STUDIA INFORMATICA 2003
Volume 24 Number 4 (56)

Katarzyna HARĘŻLAK
Politechnika Śląska, Instytut Informatyki

PORÓWNANIE WŁAŚCIWOŚCI WIRTUALNIE WSPÓŁDZIELONEJ PAMIĘCI OPARTEJ NA MODELU LINDY ORAZ JAVASPACES

Streszczenie. W opracowaniu przedstawiona została analiza możliwości wykorzystania metod zaimplementowanych w środowisku wirtualnie współdzielonej pamięci opartej na modelu Lindy w systemie JavaSpaces. Metody te dotyczą sposobu zarządzania rozproszoną bazą danych, a w szczególności realizacją rozproszonych transakcji. Rozważono możliwość wykorzystania systemu JavaSpaces do asynchronicznego uaktualniania kopii danych.

Słowa kluczowe: rozproszone bazy danych, wirtualnie współdzielona pamięć

THE COMPARISON OF VIRTUAL SHARED MEMORY FEATURES BASED ON LINDA AND JAVASPACES MODELS

Summary. The analysis of possibilities of applying methods, which are implemented in virtual shared memory based on Linda model in JavaSpaces system, are presented in this paper. These methods are concerned with the way of distributed database management specially distribution transaction processing. The possibilities of using JavaSpaces system in asynchronous replicated data update are also considered.

Keywords: distributed database, virtual shared memory

1. Wstęp

Niniejsze opracowanie zawiera opis badań stanowiących kontynuację prac związanych z rozwojem metod zarządzania rozproszoną bazą danych. Prace te prowadzone były w ramach budowanego przy współudziale autorki Eksperymentalnego Systemu Zarządzania Rozproszoną Bazą Danych [3, 4]. W systemie tym, wykorzystującym wirtualnie

współdzieloną pamięć (system Paradise – model Lindy), [2], zaimplementowane zostały algorytmy wyszukiwania danych oraz różne strategie realizacji rozproszonych transakcji.

Przedmiotem obecnych badań była analiza możliwości wykorzystania tych rozwiązań z użyciem innych narzędzi. Analizie takiej poddane zostały narzędzia Javy, do których należy między innymi usługa JavaSpaces.

2. JavaSpaces a model Lindy

Usługa JavaSpaces [8, 9,10] jest narzędziem bliźniaczo podobnym do systemu Linda, reprezentującym model systemu ze współdzieloną pamięcią. Jest on systemem rozproszonych obiektów, umożliwiającym dynamiczną komunikację, koordynację oraz współdzielenie obiektów pomiędzy klientami i serwerami zbudowanymi w oparciu o technologię Javy. Wirtualna przestrzeń JavaSpaces służy do wymiany żądań dotyczących wykonania określonych zadań oraz informacji o ich realizacji w aplikacjach rozproszonych. Umożliwia ona tworzenie i przechowywanie obiektów.

JavaSpaces jest częścią pakietu JINI, bezpośrednio bazującego na technologiach RMI (Remote Method Invocation) oraz Object Serialization (serializacja jest mechanizmem umożliwiającym zapis lub odczyt obiektów ze strumienia). System JINI korzysta z RMI jako bazy do zdalnej komunikacji. Obydwie technologie wbudowane są w standardowy pakiet JDK (Java Development Kit).

2.1. Struktury danych

Ogólna zasada korzystania z wirtualnie współdzielonej przestrzeni (ang. Virtual Shared Memory – VMS) opiera się na tworzeniu i wymianie obiektów. W przypadku systemu Paradise [6] obiektami takimi są *krotki* (ang. tuples). Stąd w systemie tym wirtualnie współdzielona przestrzeń określana jest jako przestrzeń krotek (TS). Krotka jest sekwencją pozycji (maksymalnie może ich być 16). Reprezentowana jest przez listę pól zawartych w nawiasach okrągłych, oddzielonych przecinkami. Polami takimi mogą być stałe numeryczne, tekstowe lub zmienne.

Przykładowymi krotkami w systemie Paradise są:

- ("test1", 1),
- ("test2", i), gdzie *i* jest zmienną typu integer,
- ("test3", buf), gdzie *buf* jest zmienną typu char.

Do typów danych, które można wykorzystać w polach należących do krotki, należą:

- int, long, short, char,
- float, double,
- struct,
- union,
- tablice powyższych typów, włączając w to tablice wielowymiarowe.

