Transakcje w systemach Java Enterprise

Korzystanie z systemów kolejkowania w serwerach aplikacji

Systemy kolejkowania w środowisku serwera aplikacji

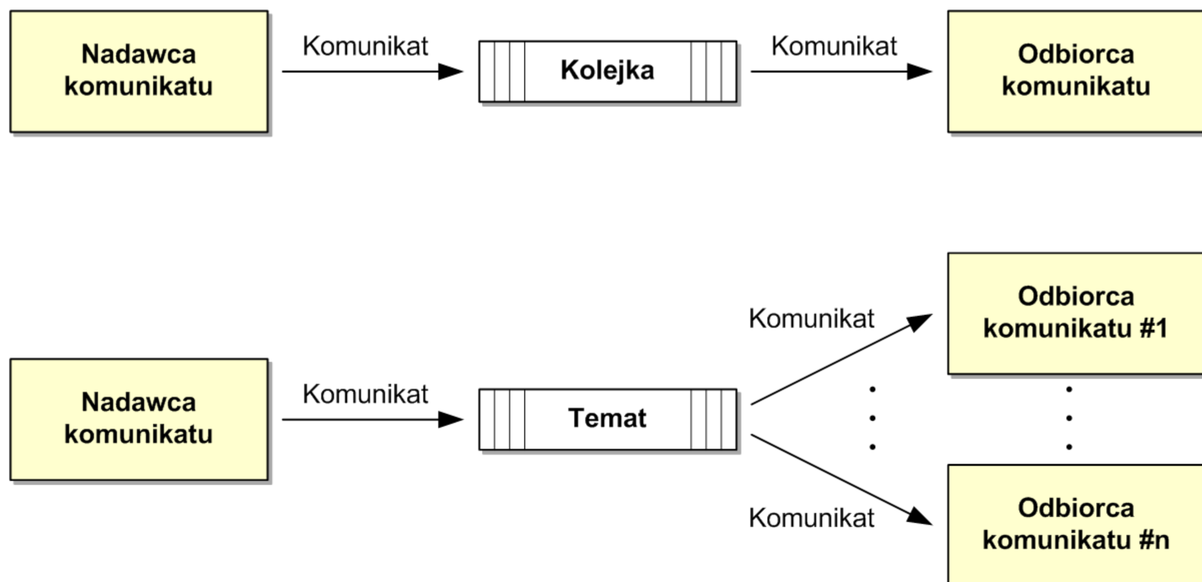
Możliwość korzystania z systemów kolejkowania w aplikacjach Java Enterprise zdefiniowana jest przez specyfikację Java Message Service (JMS) oraz przez specyfikację samej platformy Java Enterprise. Specyfikacje te definiują również zachowania transakcyjne systemów kolejkowania z punktu widzenia programisty aplikacji.

Wsparcie dla systemów kolejkowania zostało wprowadzone od specyfikacji J2EE 1.3, w której serwer aplikacji musiał wspierać JMS w wersji 1.0. Od specyfikacji J2EE 1.4 serwer aplikacji musi wspierać JMS w wersji 1.1. Z punktu widzenia JMS API od 2002 roku nic w tej materii się nie zmieniło. W praktyce oznacza to, że każdy współczesny serwer aplikacji zawiera własną implementację systemu kolejkowania oraz umożliwia podłączenie się do systemów kolejkowania oferowanych przez zewnętrznych dostawców.

