VII Seminarium PLOUG
Warszawa
Marzec 2003

Obsługa transakcji rozproszonych w języku Java

Marek Wojciechowski, Maciej Zakrzewicz

{marek, mzakrz}@cs.put.poznan.pl

Politechnika Poznańska, Instytut Informatyki

Streszczenie

Niniejszy artykuł poświęcony jest zagadnieniom dotyczącym przetwarzania transakcyjnego w aplikacjach języka Java, ze szczególnym naciskiem na obsługę transakcji rozproszonych. W artykule przedstawiono możliwości oferowane w tym zakresie przez standardy JDBC i JTA, oparte o ogólny standard XA. Artykuł opisuje typową architekturę przetwarzania transakcyjnego dla platformy J2EE oraz sposoby realizacji transakcji w standardzie JTA przez aplikacje Java warstwy pośredniej.

1. Wprowadzenie

Przetwarzanie transakcyjne stanowi kluczowy element funkcjonalności dostępnych na rynku systemów zarządzania bazami danych. Jest ono niezbędne do poprawnego działania aplikacji realizujących zadania obejmujące zbiór operacji składowych na bazie danych, stanowiących koncepcyjnie logiczną całość. Przetwarzanie transakcyjne zakłada, że logiczną jednostką interakcji użytkownika z systemem jest transakcja, posiadająca następujące własności: atomowość (ang. *atomicity*), spójność (ang. *consistency*), izolacja (ang. *isolation*) i trwałość (ang. *durability*). Atomowość oznacza, że zbiór operacji składowych transakcji musi być wykonany w całości lub wcale. Własność spójności oznacza, że transakcja przeprowadza bazę danych z jednego stanu spójnego do innego stanu spójnego. Izolacja polega na logicznym odseparowaniu od siebie transakcji współbieżnie wykonywanych w systemie. Stopień separacji współbieżnych transakcji (określany jako tzw. poziom izolacji) może być różny. Ponieważ zwiększenie poziomu izolacji ogranicza współbieżność, w praktyce często wybierany jest poziom izolacji, który pozwala jedynie na uniknięcie szczególnie niebezpiecznych anomalii. Tym poziomem jest poziom READ_COMMITTED, w którym transakcje „widzą" jedynie zmiany dokonane przez zatwierdzone transakcje, a nie widzą efektów działania transakcji, które się jeszcze nie zakończyły (READ_COMMITTED jest domyślnym trybem w Oracle9i, ponadto wspierany jest najwyższy poziom izolacji SERIALIZABLE, nie są w Oracle9i dostępne tryby READ_UNCOMMITTED i REPEATABLE_READ). Trwałość transakcji oznacza, że po jej pozytywnym zakończeniu, jej wyniki muszą zostać utrwalone w bazie danych, nawet w przypadku awarii systemu.

Typowym przykładem operacji wymagającej przetwarzania transakcyjnego jest przelew zadanej kwoty z jednego konta bankowego na drugie. Operacja przelewu stanowi transakcję, wymagającą pozytywnego zakończenia dwóch operacji składowych: „odjęcia" zadanej kwoty z jednego konta i „dodania" tej samej kwoty do drugiego konta. System realizujący przelew musi zapewnić, że zostanie on wykonany w całości lub w ogóle, np. gdy po odjęciu kwoty z pierwszego konta z jakiegoś powodu niemożliwe będzie dodanie tej samej kwoty do drugiego konta, cała transakcja musi być wycofana – przywrócony musi być stan początkowy. W zakresie izolacji system co najmniej musi zapewnić, że inne transakcje nie zobaczą wyniku odjęcia kwoty z pierwszego konta, dopóki nie zakończy się cała transakcja, ponieważ jest to stan niespójny, a ponadto gdyby transakcja została następnie wycofana, byłby to stan który logicznie nigdy nie miał miejsca. System musi oczywiście zapewnić również, że gdy użytkownik otrzyma potwierdzenie dokonania przelewu, wyniki przelewu będą zapamiętane w sposób trwały.