W przestrzeni JavaSpace obiekty przechowywane są w formie *wpisów* (ang. entries). Każda przestrzeń JavaSpace zawiera jedynie wpisy, będące obiektami typu Entry, to znaczy takie, które implementują interfejs Entry (net.jini.core.entry.Entry). Wpisy są zbiorem referencji do innych obiektów.

Obiekty Entry charakteryzują się następującymi właściwościami:

- muszą być typem klasy publicznej,
- wszystkie pola wewnętrzne muszą być publiczną referencją do obiektów; wykorzystanie typów elementarnych musi być zastąpione klasami zawijaczy tych typów, tzn. Integer, Float, Double, Character,
- mogą mieć dowolną liczbę metod i konstruktorów,
- mogą zawierać referencje do tych samych obiektów,
- muszą mieć bezparametrowy publiczny konstruktor.

W przypadku wpisów zawierających tablice należy korzystać z klas kolekcji (pakiet *java.util*, klasy *Vector, ArrayList*, etc.)

Przykład:

```
public class Wpis implements Entry {
        public String pole1;
        public Integer  pole2;
        public Float pole3;
        public Wpis() {
                pole1=null;
                pole2=null;
                pole3=null; }
        public Wpis(String pole1, Integer pole2, Float pole3){
                this.pole1=pole1;
                this.pole2=pole2;
                this.pole3=pole3;}

        public String toString() {
                return pole1; }
        }
```

2.2. Podstawowe operacje

Do manipulowania obiektami (zapis, odczyt) przygotowane zostały odpowiednie operacje. W systemie Paradise są to *in, inp* – pobranie krotki z przestrzeni krotek, *rd, rdp* – nieniszczący odczyt krotki oraz *out* – zapis krotki do przestrzeni. Natomiast w JavaSpaces są to odpowiednio: *take, takeIfExists, read, readIfExists, write* oraz dodatkowa operacja *notify* powiadomienia o pojawieniu się krotki określonego typu.

Operowanie na obiektach wirtualnie współdzielonej pamięci obu systemów opiera się na dopasowaniu tych obiektów do wzorca. W przypadku systemu Paradise wzorzec, podobnie jak krotka, zawiera sekwencje pól. Wśród nich mogą być takie, które posiadają wartości i wyrażenia (parametry aktualne) lub nazwy zmiennych, do których kopiowane są wartości z pól dopasowywanej krotki (parametry formalne). Takie pozycje powinny być poprzedzone znakiem „?". O dopasowaniu krotki do wzorca mówimy, jeżeli spełnione są następujące warunki:

- wzorzec i krotka zawierają tę samą liczbę pól,
- typy, wartości i długości pól z parametrami aktualnymi są takie same, jak w badanej krotce,
- typy parametrów formalnych wzorca są zgodne z typami odpowiadających im pól w dopasowywanej krotce.

W systemie JavaSpaces każdy wpis (Entry) posiada jedno lub więcej pól, które są wykorzystywane w procesie dopasowywania nadchodzących zgłoszeń. Wszystkie zgłoszenia czytania, pobrania bądź powiadamiania o pojawieniu się wpisu zawierają wzorzec, do którego dopasowywany jest wpis. Mówimy, że obiekty wzorca i wpisu są zgodne, jeśli:

- są tego samego typu,
- wszystkie pola wzorca, nie mające wartości *null*, mają takie same wartości, jak odpowiadające im pola wpisu; przekazanie wartości *null* jako referencji do obiektu wzorca powoduje pominięcie fazy dopasowania – odczytywany jest wtedy dowolny wpis znajdujący się w przestrzeni JavaSpace.

Do badania identyczności wykorzystywana jest metoda *equals* klasy *java.lang.Object*.

W opisie pełnej postaci operacji systemu Paradise wykorzystano symbol *ts* jako oznaczenie konkretnej przestrzeni krotek, symbol @ wiąże operację z przestrzenią *ts*, natomiast dla systemu JavaSpaces jest to zmienna o nazwie *space*.

2.2.1. *Odczyt z przestrzeni*

System Paradise

Operacja *rd @ ts (wzorzec)*, *rdp @ ts (wzorzec)* poszukuje takiej krotki, która będzie zgodna ze wzorcem będącym jej argumentem. Gdy uda się ją odnaleźć, wszystkie parametry formalne wzorca zostają zastąpione wartościami z krotki. Krotka pozostaje w przestrzeni krotek, a dzięki temu jest ona dostępna dla innych działających aplikacji.