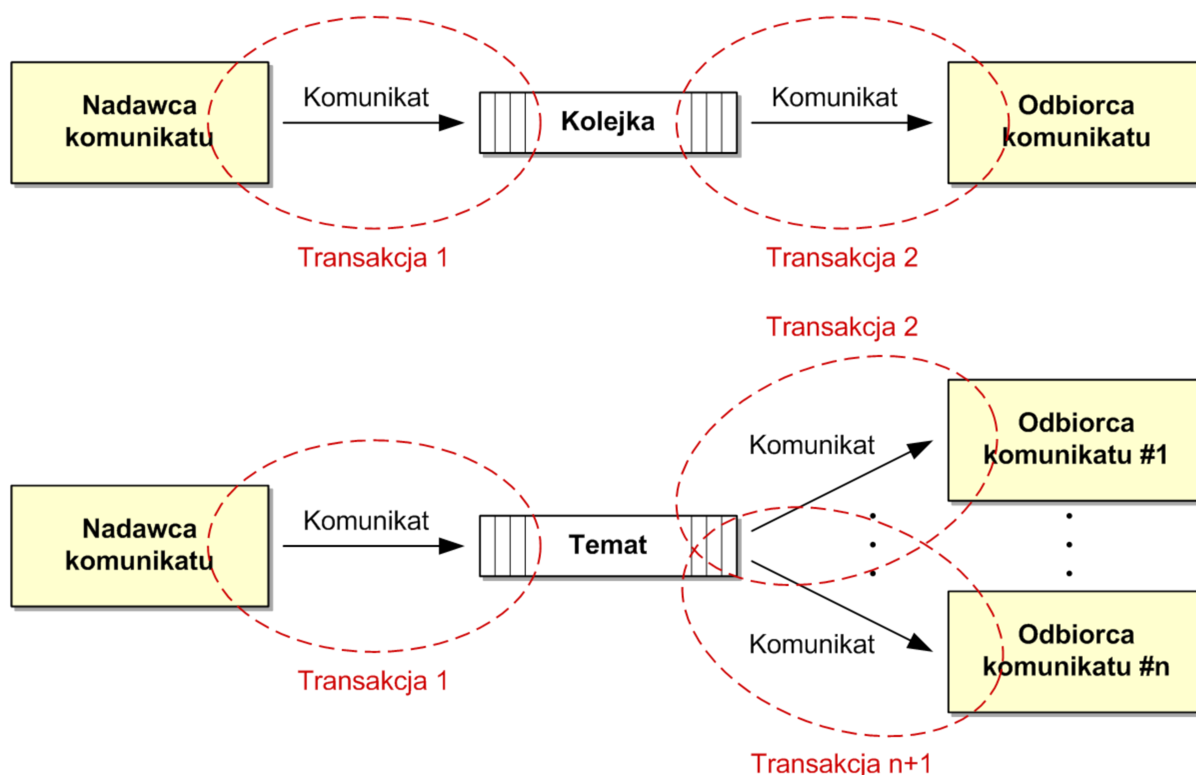
Specyfikacja JMS definiuje dwa sposoby asynchronicznego komunikowania się aplikacji z wykorzystaniem systemów kolejkowania. Są to modele:

- `Point to point`, w którym mamy jednego nadawcę i jednego odbiorcę, a komunikaty są przesyłane za pomocą kolejki (`Queue`).
- `Publish subscribe`, w którym mamy jednego nadawcę i dowolną liczbę odbiorców, a komunikaty są przesyłane (publikowane) za pomocą tzw. tematu (`Topic`).

Oba modele przedstawia poniższy rysunek.



W przypadku korzystanie z systemów kolejkowania kluczowe jest zrozumienie, że wysłanie komunikatu a następnie jego odebranie odbywa się w całkowicie odrębnych i niezależnych od siebie transakcjach. W przypadku kolejki (`Queue`) mamy dwie transakcje – jedną obsługującą wysłanie komunikatu i drugą obsługującą odebranie komunikatu. W przypadku tematu (`Topic`) mamy jedną transakcję obsługującą wysłanie komunikatu i tyle transakcji obsługujących odebranie komunikatu ile jest zarejestrowanych jego odbiorców. Zagadnienie to ilustruje poniższy rysunek.



Nie możemy zatem oczekiwać, że uda nam się wysłać komunikat i odebrać odpowiedź w jednej transakcji – jest to częsty błąd programistów stawiających pierwsze kroki w systemach kolejkowania. Próba wysłania komunikatu a następnie odebrania komunikatu odpowiedzi w jednej transakcji spowoduje zawieszenie aplikacji. Jest to spowodowane tym, że komunikat nie został w rzeczywistości wysyłany, bowiem transakcja nie została jeszcze zatwierdzona. Nie możliwe jest więc odebranie odpowiedzi na coś, co jeszcze nie zostało wysłane.

Podobnie jak w przypadku baz danych serwer aplikacji tworzy odpowiednie środowisko dla aplikacji, które chcą korzystać z systemów kolejkowania. Na środowisko to składają się następujące elementy:

- Deskryptor aplikacji, w którym twórca aplikacji definiuje logiczne nazwy zasobów JMS (kolejki, tematy), z których będzie korzystał. Dla aplikacji internetowych zasoby te są definiowane za pomocą elementów `<resource-ref>` w pliku `web.xml`.
- Definicji fizycznych zasobów JMS, w szczególności:
  - Fabryk połączeń do kolejek i tematów (`ConnectionFactory`, `QueueConnectionFactory`, `TopicConnectionFactory`).
  - Kolejek (`Queue`).
  - Tematów (`Topic`).
- Odwzorowania zdefiniowanych w aplikacjach nazw zasobów logicznych na zasoby fizyczne zdefiniowane w konfiguracji serwera. Odwzorowanie to jest realizowane za pomocą deskryptorów specyficznych dla danego serwera aplikacji i jest ustanawiane w momencie instalacji aplikacji na serwerze.
- Dostawcy JMS (`JMS Provider`), który umożliwia współpracę serwera aplikacji z konkretnym systemem kolejkowania. Jest to odpowiednik sterownika JDBC.
- Menedżera transakcji, który w imieniu aplikacji potrafi zarządzać transakcjami rozproszonymi, w których mogą może uczestniczyć operacje na systemie kolejkowania, o ile dostawca JMS wspiera transakcje rozproszone.

W tym artykule koncentruję się wyłącznie na aspektach transakcyjnego korzystania z systemów kolejkowania. Całkowicie pomijam dość złożoną kwestię konfigurowania dostępu do tych systemów z poziomu serwera aplikacji. Na potrzeby przykładów zakładam, że odpowiednie zasoby systemów kolejkowania są już poprawnie skonfigurowane i dostępne pod właściwymi kluczami w drzewie JNDI.

Podobnie jak w przypadku korzystania z baz danych w systemach kolejkowania możemy mieć do czynienia z dwoma modelami transakcji – lokalnym i globalnym (rozproszonym). Przyjrzyjmy się dokładniej obu modelom.

Transakcje lokalne

W przypadku transakcji lokalnych możemy operować wyłącznie na zasobach systemu kolejkowania (kolejki i tematy). Sterowanie transakcją odbywa się wtedy za pomocą mechanizmów, które są dostępne w JMS API. W przypadku transakcji lokalnych serwer aplikacji jest odpowiedzialny jedynie za udostępnienie odpowiednich zasobów (`ConnectionFactory`, `Queue`, `Topic`).

Zarządzanie transakcjami jak i wszystkie operacje wysyłania i odbierania komuników za pomocą JMS API odbywają się w ramach sesji (interfejs `javax.jms.Session`). Sesja JMS definiuje następujące metody pozwalające na zarządzanie lokalną transakcją:

- `void commit()` - zatwierdza transakcję, co powoduje usunięcie z kolejek pobranych komunikatów oraz dodanie do kolejek wysłanych komunikatów.
- `void rollback()` - wycofuje transakcję, co powoduje, że pobrane komunikaty nie są usuwane z kolejek, komunikaty wysłane nie są przekazywane do kolejek.
- `boolean getTransacted()` - zwraca informację, czy sesja pracuje w trybie transakcyjnym.

Jak można zauważyć, nie mamy tutaj metody, która sygnalizowałaby rozpoczęcie transakcji. O tym czy dana sesja jest transakcyjna decyduje sposób jej utworzenie w ramach połączenia z systemem kolejkowania (interfejs `javax.jms.Connection`). Interfejs ten udostępnia metodę `createSession(boolean transacted, int acknowledgeMode)` pozwalającą w zależności od parametru `transacted` stworzyć sesję transakcyjną bądź nietransakcyjną.

W związku z tym, że w JMS API występuje bardziej rozbudowana hierarchia sesji i połączeń, metod tworzenia sesji jest więcej. Pełny obraz sytuacji przedstawia poniższa tabela (wszystkie interfejsy są zdefiniowane w pakiecie `javax.jms`):

Fabryka połączeń	Połączenie	Metoda tworzenia sesji	Sesja
`ConnectionFactory`	`Connection`	`createSession`	`Session`
`QueueConnectionFactory`	`QueueConnection`	`createQueueSession`	`QueueSession`
`TopicConnectionFactory`	`TopicConnection`	`createTopicSession`	`TopicSession`

Hierarchia ta ma również znaczenie w kontekście zarządzania transakcjami. Jeżeli używamy sesji `QueueSession` to nie możemy w jej ramach w sposób transakcyjny operować na tematach (`Topic`). Analogicznie, jeśli używamy `TopicSession` to nie możemy w jej ramach w sposób transakcyjny operować na kolejkach (`Queue`). Aby móc w sposób transakcyjny operować zarówno na kolejkach jak i tematach musimy korzystać z bardziej ogólnego zestawu (`ConnectionFactory`, `Connection`, `Session`). Możliwość tą wprowadzono w specyfikacji JMS 1.1.