W najprostszym przypadku transakcja obejmuje operacje na jednej bazie danych. Taka prosta transakcja jest obsługiwana lokalnie przez odpowiedni system zarządzania bazą danych (konkretnie przez moduł zarządzania transakcjami wbudowany w system zarządzania bazą danych). W przypadku złożonych aplikacji, możliwe są jednak sytuacje, w których transakcja wykorzystuje dane z wielu baz danych. Transakcja, która modyfikuje dane w więcej niż jednej bazie danych określana jest jako transakcja rozproszona. Przykładowo, gdyby omawiana powyżej operacja przelewu dotyczyła dwóch różnych banków, posiadających własne systemy informatyczne oparte o własne systemy baz danych, realizacja przelewu stanowiłaby transakcję rozproszoną.

Jak można się spodziewać zapewnienie wymaganych własności transakcji, w przypadku transakcji rozproszonych znacząco się komplikuje, w porównaniu z transakcjami działającymi na jednej bazie danych. Mechanizmem opracowanym dla realizacji transakcji rozproszonych jest technika zatwierdzania dwufazowego (ang. *2-phase commit*). Polega ona na realizacji operacji zatwierdzania transakcji w dwóch fazach. W pierwszej fazie, określanej jako faza przygotowania (ang. *prepare*), systemy biorące udział w transakcji przygotowują się do zatwierdzenia lokalnych części transakcji rozproszonej (globalnej). Gdy wszyscy uczestnicy transakcji potwierdzą gotowość do zatwierdzenia transakcji, realizowana jest druga faza – faza rzeczywistego zatwierdzania (ang *commit*). Cały ten proces musi być oczywiście koordynowany. Koordynatorem może być zarządca transakcji

(ang. *transaction manager*) działający w ramach jednego z systemów zarządzania bazą danych, biorących udział w transakcji. Może być nim również „zewnętrzny" zarządca transakcji np. wbudowany w serwer aplikacji.

W przypadku aplikacji tworzonych w języku Java, operacje na bazach danych dokonywane są w standardzie JDBC [1][10] opracowanym przez firmę Sun, powszechnie zaakceptowanym i implementowanym. Obecnie pewnego rodzaju alternatywą dla JDBC jest standard SQLJ [13], oferujący wygodniejszy i mniej podatny na błędy sposób zagnieżdżania operacji SQL w kodzie Java. SQLJ w przeciwieństwie do JDBC jest standardem ISO i ANSI, stanowiąc jedną z części specyfikacji języka SQL. Należy jednak podkreślić, że praktyczne implementacje SQLJ (np. dostarczana przez Oracle [11]) bazują na prekompilacji instrukcji SQLJ do wywołań w standardzie JDBC. Dlatego też omawiając problemy związane z realizacją operacji na bazach danych przez programy Java, można ograniczyć się do standardu JDBC.

Standard JDBC już we wczesnych specyfikacjach umożliwiał realizację transakcji w jednej bazie danych, pod warunkiem, że wykorzystywany system zarządzania bazą danych wspierał przetwarzanie transakcyjne. Wsparcie dla transakcji rozproszonych pojawiło się w wersji JDBC 2.0, wraz z wieloma rozszerzeniami funkcjonalnymi i dotyczącymi zwiększenia efektywności przetwarzania. Najnowsza wersja standardu – JDBC 3.0 w zakresie przetwarzania transakcyjnego wprowadza kolejne, tym razem mniej istotne, ale z pewnością użyteczne rozszerzenia np. wcześniej niedostępną instrukcję SAVEPOINT. Standard JDBC 2.0 dostarcza bazę dla realizacji transakcji rozproszonych w języku Java. W praktyce aplikacje Java realizujące transakcje rozproszone mają postać aplikacji komponentowych (EJB) wykonujących się w środowisku oferującym obsługę transakcji (np. serwer EJB). Realizacja transakcji rozproszonych w takim środowisku wymaga współpracy aplikacji użytkowej, serwera aplikacji, zarządcy transakcji i zarządców zasobów (np. serwerów baz danych). Interfejsy opisujące współpracę „uczestników" transakcji (w szczególności rozproszonej) realizowanej w warstwie pośredniej (na platformie zgodnej z J2EE) definiuje standard JTA [2].

Niniejszy artykuł poświecony jest zagadnieniu przetwarzania transakcyjnego w aplikacjach języka Java, ze szczególnym naciskiem na obsługę transakcji rozproszonych. Artykuł przedstawia rolę i funkcjonalność standardów JDBC i JTA w zakresie przetwarzania transakcyjnego w języku Java, a także związki tych standardów z ogólnymi standardami poświęconymi transakcjom rozproszonym tj. standardami X/Open XA [7] i CORBA OTS [8].