Może zdarzyć się, że w TS istnieje więcej krotek pasujących do wzorca operacji *rd*. W takiej sytuacji odczytana jest dowolna z nich. Natomiast, jeżeli w obszarze poszukiwań nie istnieje żaden obiekt spełniający określone warunki, wtedy operacja:

- *rd* zawiesza aplikację do czasu nadejścia odpowiedniej krotki,
- *rdp* nie zatrzymuje działania aplikacji, lecz zwraca wartość 0.

Przykład – dla zbioru krotek znajdujących się w przestrzeni *ts*

1. ("test1", 1), 2. ("test1", 2), 3. ("test1", 1, 10)

operacja

- rd @ ts ("test1", 1) – odczyta krotkę oznaczoną numerem 1,
- rd @ ts ("test1", ?i) – odczyta dowolną z krotek 1 i 2 (i jest zmienną typu integer),
- rd @ ts ("test1", 8) – spowoduje zatrzymanie aplikacji.

System JavaSpaces

Operacje odczytu poszukują w przestrzeni JavaSpace *wpisów* pasujących do wzorca.

public Entry **read**(Entry tmpl, Transaction txn, long timeout)
public Entry **readIfExists**(Entry tmpl, Transaction txn, long timeout)

W przypadku gdy zostanie znaleziony odpowiedni wpis, zwracana jest jego kopia; jeśli nie, obie metody zwracają wartość *null*. Różnica pomiędzy *read* i *readIfExist* polega na tym, że pierwsza z metod jest metodą blokująca, tzn. zwracającą sterowanie dopiero w przypadku, gdy zostanie znaleziony odpowiedni wpis lub zostanie przekroczony maksymalny czas oczekiwania (parametr timeout). Natomiast metoda *ReadIfExist*, w przypadku gdy nie ma *wpisu* pasującego do wzorca, od razu zwróci wartość *null*. Parametr timeout dla tej metody oznacza maksymalny czas, przez jaki należy czekać na zwolnienie blokad związanych z transakcjami, dla obiektu pasującego do wzorca, który już w momencie wywołania znajduje się w przestrzeni.

Przykłady:

```
//Wpisy zostaną dopasowane
wpis.pole1=null; //rezygnacja z dopasowywania wartości pola pole1
wpis.pole2=new Integer(10); //Pole pole2 musi mieć wartość 10
wpis.pole3=newFloat(1.0); //Pole pole3 musi mieć wartość 1.0
wpis=(Wpis)space.read(wpis, null, defaultLease);
```

2.2.2. *Pobranie z przestrzeni*

System Paradise

Operacje *in @ ts (wzorzec), inp @ ts (wzorzec)* działają na podobnej zasadzie jak operacja *rd* oraz *rdp* z tą różnicą, że dopasowane krotki zostają usunięte z przestrzeni krotek.

JavaSpaces

Podobnie i w tym systemie operacje pobrania:

public Entry **take**(Entry tmpl, Transaction txn, long timeout)
public Entry **takeIfExists**(Entry tmpl, Transaction txn, long timeout)

zachowują się dokładnie tak samo jak operacje odczytu, z tym że dodatkowo usuwają przeczytane *wpisy* z przestrzeni JavaSpace.

2.2.3. *Zapis do przestrzeni*

System Paradise

Operacja ***out @ ts (krotka)*** jest operacją zapisującą krotkę określonego typu do przestrzeni krotek. Wykonanie tej operacji nie powoduje żadnego wyszukiwania, ale wymaga wcześniejszego wyznaczenia wartości wszystkich wyrażeń znajdujących się w polach krotki oraz zastąpienia parametrów formalnych konkretnymi wartościami.

Przykład:

```
i = 5;
j = 6.25;
out @ ts ("test4", i , j, i+4)
```

JavaSpaces

Metoda *write* umieszcza kopię wpisu w przestrzeni JavaSpace.

public Lease **write**(Entry entry, Transaction txn, long lease)

Każda operacja zapisu powoduje utworzenie nowego wpisu, nawet w przypadku, gdy wywołano ją wielokrotnie z referencją do tego samego obiektu Entry. Wywołanie metody *write* zwraca obiekt typu Lease, przechowujący informację o maksymalnym czasie życia wpisu w przestrzeni JavaSpace. Zwykle jest on zgodny z wartością argumentu wywołania *lease*. W przypadku gdy podany czas jest zbyt duży, może on zostać zredukowany. Utworzenie obiektu może powodować wygenerowanie powiadomień dla zarejestrowanych obiektów.