Warto pamiętać, że drugi parametr metod pozwalających na tworzenie sesji, który determinuje sposób potwierdzania odbioru wiadomości, w przypadku tworzenia sesji transakcyjnych nie ma znaczenia. Zwyczajowo ustawia się go na zero, ale ustawienie go na dowolną wartość i tak nie zmienia zachowania systemu. Jeśli sprawdzimy wartość tego parametru za pomocą metody `Session#getAcknowledgeMode()` to w przypadku sesji transakcyjnej zawsze otrzymamy wartość `Session#SESSION_TRANSACTED`.

Wykonywanie transakcji lokalnych ma zatem następujący przebieg:

- Pobranie z JNDI odpowiedniej fabryki połączeń (`ConnectionFactory`).
- Pobranie z JNDI zasobów na których będziemy operować (`Queue, Topic`).
- Utworzenie połączenia (`Connection`).
- Utworzenie transakcyjnej sesji JMS (`Connection#createSession(true, 0)`).
- Wykonanie w ramach sesji transakcyjnej operacji na pobranych zasobach (odbieranie, wysyłanie komunikatów).
- Zatwierdzenie lub wycofanie transakcji (`Session#commit()`, `Session#rollback()`)
- Zamknięcie sesji i połączeń.

W ramach jednej sesji możemy przeprowadzić wiele transakcji lokalnych. Każde zatwierdzenie lub wycofanie transakcji powoduje, że niejawnie otwierana jest nowa transakcja.

Jeżeli sesja zostanie utworzona jako nietransakcyjna wszystkie komunikaty będą pobierane i wysyłane na bieżąco. Przypomina to pracę z bazą danych w trybie automatycznego zatwierdzania transakcji (`java.sql.Connection#setAutoCommit(true)`).

Zobaczmy na przykładach zastosowanie tych mechanizmów.

Wysyłanie wielu wiadomości w jednej transakcji lokalnej

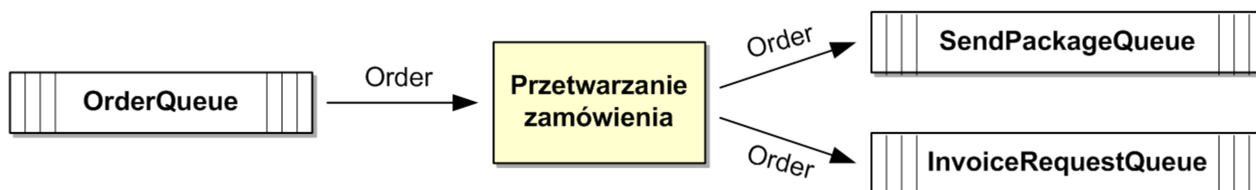
Na początek prosty przykład pokazujący jak można wysłać kilka komunikatów w jednej transakcji. Załóżmy, że dysponujemy prostą klasą reprezentującą zamówienie w sklepie internetowym (`Order`) i chcemy przekazać do kolejki dwa zamówienia w jednej transakcji. Załóżmy również, że serwer aplikacji jest tak skonfigurowany, że w drzewie JNDI pod kluczem `jms/QCF` znajduję się fabryka połączeń do kolejek, a pod kluczem `jms/OrderQueue` kolejka, do której można przekazywać zamówienia.

```
...
try {
    Context ctx = new InitialContext();

    QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup("jms/QCF");
    QueueConnection connection = qcf.createQueueConnection();
    Queue orderQueue = (Queue) ctx.lookup("jms/OrderQueue");
    QueueSession session = null;
    QueueSender orderQueueSender = null;

    connection.start();
    boolean transacted = true;
    session = connection.createQueueSession(transacted, 0);
    orderQueueSender = session.createSender(orderQueue);

    Order firstOrder = getFirstOrder();
    Order secondOrder = getSecondOrder();

    ObjectMessage firstOrderMessage = session.createObjectMessage(firstOrder);
    ObjectMessage secondOrderMessage = session.createObjectMessage(secondOrder);

    orderQueueSender.send(firstOrderMessage);
    orderQueueSender.send(secondOrderMessage);

    session.commit();

    orderQueueSender.close();
    session.close();
    connection.close();
} catch (Exception e) {
    throw new RuntimeException(e);
}
...
```

Jeśli zamiast do kolejki chcielibyśmy w jednej transakcji wysłać wiele komunikatów do tematu (`Topic`) to przykład byłby analogiczny. Należałoby jedynie zastąpić klasy `QueueConnectionFactory`, `QueueConnection`, `QueueSender` ich odpowiednikami związanymi z obsługą tematów, tj. `TopicConnectionFactory`, `TopicConnection`, `TopicPublisher`. Oczywiście w drzewie JNDI powinny znajdować się obiekty reprezentujące temat a nie kolejkę.