2. Proste transakcje w standardzie JDBC

Tradycyjne aplikacje w języku Java (wykorzystujące jedynie mechanizmy przewidziane przez wczesne specyfikacje standardu JDBC, np. JDBC 1.2), wykonują operacje na bazie danych poprzez obiekt *Connection*, uzyskany w wyniku otwarcia połączenia z bazą danych za pośrednictwem zarządcy sterowników (*DriverManager*). Zarządca sterowników otwiera połączenie korzystając z jednego z zarejestrowanych wcześniej w aplikacji sterowników JDBC. Odpowiedni sterownik wybierany jest na podstawie zawartości łańcucha znaków z opisem sposobu połączenia z bazą. Poniższy fragment kodu ilustruje rejestrację sterownika JDBC, nawiązanie połączenia i wykonanie przykładowego polecenia SQL w bazie danych (dla uproszczenia w przykładach pominięto obsługę mogących wystąpić wyjątków):

```
// rejestracja sterownika Oracle JDBC
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
// otwarcie połączenia z bazą poprzez DriverManager
conn = DriverManager.getConnection(
      "jdbc:oracle:thin:@host1:1521:orcl", "scott","tiger");
// utworzenie i wykonanie instrukcji SQL
Statement stmt = conn.createStatement();
stmt.executeUpdate("DELETE FROM emp WHERE empno = 3981");
```

Powyższy przykład nie zawiera żadnych instrukcji sterujących przebiegiem transakcji. Takie rozwiązanie jest dopuszczalne, ponieważ domyślnie połączenia zwracane przez *DriverManager* pracują w trybie automatycznego zatwierdzania (ang. *auto commit*). W tym wypadku każda wykonywana instrukcja SQL stanowi osobną transakcję i jest automatycznie zatwierdzana. Rozwiązanie to jest o tyle sensowne, że nie wszystkie bazy danych muszą wspierać przetwarzanie transakcyjne. Jeśli jednak aplikacja wymaga realizacji transakcji obejmujących kilka instrukcji SQL, a wykorzystywany system zarządzania bazą danych wspiera transakcje, należy wyłączyć tryb automatycznego zatwierdzania odpowiednią metodą obiektu *Connection* i jawnie kończyć transakcję, co ilustruje poniższy przykład:

```
...
Connection conn = DriverManager.getConnection(...);
// wyłączenie trybu automatycznego zatwierdzania
conn.setAutoCommit(false);
...
// polecenia SQL
...
// zatwierdzenie transakcji
conn.commit();
```

W powyższym przykładzie transakcja kończy się zatwierdzeniem (COMMIT). W przypadku zakończenia transakcji poprzez jej wycofanie (operacja ROLLBACK), należałoby zastąpić wywołanie metody *commit*() na rzecz obiektu *Connection* przez wywołanie metody *rollback*(). Należy zwrócić uwagę, że przy realizacji transakcji w standardzie JDBC nie ma jawnych operacji rozpoczęcia transakcji – nowa transakcja rozpoczyna się w chwili otwarcia połączenia z bazą lub po zakończeniu poprzedniej transakcji.

Wersja 2.0 standardu JDBC (w ramach części specyfikacji określanej jako JDBC Optional Package) wprowadza nowy, zalecany sposób uzyskiwania połączeń z bazą danych w oparciu o tzw. źródła danych (ang. *data source*). Definicje źródeł danych typowo znajdują się na zewnątrz w stosunku do aplikacji, są tworzone przez administratora i dostępne dla aplikacji poprzez serwis nazw, z którego aplikacja korzysta w oparciu o interfejs JNDI [6]. Istnieje również możliwość tworzenia instancji źródeł danych bezpośrednio w aplikacji, jednak w przypadku aplikacji uruchamianych w środowisku serwera aplikacji, preferowane jest rozwiązanie polegające na definiowaniu źródeł danych w plikach konfiguracyjnych aplikacji. Typowy scenariusz korzystania ze źródeł danych w aplikacji wygląda następująco:

1) uzyskanie przez aplikację dostępu do serwisu nazw dla niej przeznaczonego
2) wyszukanie danego źródła danych poprzez jego nazwę (ścieżkę dostępu do źródła ustaloną przez administratora w memencie definiowania źródła i jego rejestracji w serwisie nazw)
3) uzyskanie obiektu *Connection* poprzez źródło danych

Korzystanie ze źródeł danych nie wymaga wielu zmian w aplikacji w stosunku do wersji korzystającej z *DriverManager* – inna jest jedynie sekwencja operacji prowadząca do uzyskania obiektu *Connection*. Korzyści płynące z korzystania ze źródeł danych to po pierwsze uniknięcie zaszywania parametrów połączenia z bazą danych w kodzie aplikacji. Parametry te są częścią definicji źródła danych i typowo znajdują się w plikach konfiguracyjnych aplikacji. Po drugie, połączenie uzyskane poprzez źródło danych może oferować dodatkową funkcjonalność. Standard JDBC przewiduje źródła danych wspierające mechanizm connection pooling i transakcje rozproszone (implementacje takich źródeł danych dostarcza np. Oracle). Mechanizm connection pooling polega na buforowaniu połączeń (utrzymywana jest pewna pula otwartych połączeń, które przydzielane są aplikacjom, zamknięcie połączenia z poziomu aplikacji fizycznie go nie zamyka – połączenie wraca do puli). Mechanizm ten jest transparentny z punktu widzenia aplikacji, a powoduje znaczące skrócenie czasu nawiązywania połączenia z bazą danych. Z kolei wsparcie dla transakcji rozproszonych wiąże się z wspieraniem mechanizmu zatwierdzania dwufazowego (Źródła wspierające transakcje rozproszone są w praktyce rozszerzeniem źródeł wspierających mechanizm connec-

tion pooling). Poniższy fragment kodu ilustruje sposób uzyskania połączenia z bazą danych w oparciu o źródło danych zarejestrowane w serwisie nazw.

```
...
// uzyskanie obiektu reprezentującego kontekst JNDI
Context ctx = new InitialContext();
// wyszukanie źródła danych przez nazwę logiczną
OracleDataSource ds
   = (OracleDataSource)ctx.lookup("jdbc/FinanceDB");
// uzyskanie obiektu Connection
Connection conn = ds.getConnection();
```

Pierwszą operacją w powyższym przykładzie jest uzyskanie przez aplikację dostępu do kontekstu JNDI. Wykorzystany w przykładzie bezargumentowy konstruktor *InitialContext*() zwraca obiekt kontekstu dla domyślnego środowiska kontekstu aplikacji. W przypadku aplikacji uruchamianych na serwerze aplikacji, takich jak serwlety, JSB, EJB, serwer aplikacji tworzy dla aplikacji odpowiedni domyślny kontekst. Aplikacje działające po stronie klienta muszą skorzystać z konstruktora *InitialContext* (*Hashtable env*), któremu należy przekazać tablicę z zestawem parametrów umożliwiającym połączenie z serwisem nazw.

3. Transakcje rozproszone w standardzie JDBC 2.0

Jak wspomniano wcześniej, JDBC 2.0 Optional Package oferuje wsparcie dla transakcji rozproszonych poprzez mechanizm źródeł danych. Wsparcie to jest oparte o ogólny standard XA dla transakcji rozproszonych, będący częścią standardu X/Open [7]. Ten rozdział poświęcony jest implementacji XA w JDBC 2.0, architektura przetwarzania dla transakcji rozproszonych w aplikacjach Java uruchamianych na serwerze aplikacji omówiona zostanie w rozdziale następnym. Należy jednak już w tej chwili podkreślić, że omawiana funkcjonalność XA jest w praktyce implementowana w serwerach aplikacji i izolowana od aplikacji użytkowych.