Przykład:

```
//korzystamy z wpisu zdefiniowanego w przykładzie na str. 3.
Wpis wpis=new Wpis("Opis", new Integer(20), new Float(1.0));
space.write(wpis, null, defaultLease);
```

2.2.4. Powiadomienie o utworzeniu obiektu

Ten rodzaj operacji dostępny jest tylko w systemie JavaSpaces. Metoda *notify* pozwala śledzić pojawianie się w przestrzeni JavaSpace wpisów o określonym wzorcu. W momencie, gdy w przestrzeni pojawi się szukany obiekt, generowane jest zdarzenie RemoteEvent i wywoływana jego procedura obsługi *listener*, podana jako parametr wywołania metody *notify*. Argument *lease* określa czas w milisekundach, przez jaki będzie śledzone pojawienie się wpisu. Parametr *handback* to obiekt przekazywany do procedury obsługi zdarzenia jako dodatkowa informacja o źródle rejestracji powiadomienia. Postać wywołania metody:

public EventRegistration **notify**(Entry tmpl, Transaction txn, RemoteEventListener listener,
long lease, java.rmi.MarshalledObject handback).

2.3. Transakcje

W obu omawianych systemach można wprowadzać transakcje, które pozwalają traktować grupę operacji dotyczących wirtualnie współdzielonej pamięci jako całość. Zastosowanie transakcji zmienia zakres dostępu do obiektów, którymi manipulują jej operacje.

1. Każdy obiekt zapisywany do VSM nie jest widoczny poza transakcją, aż do momentu jej zatwierdzenia. W momencie wycofania transakcji wszystkie obiekty utworzone w czasie jej trwania są usuwane.
2. Operacja odczytu może dopasowywać obiekty w ramach całej przestrzeni pod warunkiem, że nie należą one do innych transakcji.
3. Reguły dopasowania pozostają bez zmian.
4. W przypadku wycofania transakcji pobrane obiekty są ponownie wstawiane do przestrzeni.

W systemie Paradise transakcję realizuje się za pomocą operacji ***xaction(), commit*** oraz ***cancel.*** Jeżeli aplikacja wykonująca operację ulegnie awarii, to po czasie określonym w parametrach systemu zostanie przywrócony stan przestrzeni sprzed rozpoczęcia transakcji.

Natomiast w systemie JavaSpaces transakcje tworzymy za pomocą statycznej metody *create* klasy TransactionFactory:

public static Transaction.Created **create**(TransactionManager mgr, long leaseTime).

Pierwszy parametr tej metody to menadżer transakcji, który jest serwisem JINI. Drugi to maksymalny czas trwania transakcji podany w milisekundach. Wszystkie operacje na przestrzeni JavaSpace jako jeden z argumentów wywołania pobierają identyfikator transakcji,

będący obiektem klasy Transaction. Wartość null tego argumentu oznacza powstanie transakcji zawierającej tylko wskazaną operację w trybie automatycznego potwierdzenia (ang. autocommit mode). Transakcję można zakończyć metodą *commit()* lub *abort()*.

Przykłady:

Przykład ilustruje modyfikację zawartości krotki przechowującej numer transakcji. Pobranie krotki ze starą wartością i wprowadzenie nowej wartości odbywa się w ramach transakcji, by nie dopuścić do bezpowrotnej utraty krotki.

```
– system Paradise:
    xaction @ ts();
    in @ ts ("idTrans", ? nTr);
    nTr++;
    out@ ts ("idTrans", nTr) ;
    commit @ ts();

– system JavaSpaces:
    public class Tr implements Entry {
        public String opis;
        public Integer nTr;
        public Tr (){
            opis = "idTrans";
            nTr = null;
        }
        public Tr (Integer nTr){
            opis = "idTrans";
            this.nTr = nTr;
        }

        public Inc(){
            return nTr++;
        }
    }
    Tr tr = new Tr();
    Transaction transaction = TransactionFactory.create(txnManager, defaultLease).transaction;
    tr = (Tr) spaces.take(tr, transaction, defaultLease);
    tr.Inc();
    spaces.write(tr, transaction, defaultLease);
    transaction.commit();
```

3. Implementacja wybranych algorytmów Eksperymentalnego Systemu Zarządzania Rozproszoną Bazą Danych w systemie JavaSpaces