Odbieranie i wysłanie wiadomości w jednej transakcji lokalnej

Rozważmy bardziej złożony przypadek transakcji lokalnej obejmującej zarówno odbieranie jak i wysyłanie komunikatów. Załóżmy, że chcemy napisać kawałek kodu przetwarzającego zamówienia. Zamówienia spływają do kolejki `OrderQueue`, następnie są przetwarzane przez nasz program i trafiają do dwóch kolejek – `SendPackageQueue` (skąd trafią do systemu zajmującego się wysyłką paczki z magazynu) oraz `InvoiceRequestQueue` (skąd trafią do systemu finansowego, który wystawi fakturę za zamówienie). Sytuację tę ilustruje poniższy rysunek.

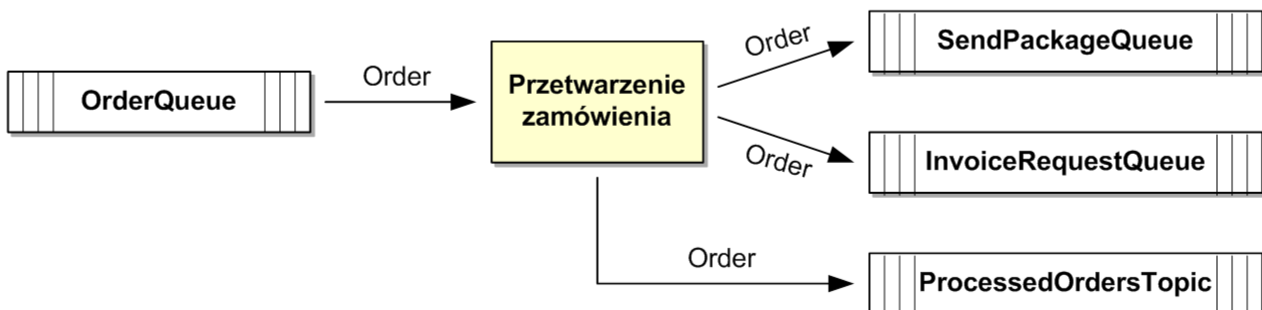


Kod który realizuje funkcjonalność opisanego przypadku może wyglądać następująco:

```
...
try {
    Context ctx = new InitialContext();

    QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup("jms/QCF");
    QueueConnection connection = qcf.createQueueConnection();
    Queue orderQueue = (Queue) ctx.lookup("jms/OrderQueue");
    Queue sendPackageQueue = (Queue) ctx.lookup("jms/SendPackageQueue");
    Queue invoiceRequestQueue = (Queue) ctx.lookup("jms/InvoiceRequestQueue");
    QueueSession session = null;
    QueueReceiver orderQueueReceiver = null;
    QueueSender sendPackageQueueSender = null;
    QueueSender invoiceRequestQueueSender = null;

    boolean transacted = true;
    session = connection.createQueueSession(transacted, 0);
    orderQueueReceiver = session.createReceiver(orderQueue);
    sendPackageQueueSender = session.createSender(sendPackageQueue);
    invoiceRequestQueueSender = session.createSender(invoiceRequestQueue);

    connection.start();

    // Odebranie zamówienia z koleki
    ObjectMessage orderMessage = (ObjectMessage)orderQueueReceiver.receive();
    Order order = (Order)orderMessage.getObject();

    // Przetworzenie zamówienia...

    // Wysłanie nowych komunikatów
    ObjectMessage packageMessage = session.createObjectMessage(order);
    ObjectMessage invoiceMessage = session.createObjectMessage(order);

    sendPackageQueueSender.send(packageMessage);
    invoiceRequestQueueSender.send(invoiceMessage);

    session.commit();

    orderQueueReceiver.close();
    sendPackageQueueSender.close();
    invoiceRequestQueueSender.close();
    session.close();
    connection.close();
} catch (Exception e) {
    throw new RuntimeException(e);
}
...
```

W przykładzie tym zastosowano blokującą operację odbierania komunikatu. Oznacza to, że program będzie czekał na wywołaniu metody `orderQueueReceiver.receive()`, aż w kolejce pojawi się jakieś zamówienie. Zgodnie ze specyfikacją J2EE można to odbieranie zamienić na nieblokujące w przypadku, gdy pracujemy z poziomu kontenera klienta (Application Client Container). Zastosowanie odbierania blokującego jest jednak bardziej ogólne.