Ogólnie mówiąc funkcjonalność XA zapewnia zarządcy transakcji koordynację poszczególnych gałęzi transakcji (ang. *transaction branches*) reprezentowanych przez zasoby XA (*XA resource*) i zatwierdzanie lub wycofywanie gałęzi transakcji. Kluczowe elementy XA to:

- **Źródła danych XA** (ang. *XA data sources*) - jedno źródło danych dla jednej bazy danych biorącej udział w transakcji (typowo tworzone na poziomie serwera aplikacji)
- **Połączenia XA** (ang. *XA connections*) - uzyskane ze źródeł danych XA, służące do uzyskania instancji XA resource i połączeń JDBC (jedno połączenie XA reprezentuje jedną sesję w bazie danych i może zwrócić jedną instancję zasobu XA)
- **Zasoby XA** (ang. *XA resources*) - wykorzystywane przez zarządcę transakcji do koordynacji gałęzi transakcji rozproszonej, ich funkcjonalność obejmuje operacje: *start*, *end*, *prepare*, *commit*, i *rollback*
- **Identyfikatory transakcji** (ang. *transaction IDs*) – służące do identyfikacji poszczególnych gałęzi transakcji rozproszonych, każdy z identyfikatorów składa się z części identyfikującej gałąź w obrębie transakcji oraz z części identyfikującej daną transakcję rozproszoną (identycznej dla wszystkich instancji zasobów XA związanych z daną transakcją rozproszoną)

Poniższy przykładowy program ilustruje sposób realizacji transakcji rozproszonej w standardzie JDBC 2.0 (z wykorzystaniem mechanizmów pochodzących ze standardu XA). Jak podkreślono wcześniej, w rzeczywistych aplikacjach warstwy pośredniej, funkcjonalność XA byłaby realizowana przez zarządcę transakcji w ramach serwera aplikacji. Przedstawiony przykład został oparty o jeden z programów demonstracyjnych rozpowszechnianych wraz z serwerem Oracle9i. Dla uproszczenia ukryte zostały w nim szczegóły dotyczące generowania identyfikatorów gałęzi trans-

akcji i operacji SQL składających się na transakcję. Działanie poniższego kodu można podzielić na następujące kroki:

1) Przygotowanie źródeł danych, połączeń i identyfikatorów gałęzi transakcji
2) Rozpoczęcie gałęzi transakcji
3) Wykonanie operacji SQL w poszczególnych gałęziach transakcji
4) Zakończenie gałęzi transakcji
5) Przeprowadzenie zatwierdzania dwufazowego (2-phase commit)

```
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;

class XA_M
{
  public static void main (String args []) throws SQLException
  {
    try
    {
        String URL1 = "jdbc:oracle:thin:@serv1:1521:orcl1";
        String URL2 = "jdbc:oracle:thin:@serv2:1521:orcl2";

        // utworzenie źródeł danych XA
        OracleXADataSource oxds1 = new OracleXADataSource();
        oxds1.setURL(URL1);
        oxds1.setUser("scott");
        oxds1.setPassword("tiger");

        OracleXADataSource oxds2 = new OracleXADataSource();
        oxds2.setURL(URL2);
        oxds2.setUser("scott");
        oxds2.setPassword("tiger");

        // uzyskanie połączeń XA ze źródeł danych
        XAConnection pc1  = oxds1.getXAConnection();
        XAConnection pc2  = oxds2.getXAConnection();

        // uzyskanie połączeń JDBC
        Connection conn1 = pc1.getConnection();
        Connection conn2 = pc2.getConnection();

        // uzyskanie instancji XA Resource
        XAResource oxar1 = pc1.getXAResource();
        XAResource oxar2 = pc2.getXAResource();

        // utworzenie ID gałęzi (z tym samym ID trans. rozproszonej)
        Xid xid1 = createXid(10, 1);
        Xid xid2 = createXid(10, 2);

        // rozpoczęcie gałęzi transakcji
        oxar1.start (xid1, XAResource.TMNOFLAGS);
        oxar2.start (xid2, XAResource.TMNOFLAGS);
```

```
      // operacje SQL w ramach poszczególnych gałęzi transakcji
      doSomeWork1 (conn1);
      doSomeWork2 (conn2);

      // zakończenie gałęzi transakcji
      oxar1.end(xid1, XAResource.TMSUCCESS);
      oxar2.end(xid2, XAResource.TMSUCCESS);

      // 2-phase commit: faza prepare
      int prp1 =  oxar1.prepare (xid1);
      int prp2 =  oxar2.prepare (xid2);

      // określenie wyniku transakcji (commit czy rollback?)
      boolean do_commit = true;
      if(!((prp1 == XAResource.XA_OK) || (prp1 == XAResource.XA_RDONLY)))
         do_commit = false;
      if(!((prp2 == XAResource.XA_OK) || (prp2 == XAResource.XA_RDONLY)))
         do_commit = false;

      // 2-phase commit: commit lub rollback
      if (prp1 == XAResource.XA_OK)
        if (do_commit)
           oxar1.commit (xid1, false);
        else
           oxar1.rollback (xid1);

      if (prp2 == XAResource.XA_OK)
        if (do_commit)
           oxar2.commit (xid2, false);
        else
           oxar2.rollback (xid2);

      // zamknięcie połączeń
      conn1.close(); conn1 = null;
      conn2.close(); conn2 = null;
      pc1.close();   pc1 = null;
      pc2.close();   pc2 = null;

   } catch (Exception) { ... }
}

static Xid createXid(int tid, int bid) throws XAException
{
  ... // utworzenie identyfikatora gałęzi
}

private static void doSomeWork1 (Connection conn)
 throws SQLException
{
  ... // operacje SQL
}

private static void doSomeWork2 (Connection conn)
 throws SQLException
{
  ... // operacje SQL
}
```