Jako pierwszy zaprezentowany jest dwufazowy protokół zatwierdzania transakcji [1]. Protokół ten realizowany jest pomiędzy modułem *klienta* i modułem *lokalnego serwera* [4]. W systemie Paradise przedstawia się on następująco (w przykładzie z oryginalnego

rozwiązania, dla zwiększenia czytelności, usunięto pola nieistotne z punktu widzenia prezentowanych informacji):

Klient

```
1.  out @ ts ("Prepare", nClienta, nTr);
2.  i = 1;
3.  while (i < serverCount) {
4.    in @ ts("Prep_res", ?i, nClienta, nTr, ?odp);
5.    if (odp == NO)
6.      break;
7.    i++;
    }
8.  if (odp == NO)
9.    out @ ts ("Comm_roll", nClienta, nTr, ROLLBACK);
    else
10.   out @ ts ("Comm_roll", nClienta, nTr, COMMIT);
```

Lokalny serwer

```
1.  rd @ ts ("Prepare", ?nClienta, ?nTr)
  ....// operacje na lokalnej bazie danych
2.  out @ ts ("Prep_res", lserver, nClienta, nTr, YES)
3.  rd @ ts ("Comm_roll", ?nClienta, ?nTr, ?kodkon)
  ...// operacje na lokalnej bazie danych
```

Opis wykorzystywanych zmiennych:

nClienta – numer klienta będącego koordynatorem protokołu,

lserver – numer lokalnego serwera,

serverCount – liczba serwerów biorących udział w transakcji,

nTr – numer transakcji,

nOp – numer operacji w ramach transakcji,

odp – zmienna reprezentująca głos uczestnika transakcji (YES/ NO),

kodkon – kod zakończenia transakcji COMMIT/ROLLBACK.

Klient w punkcie 1 rozpoczyna protokół 2PC, wysyłając zachętę do głosowania i w pętli oczekuje na odpowiedź od lokalnych serwerów (pkt. 3 - 7). Pętla ta jest przerywana, jeżeli od któregoś z serwerów nadchodzi odpowiedź negatywna. W punktach 9 i 10 następuje wstawienie do przestrzeni *ts* krotek z decyzją o sposobie zakończenia transakcji. W tym samym czasie lokalny serwer odczytuje krotkę z zachętą do głosowania i rozpoczyna protokół 2PC z lokalną bazą danych. Następnie wstawia krotkę z odpowiedzią i oczekuje na decyzję kończącą, którą realizuje na lokalnej bazie danych. Lokalny serwer dwukrotnie korzysta z operacji *rd,* ponieważ krotki zachęty i decyzji występują tylko w jednym egzemplarzu. Muszą one zatem pozostać dostępne dla innych lokalnych serwerów.

Implementacja tego algorytmu w systemie JavaSpaces wymaga zdefiniowania następujących klas:

```
public class Stale {
        static final int COMMIT = 0;
```

```
        static final int ROLLBACK = 1;
        static final int YES = 0;
        static final int NO = -1;
}

public class Krotka implements Entry {
        public String opis;
        public Integer nClienta;
        public Integer nTr;
        public Integer decyzja;
        public Integer lserver;
        public Krotka() {
                opis = null;
                nClienta = null;
                nTr = null;
                decyzja = null;
                lserver = null;
        }
        public Krotka(String opis, Integer nClienta, Integer nTr, Integer decyzja, Integer lserver) {
                this.opis =opis;
                this. nClienta = nClienta;
                this. nTr = nTr
                this.decyzja = decyzja;
                this. Lserver = lserver;

        }
        public Integer GetDecyzja(){
                return decyzja;
        }
}
```

Realizacja protokołu 2PC pomiędzy klientem i lokalnymi serwerami w systemie JavaSpaces przedstawia się następująco:

Klient

```
decyzja = YES;
Krotka prep = new Krotka("Prepare", new Integer(1), new Integer(1), null, null);
space.write(prep, null, defaultLease);
i = 1;
while (i < serwerCount){
  Krotka odp = new Krotka ("Prep_res", new Integer(1), new Integer(1), null, new Integer(i));
  odp = (Krotka) space.take (odp, null, defaultLease);
  if (odp.GetDecyzja().intValue() = Stale.NO){
        decyzja = NO;
        break;
  }
  i++;
}
Krotka com_roll = new Krotka();
com_roll.opis = "Com_roll";
com_roll.nTr = prep.nTr;
if (decyzja = NO)
  com_roll. decyzja = new Integer(ROLLBACK);
else
  com_roll.decyzja = new Integer(COMMIT);
space.write (com_roll, null, defaultLease);
```