Spróbujemy jeszcze nieco skomplikować powyższy przykład dokładając do kodu obsługi zamówienia wysłanie komunikatu do tematu `ProcessedOrdersTopic`. Ilustruje to poniższy rysunek.



Niestety okazuje się, że proste zmodyfikowanie poprzedniego przykładu nie jest możliwe. Wynika to z tego, że jest

zastosowany interfejs `QueueSession`, w ramach którego nie możemy tworzyć obiektów odbierających czy wysyłających komunikatów związanych z tematem (`Topic`). Należy zastosować nowy mechanizm wysyłania i odbierania komunikatów wprowadzony w specyfikacji JMS 1.1. Zatem kod, który w pełni realizuje nowe wymaganie może wyglądać następująco:

```
...
try {
    Context ctx = new InitialContext();

    ConnectionFactory cf = (ConnectionFactory) ctx.lookup("jms/QF");;
    Connection connection = cf.createConnection();;
    Queue orderQueue = (Queue) ctx.lookup("jms/OrderQueue");
    Queue sendPackageQueue = (Queue) ctx.lookup("jms/SendPackageQueue");
    Queue invoiceRequestQueue = (Queue) ctx.lookup("jms/InvoiceRequestQueue");
    Topic processedOrdersTopic = (Topic)ctx.lookup("jms/ProcessedOrdersTopic");
    Session session = null;
    MessageConsumer orderMessageConsumer = null;
    MessageProducer sendPackageMessageProducer = null;
    MessageProducer invoiceRequestMessageProducer = null;
    MessageProducer processedOrdersMessageProducer = null;

    boolean transacted = true;
    session = connection.createSession(transacted, 0);
    orderMessageConsumer = session.createConsumer(orderQueue);
    sendPackageMessageProducer = session.createProducer(sendPackageQueue);
    invoiceRequestMessageProducer = session.createProducer(invoiceRequestQueue);
    processedOrdersMessageProducer = session.createProducer(processedOrdersTopic);

    connection.start();

    // Odebranie zamówienia z koleki
    ObjectMessage orderMessage = (ObjectMessage)orderMessageConsumer.receive();
    Order order = (Order)orderMessage.getObject();

    // Przetworzenie zamówienia...

    // Wysłanie nowych komunikatów
    ObjectMessage packageMessage = session.createObjectMessage(order);
    ObjectMessage invoiceMessage = session.createObjectMessage(order);
    ObjectMessage processedOrderMessage = session.createObjectMessage(order);

    sendPackageMessageProducer.send(packageMessage);
    invoiceRequestMessageProducer.send(invoiceMessage);
    processedOrdersMessageProducer.send(processedOrderMessage);

    session.commit();

    orderMessageConsumer.close();
    sendPackageMessageProducer.close();
    invoiceRequestMessageProducer.close();
    processedOrdersMessageProducer.close();
    session.close();
    connection.close();
} catch (Exception e) {
    throw new RuntimeException(e);
}
...
```

Jak można zauważyć sposób odbierania i wysyłania komunikatów różni się znacząco od zastosowanego w poprzednim przykładzie. Ten dualizm sposobów odbierania i wysyłania komunikatów ma dość przykre konsekwencje praktyczne. Zazwyczaj rozbudowa istniejącego kodu wymaga dość znacznych przeróbek polegających na wymianie kodu specyficznego dla kolejek czy tematów na kod bardziej ogólny potrafiący obsłużyć jednocześnie kolejki i tematy.

Ograniczenia lokalnych transakcji JMS

Podstawowym i jednocześnie najważniejszym ograniczeniem lokalnych transakcji JMS jest brak możliwości uczestniczenia w takiej transakcji innych zasobów, np. baz danych. Jest to o tyle dotkliwe, że w praktyce większość aplikacji wykorzystuje bazy danych a systemy kolejkowania stanowią jedynie uzupełnienie związane zazwyczaj z

integracją systemów. Teoretycznie możliwe jest wykonanie aplikacji bazującej wyłącznie na przetwarzaniu i przesyłaniu komunikatów. Jednakże w przypadku współczesnych aplikacji biznesowych byłoby to zadanie czysto akademickie, mające niewiele wspólnego w praktyką.

Kolejnym ograniczeniem transakcji lokalnych jest problem związany z odbieraniem komunikatów. W architekturze Java Enterprise w sensowny sposób można odbieranie komunikatów zrealizować jedynie w kontenerze klienta (Application Client Container), który w praktyce rzadko jest wykorzystywany. Oczywiście standard Java Enterprise przewidział na potrzeby odbierania komunikatów specjalny typ komponentów EJB – Message Driven Bean, ale jest to już związane z zastosowaniem transakcji globalnych (rozproszonych).

