid of data, which stores data in memory, in order to obtain the highest possible performance. This solution is characterized by data redundancy – by keeping copies of information synchronized with individual servers –

used to obtain greater flexibility and ensure data availability in case of failure of one of the servers. Data stored within this grid they are objects used in the application. These objects require, among other things, a short access time, high capaity, predictable scalability and constant availability. These objects are usually created in the object-oriented programming languages such as Java, C #, or Ruby. The most popular implementations of In-Memory Data Grid include: Oracle Coherence, Hazelcast, Infinispan. The article discusses the various configuration options of this type of solutions. The issues were presented on the example application created with the use of cluster Hazelcast.

Adresy

Krzysztof CZAJKOWSKI: Politechnika Krakowska, Wydział Fizyki, Matematyki i Informatyki, Instytut Teleinformatyki, ul. Warszawska 24, 31-155 Kraków, Polska, kc@pk.edu.pl.
Damian PEZDA: Politechnika Krakowska, Wydział Fizyki, Matematyki i Informatyki, Instytut Teleinformatyki, ul. Warszawska 24, 31-155 Kraków, Polska, damian.pezda@op.pl.