Lokalny serwer

```
Krotka prep = new Krotka ("Prepare", null, null, null, new Integer(1));
prep = (Krotka) space.read(prep,null,defaultLease);
....// operacje na lokalnej bazie danych
Krotka prep_res = new Krotka("Prep_res", prep.nClienta , prep.nTr, new Integer(Stale.YES), new Integer(1));
space.write(prep_res, null, defaultLease);
Krotka com_roll = new Krotka("Com_roll", prep.nClienta , prep.nTr ,null, null);
com_roll = (Krotka) space.read(com_roll, null, defaultLease);
....// operacje na lokalnej bazie danych
```

Jako drugi prezentowany jest algorytm asynchronicznego uaktualniania kopii danych [5]. Algorytm taki stosowany jest jako uzupełnienie protokołu 2PC dla węzłów, które z różnych przyczyn nie mogły brać udziału w bieżącej transakcji, np. wystąpienie awarii lub przyjęta strategia aktualizacji. W algorytmie tym należy przewidzieć:

- zachowanie bieżącej transakcji do późniejszego wykonania, co jest zadaniem koordynatora transakcji,
- właściwą obsługę kopii danych, które nie brały udziału w protokole 2PC, za co odpowiedzialny jest zarządca kopii; do realizacji tego zadania wykorzystuje się mechanizm wersjonowania danych [5]; globalna wersja danych, odpowiadająca liczbie wszystkich transakcji, znajduje się w przestrzeni krotek. Wprowadza ją tam operacja o postaci:

  out @ ts("wersjaGl", numerTab, nWersji, nClienta);

  lokalne wersje przechowują zarządcy kopii.

Zadania poszczególnych modułów podczas realizacji asynchronicznej aktualizacji danych z wykorzystaniem systemu Paradise przedstawione zostały poniżej.

Klient

```
//w trakcie realizacji transakcji
1. xaction @ts();
2. in@ tsh_distDB_tr ("wersjaGl", numerTab, ?nWersjiGl, ?nClienta);
3. nWersjiGl++;
4. out @ ts("HistoriaOp", numerTab, nWersjiGl, nOp, op, nazwaBD);
   // po zakończonym ewentualnym protokole 2PC
5. licznik_serwerow = 1;
6. while (dla wszystkich tabel biorących udział w transakcji){
7.   out @ tsh ("HistoriaTr", numerTab, nWersjiGl, nTr, nOp, clientID, licznik_serwerow);
8.   out @ tsh_distDB_tr ("wersjaGl", numerTab, nWersjiGl, nClienta);
   }
9. commit @ ts();
```

Zarządca kopii

```
// w przypadku różnicy wersji dla tabeli numerTab
1. rd @ ts("wersjaGl", numerTab, ?nrWersjiGl, ?nClienta)
2. while (wersjaLok <= nWersjiGl){
3.    wersjaLok++;
4.    rd @ ts ("HistoriaTr", numerTab, wersjaLok,  ?nTr, ?nOp, ?clientIDlast, ? licznikserwerow);
5.    i = 1;
6.    while (i<= nOp){
7.       rd @ ts ("HistoriaOp", numerTab, wersjaLok,  i,  ?op, ? lokbdp)
8.       ....// wykonanie operacji na lokalnej bazie danych
9.       i++;
      }
10.   in @ts ("HistoriaTr", numerTab, wersjaLok, nTr, nOp, clientIDlast, ? licznikserwerow);
11.   licznikserwerow++;
12.   out @ts ("HistoriaTr", numerTab, wersjaLok, nTr, nOp, clientIDlast,  licznikserwerow);
   }
```

Klient w trakcie realizacji bieżącej transakcji wstawia do przestrzeni krotek obiekty stanowiące historię transakcji (pkt. 4). Następnie, po jej zakończeniu, dla każdej tabeli umieszcza krotkę, w której opisana jest transakcja (pkt. 7). Ponieważ transakcja może dotyczyć tylko wybranych tabel bazy danych, dlatego wersja danych pamiętana jest na poziomie tabeli. Znalazło to odzwierciedlenie w krotkach tworzących historię transakcji. W punkcie 8 algorytmu następuje uaktualnienie wersji globalnej związanej z daną tabelą. Wersja globalna pobierana jest z TS w celu zablokowania dostępu do tabeli na czas trwania transakcji. Operacje *xaction()* (pkt.1) oraz *commit()* (pkt.10) zakreślają ramy transakcji dotyczącej przestrzeni krotek. Jeżeli nie zakończy się ona powodzeniem, wszystkie zmiany dokonane w TS zostaną wycofane.