4. Przetwarzanie transakcyjne na platformie J2EE - Standard JTA

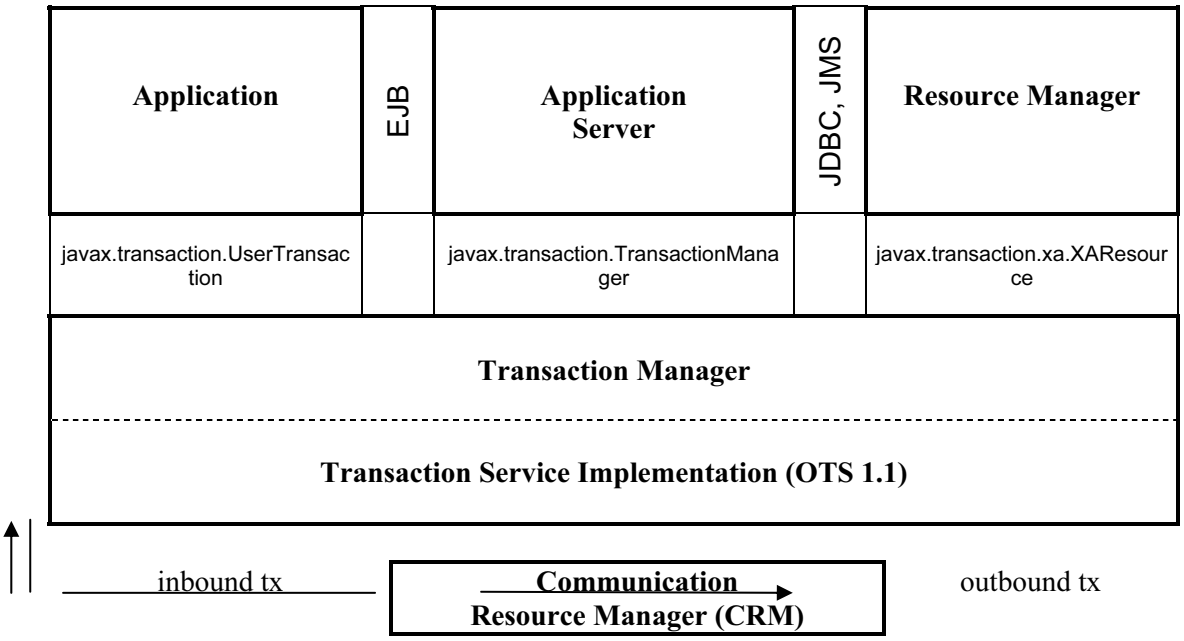
Realizacja transakcji rozproszonych na platformie J2EE wymaga współpracy oprogramowania różnego rodzaju: zarządcy transakcji, serwera aplikacji, zarządców zasobów, aplikacji użytkowej oraz zarządcy komunikacji. Każdy z „uczestników" transakcji odpowiada za określoną funkcjonalność:

- **Zarządca transakcji** (ang. *transaction manager*) dostarcza funkcje i usługi wymagane do wyznaczania początku i końca transakcji, transakcyjnego zarządzania zasobami i synchronizacji
- **Serwer aplikacji** (ang. *application server*) stanowi środowisko uruchomieniowe dla aplikacji z uwzględnieniem zarządzania stanem transakcji (przykładem takiego serwera aplikacji jest serwer EJB)
- **Zarządca zasobów** (ang. *resource manager*) umożliwia aplikacjom dostęp do zasobów. Aplikacje łączą się z zarządcą zasobów poprzez dedykowaną dla danego typu zarządcy zasobów bibliotekę programową. Przykładem zarządców zasobów są serwery relacyjnych baz danych (biblioteką umożliwiającą aplikacji komunikację z zarządcą zasobów jest w tym przypadku odpowiedni sterownik JDBC) lub serwery zarządzania komunikatami (dostępne poprzez interfejs Java Message Service (JMS) [4])
- **Transakcyjna aplikacja**, typowo składająca się z wielu komponentów (np. oparta o standard Enterprise JavaBeans (EJB) [5]), powierza zarządzanie transakcjami serwerowi aplikacji
- **Zarządca komunikacji** (ang. *communication resource manager*) umożliwia współpracę zarządców transakcji poprzez przekazywanie kontekstu transakcji

Standard JTA (Java Transaction API) specyfikuje interfejsy między zarządcą transakcji a innymi „uczestnikami" tj. aplikacją, serwerem aplikacji i zarządcami zasobów (np. serwerami baz danych). Standard JTA oparty jest w dużym stopniu na specyfikacji JDBC 2.0 i standardzie XA. Obejmuje on następujące trzy główne części:

- Interfejs wysokiego poziomu umożliwiający aplikacji transakcyjnej wytyczanie granic transakcji (*javax.transaction.UserTransaction*)
- Interfejs zarządcy transakcji umożliwiający serwerowi aplikacji obsługę wyznaczania granic transakcji dla aplikacji zarządzanych przez dany serwer aplikacji (*javax.transaction.TransactionManager*)
- Mapowanie do języka Java interfejsu XA umożliwiającego transakcyjnemu zarządcy zasobów udział w globalnej (rozproszonej) transakcji zarządzanej przez zewnętrznego zarządcę transakcji (*javax.transaction.xa.XAResource*)

Role powyższych interfejsów definiowanych przez standard JTA pokazuje Rys. 1.



Rys. 1 Rola interfejsów definiowanych przez standard JTA

Standard JTA zakłada współpracę z zarządcą transakcji zgodnym ze specyfikacją Java Transaction Service (JTS) [3]. Standard JTS dotyczy implementacji zarządcy transakcji (ang. *transaction manager*) wspierającego na wysokim poziomie standard JTA, poprzez mapowanie do języka Java specyfikacji OMG Object Transaction Service (OTS) [8] na niskim poziomie. JTS obejmuje opis mechanizmów współpracy zarządców transakcji (komunikacji między nimi).

5. Transakcje JTA w aplikacjach Java

JTA znajduje zastosowanie przede wszystkim w kontekście aplikacji budowanych w oparciu o komponenty Enterprise JavaBeans (EJB). Należy jednak podkreślić, że również aplikacje innego rodzaju (np. serwlety) mogą wykorzystywać standard JTA do specyfikacji transakcji. Z kolei w przypadku komponentów EJB niektórych typów (komponenty sesyjne i komunikatowe) dopuszczalne jest wykorzystanie standardu JDBC do sterowania transakcjami.

Realizacja transakcji JTA z punktu widzenia aplikacji J2EE wymaga zarejestrowania zasobów i wyznaczenia granic transakcji. Od liczby zarejestrowanych zasobów zależy konieczność zastosowania mechanizmu 2-phase commit (gdy aplikacja korzysta z jednego zasobu – wystarczy zatwierdzanie jednofazowe, gdy z więcej niż jednego – konieczne jest zatwierdzanie dwufazowe). Rejestracja zasobu dla aplikacji wymaga konfiguracji odpowiedniego źródła danych w jednym z plików konfiguracyjnym aplikacji (data-sources.xml). Następnie aplikacja po rozpoczęciu transakcji (!) wyszukuje dane źródło danych w serwisie nazw (operacja *lookup* JNDI) i uzyskuje za jego pośrednictwem obiekt *Connection*.