Zatem, o ile omawiane w poprzedniej części artykułu, lokalne transakcje JDBC mają duże znaczenie praktyczne, to lokalne transakcje JMS w zasadzie są ograniczone do wąskiej grupy programów standalone, których zadaniem jest przetwarzanie lub rutowanie komunikatów.

Transakcje globalne (rozproszone)

W przypadku transakcji globalnych operacje na zasobach systemu kolejkowania (`Queue`, `Topic`) mogą być częścią większej transakcji obejmującej również operacje na innych zasobach (np. bazach danych). W tym przypadku zarządzanie transakcjami odbywa się za pomocą mechanizmów dostarczanych przez serwer aplikacji, a dokładniej przez menedżera transakcji rozproszonych, dostępnego dla programisty aplikacji przez obiekt `UserTransaction`. Odbywa się to dokładnie w taki sam sposób jak przedstawiłem w poprzednich częściach artykułu.

Aby systemy kolejkowania mogły uczestniczyć w transakcji rozproszonej ich dostawca musi zapewnić pracę w ramach dwufazowego protokołu zatwierdzania transakcji. Dostawca takiego systemu musi również przygotować bibliotekę (`JMS Provider`), która zapewni poprawną współpracę z serwerem aplikacji w ramach interfejsów XA (`XAConnectionFactory`, `XAConnection`, `XASession` itp.). Mechanizm współpracy z serwerem jest analogiczny do tego jaki omawiałem przy okazji transakcji rozproszonych w bazach danych.

Aby pokazać zagadnienie transakcji globalnych w JMS rozbudujmy nasz przykład z systemem przetwarzania zamówień. Składał się on będzie teraz z następujących elementów:

- Aplikacji internetowej zbierającej zamówienia od klientów i posiadającej własną bazę danych zamówień.
- Elementu przetwarzającego zamówienia, który będziemy realizować jako komponent Message Driven Bean (MDB).
- Systemu magazynowego, z którego są wysyłane paczki z zamówieniami.
- Systemu finansowego, w którym są wystawiane faktury.
- Systemu aktualizującego statystyki zamówień.
- Systemu wysyłającego potwierdzenia obsłużenia złożonego zamówienia.
- Kolejki przechowującej zamówienia – `OrderQueue`.
- Kolejki przechowującej żądania wysłania paczki z magazynu – `SendPackageQueue`.
- Kolejki przechowującej żądania wystawiania faktury za zamówienie – `InvoiceRequestQueue`.
- Tematu przechowującego informacje o przetworzonych zamówieniach – `ProcessedOrdersTopic`.

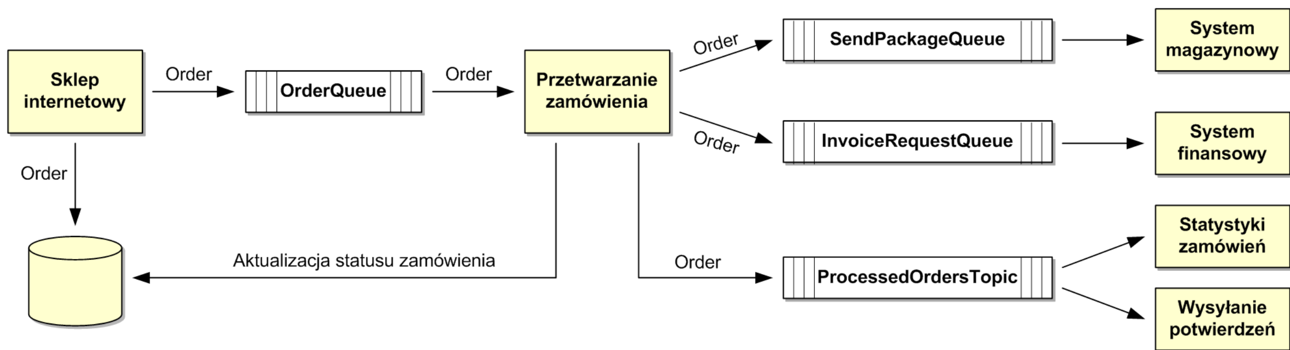
Zadaniem aplikacji internetowej, realizowanym w jednej transakcji globalnej, będzie:

- Zachowanie zamówienia w bazie danych.
- Przekazanie zamówienia do kolejki `OrderQueue`.