Zadaniem zarządcy kopii jest okresowe przeglądanie przestrzeni krotek i poprzez porównanie wersji danych wykrywanie zmian dokonanych na innych kopiach. Uaktualnienie danych możliwe jest dzięki historii znajdującej się w TS.

Wersja dla systemu JavaSpace zawiera fragmenty algorytmu dotyczące kontaktów z wirtualnie współdzieloną pamięcią. Operują one na następujących klasach:

```
public class Wersja implements Entry{
    public String opis;
    public Integer numerTab;
    public Integer nWersji;
    public Integer nClienta;

    public Wersja(){
        opis = null;
        numerTab = null;
```

```
            nWersji = null;
            nClienta = null;
        }
        public Wersja (String opis, Integer nWersji, Integer nClienta){
            this.opis = opis;
            this.numerTab = numerTab;
            this.nWersji = nWersji;
            this.nClienta = nClienta;
        }
        public getWersja(){
            return nWersji;
        }
    }
    public class Historia implements Entry{
        public String opis;
        public Integer numerTab;
        public Integer nWersji;
        public Integer nOp;
        public Historia (){
                opis = null;
                numerTab = null;
                nWersji = null;
                nOp = null;
        }
        public Historia (String opis, Integer numerTab, Integer nWersji, Integer nOp){
                this.opis = opis;
                this.numerTab = numerTab;
                this.nWersji = nWersji;
                this.nOp = nOp;
        }
}

public class HistoriaOp extends Historia {
    public String op;
    public String nazwaBD;
    public HistoriaOp{
        super();
        op = null;
        nazwaBD = null;
    }
    public HistoriaOp (String opis, Integer numerTab, Integer nWersji, Integer nOp, String op, String
                    nazwaBD){
        super (opis, numerTab, nWersji, nOp);
        this.op = op;
```

```
    this.nazwaBD = nazwaBD;
    }
}

public class HistoriaTr extends Historia {
    public Integer nTr;
    public Integer nClienta;
    public Integer licznik;
    public HistoriaTr (){
        super();
        nTr = null;
        nClienta = null;
        licznik = null;
    }
    public HistoriaTr (String opis, Integer numerTab, Integer nWersji, Integer nOp, Integer nTr,
                       Integer nClienta, Integer licznik){
        super (opis, numerTab, nWersji, nOp);
        this. nTr = nTr;
        this.nClienta = nClienta;
        this.licznik = licznik;
    }
}
```

Operacjom systemu Paradise w systemie JavaSpaces przy realizacji aktualizacji asynchronicznej odpowiadają:

Klient

```
xaction @ts();
        //pobranie referencji do obiektu menadżera transakcji
        TransactionManager txnManager=Space.getTransactionManager();
        //Stworzenie nowej transakcji
        Transaction transaction=TransactionFactory.create(txnManager, defaultLease).transaction;
in@ ts ("wersjaGl", numerTab, ?nWersjiGl, ?nClienta);
        Wersja wersjaGl = new Wersja("wersjaGl", new Integer(numerTab), null, null);
        wersjaGl = (Wersja) space.take(wersjaGl, transaction, defaultLease);
out @ ts("HistoriaOp", numerTab, nWersjiGl, nOp, op, nazwaBD);
        HistoriaOp histOp = new HistroiaOp ("HistoriaOp", new Integer (numerTab), new Integer (nWersji),
                            new Integer (nOp), op, nazwaBD);
        space.write(histOp, transaction, defaultLease);
out @ts ("HistoriaTr", numerTab, wersjaLok, nTr, nOp, nClienta, licznikserwerow);
        HistoriaTr histTr = new HistoriaTr("HistoriaTr", new Integer (numerTab), new Integer (wersjaLok),
                            new Integer (nOp), new Integer (nClienta), new Integer (licznikserwerow));
        space.write(histTr, transaction, defaultLease);
commit @ ts();
        transaction.commit();
```