W zakresie wyznaczania granic transakcji, transakcje mogą być zarządzane przez komponent (ang. *bean-managed*) lub przez kontener (ang. *container-managed*). W przypadku transakcji zarządzanych przez komponent, aplikacja wykonuje jawnie operacje rozpoczęcia i zakończenia transakcji poprzez interfejs *UserTransaction* (granice transakcji wyznaczane są programistycznie). Transakcje zarządzane przez komponent dostępne są dla sesyjnych (ang. *session*) i komunikatowych (ang. *message-driven*) komponentów EJB, muszą być wykorzystywane przez serwlety i JSP, nie mogą być stosowane w przypadku encyjnych (ang. *entity*) komponentów EJB. Poniższy fragment kodu ilustruje sposób programistycznego wyznaczania granic transakcji w standardzie JTA za pośrednictwem interfejsu *UserTransaction*:

```
...
Context ctx = new InitialContext();
UserTransaction ut
   = (UserTransaction) ctx.lookup("java:comp/env/UserTransaction");
// rozpoczęcie transakcji
ut.begin();
DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/OracleDS");
// uzyskanie obiektu Connection po (!) rozpoczęciu transakcji
Connection conn = ds.getConnection();
...
// zatwierdzenie transakcji
ut.commit();
...
```

Transakcje zarządzane przez kontener są dostępne dla komponentów EJB (komponenty EJB są uruchamiane w środowisku kontenera EJB w ramach serwera EJB, kontener oferuje m.in. właśnie usługi transakcyjne). W tym wypadku transakcją steruje kontener, na podstawie informacji podanych w sposób deklaratywny w pliku deployment descriptor zawierającym opis instalacji komponentu (informacja o tym czy komponent EJB wykorzystuje transakcje zarządzane przez komponent czy kontener również zawarta jest w tym pliku). Dostępnych jest sześć różnych trybów obsługi transakcji przez kontener (Required, RequiresNew, Supports, NotSupported, Manadatory i Never). Tryb odpowiedni w typowych sytuacjach to Required, nakazujący kontenerowi rozpoczęcie transakcji przed rozpoczęciem wykonywania wywołanej metody komponentu, a kończenie transakcji zaraz po zakończeniu działania metody, chyba że klient wywołujący metodę był w trakcie transakcji – wtedy wykonanie metody odbywa się w ramach tej bieżącej transakcji.

6. Podsumowanie

W artykule przedstawiono zagadnienia dotyczące przetwarzania transakcyjnego w aplikacjach języka Java, ze szczególnym naciskiem na obsługę transakcji rozproszonych. Omówiona została rola i funkcjonalność standardów JDBC i JTA w zakresie przetwarzania transakcyjnego w języku Java. Opisane zostały niskopoziomowe mechanizmy do obsługi transakcji rozproszonych zawarte w standardzie JDBC, typowa architektura przetwarzania transakcyjnego dla platformy J2EE oraz sposoby realizacji transakcji w standardzie JTA przez aplikacje Java warstwy pośredniej.

7. Bibliografia

[1] JDBC Specification (http://java.sun.com/products/jdbc)
[2] Java Transaction API (JTA) Specification (http://java.sun.com/products/jta)
[3] Java Transaction Service (JTS) Specification (http://java.sun.com/products/jts)
[4] Java Message Service (JMS) Specification (http://java.sun.com/products/jms)
[5] Enterprise JavaBeans Specification (http://java.sun.com/products/ejb)
[6] Java Naming And Directory Interface (JNDI) Specification (http://java.sun.com/products/jndi)
[7] X/Open CAE Specification – Distributed Transaction Processing: The XA Specification, X/Open Document No. XO/CAE/91/300
[8] OMG Object Transaction Service (http://www.omg.org/corba/sectrans.html)
[9] Oracle9iAS Containers for J2EE Services Guide, Release 2 (9.0.3), 2002
[10] Oracle9i JDBC Developer's Guide and Reference, Release 2 (9.2), 2002
[11] Oracle9i SQLJ Developer's Guide and Reference, Release 2 (9.2), 2002

[12] Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference, Release 2 (9.0.3), 2002
[13] Information Technology – Database Languages – SQL – Part 10: Object Language Bindings (SQL/OLB), ISO/IEC 9075-10:2000, 2000