Zarządca kopii

```
rd@ ts ("wersjaGl", numerTab, ?nWersjiGl, ?nClienta);
        Wersja wersjaGl = new Wersja("wersjaGl", new Integer(numerTab), null, null);
```

```
        wersjaGl = (Wersja) space.read(wersjaGl, transaction, defaultLease);
rd @ts ("HistoriaTr", numerTab, wersjaLok, ?nTr, ?nOp, ?nClienta,licznikserwerow);
        HistoriaTr histTr = new HistoriaTr("HistoriaTr", new Integer (numerTab), new Integer (wersjaLok),
                            null, null, null, null);
        histTr  = (HistoriaTr) space.read(histTr, transaction, defaultLease);
rd @ ts("HistoriaOp", numerTab, nWersjiGl, nOp, ? op, ? nazwaBD);
        HistoriaOp histOp =  new HistroiaOp ("HistoriaOp", new Integer (numerTab), new Integer (nWersji),
                             new Integer (nOp), null, null);
        histOp  = (HistoriaOp ) space.read(histOp, transaction, defaultLease);
```

4. Wnioski

Przeprowadzona w artykule analiza wykazała, że opracowane algorytmy dla środowiska wirtualnie współdzielonej pamięci wykorzystującej model Lindy mogą być stosowane w systemie JavaSpaces. Zadanie to ułatwia fakt, że aplikacje korzystające z przestrzeni krotek systemu Paradise mogą być tworzone na platformie Javy, dla której system Paradise zaimplementowany został jako zbiór klas. Aby można było z niego skorzystać, należy go umieścić na ścieżce CLASSPATH. Wynika z tego, że zmiana systemu VSM dla istniejącej aplikacji może polegać na podmianie jej bibliotek, w których znajdują się odwołania do wirtualnie współdzielonej pamięci.

LITERATURA

1. Bernstein P. et al.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
2. Carriero N., Gelernter D., Mattson T., Sherman A.: The Linda Alternative to Message-Passing Systems. Parallel Computing, Vol. 20, 1994, pp. 633-655.
3. Chłopek M., Harężlak K., Josiński H.: Implementacja podstawowych mechanizmów zarządzania rozproszoną bazą danych w eksperymentalnym systemie wykorzystującym model wirtualnie współdzielonej pamięci. ZN Pol. Śl. s. Informatyka, z. 32, Gliwice 1997.
4. Chłopek M., Harężlak K., Josiński H.: Sterowanie współbieżnym dostępem do danych podczas realizacji transakcji rozproszonych – rozwój eksperymentalnego systemu rozproszonej bazy danych. ZN Pol. Śl. s. Informatyka, z. 34, Gliwice 1998.
5. Harężlak K.: Realizacja transakcji na kopiach danych w środowisku wirtualnie współdzielonej pamięci. Studia Informatica vol. 22, no 3 (45), Gliwice 2001.
6. Virtual Shared Memory and the Paradise System for Distributed Computing – A technical White Paper. SCIENTIFIC Computing Associates, Inc. Via Internet: http://www.lindaspaces.com/downloads/vsm.pdf, 1999.

7. Freeman E., Hupfer S.: Ease the Development of Distributed Apps with JavaSpaces. Via Internet: http://www.artima.com/jini/jiniology/js1.html, JavaWorld, November 1999
8. Gorrieri R., Busi N., Zavattaro G.:On the Semantics of JavaSpaces. Technical Report.University of Bologna, Italy.
9. http://sern.ucalgary.ca/Courses/CPSC/547/W2000/webnotes/JavaSpaces/JavaSpaces.htm, 2002.
10. http://java.sun.com/products/javaspaces/, 2002.

Recenzent: Dr hab. inż. Stanisław Wołek Prof. Pol. Rzeszowskiej

Wpłynęło do Redakcji 10 lipca 2003 r.

Abstract

This paper presents comparison of two systems based on the virtual shared memory – Paradise and JavaSpaces systems. The description of the basic functions of both systems enveloping reading, taking and writing object was included in chapter two. The transaction model on virtual shared memory was also compared. The difference between Linda and JavaSpaces models presents chapter two.

The analysis of possibilities of applying methods, which are implemented in virtual shared memory based on Linda model in JavaSpaces system, are presented in this paper. These methods are concerned with the way of distributed database management specially distribution transaction processing. The author discusses the processing of two–phase commitment protocol in Paradise and JavaSpaces systems. The possibilities of using JavaSpaces system in asynchronous replicated data update are also considered.

Adres

Katarzyna HARĘŻLAK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-101 Gliwice, Polska, k.harezlak@zti.iinf.polsl.gliwice.pl