Zadaniem elementu przetwarzającego zamówienie, również w jednej transakcji globalnej, będzie:

- Odebranie zamówienia z kolejki `OrderQueue`.
- Zlecenie wysłania paczki do systemu magazynowego przez wysłanie komunikatu do kolejki `SendPackageQueue`.
- Zlecenie wystawienia faktury do systemu finansowego przez wysłanie komunikatu do kolejki `InvoiceRequestQueue`.
- Aktualizacja statusu zamówienia w bazie danych aplikacji internetowej.
- Powiadomienie innych systemów o przetworzeniu zamówienia przez wysłanie komunikatu do tematu `ProcessedOrdersTopic`.

Całość zagadnienia ilustruje poniższy rysunek.



Operacje na bazie danych i wysyłanie komunikatów do kolejki w jednej transakcji globalnej

Rozpocznijmy od kodu realizującego interesujący nas fragment funkcjonalności aplikacji internetowej. Korzystamy w nim z omawianego w poprzednich częściach artykułu interfejsu `UserTransaction`, żeby zarządzać zakresem naszej transakcji (`ut.begin()`, `ut.commit()`). Wewnątrz transakcji wykonujemy kolejno operacje na bazie danych i kolejce zamówień. Kod realizujący omawianą funkcjonalność może wyglądać następująco:

```
...
try {
    Context ctx = new InitialContext();
    UserTransaction ut = (UserTransaction)ctx.lookup("java:comp/UserTransaction");;

    ut.begin();

    Order order = getOrder();

    // wstaw zamówienie do bazy danych
    DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS");
    Connection conn = ds.getConnection();
    insertOrder(order, conn);
    conn.close();

    // wstaw zamówienie do kolejki
    QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup("jms/QCF");;
    QueueConnection connection = qcf.createQueueConnection();;
    Queue orderQueue = (Queue) ctx.lookup("jms/OrderQueue");
    QueueSession session = null;
    QueueSender queueSender = null;

    connection.start();
    session = connection.createQueueSession(false, 0);
    queueSender = session.createSender(orderQueue);

    ObjectMessage orderMessage = session.createObjectMessage(order);

    queueSender.send(orderMessage);

    queueSender.close();
    session.close();
    connection.close();

    ut.commit();
} catch (Exception e) {
    throw new RuntimeException(e);
}
...
```

Zwróćmy uwagę, że nie zakładamy transakcyjnej sesji JMS oraz nie wykonujemy na niej operacji `commit()`. Wynika to z tego, że w trakcie trwania transakcji globalnej nie jest dozwolone wywoływanie metod `commit()/rollback()` na sesji JMS. Jest to sytuacja analogiczna do tej, z którą mieliśmy do czynienia przy transakcjach globalnych w bazie danych. Tam również na obiekcie połączenia do bazy danych nie mogliśmy wykonywać operacji `commit()/rollback()`.

Odbieranie i wysyłanie komunikatów oraz operacje na bazie danych w jednej transakcji globalnej

Przyjrzyjmy się teraz implementacji kluczowego elementu naszego systemu, czyli komponentu którego zadaniem jest przetworzenie zamówienia według opisanych wcześniej założeń. Specyfikacja J2EE przewidziała do tego celu specjalny typ komponentu EJB – Message Driven Bean (MDB). Jest to bardzo prosty komponent, posiadający jedną istotną metodę `onMessage()`, która jest wywoływana w momencie, gdy w kolejce lub temacie znajdzie się komunikat gotowy do pobrania. Wymaga to oczywiście spięcia komponentów MDB odpowiedzialnych za przetwarzanie właściwych typów komunikatów z odpowiednimi zasobami na serwerze – odpowiednio kolejkami lub tematami. Odbywa się podczas deploymentu aplikacji na serwerze i jest realizowane za pomocą deskryptorów aplikacji specyficznych dla danego serwera. Więcej informacji na temat komponentów MDB można znaleźć w specyfikacji EJB.

Zobaczmy zatem jak może wyglądać implementacja naszego elementu przetwarzającego zamówienia w postaci komponentu MDB:

```
public class ProcessOrderMDB implements MessageDrivenBean, MessageListener {
    MessageDrivenContext mdbctx = null;

    public ProcessOrderMDB() {
    }

    public void setMessageDrivenContext(MessageDrivenContext mdbctx) throws EJBException{
        this.mdbctx  = mdbctx;
    }

    public void ejbRemove() throws EJBException {
    }

    public void ejbCreate() throws EJBException {
    }

    public void onMessage(Message msg) {
        ObjectMessage objectMessage = (ObjectMessage) msg;
        try {
            Order order = (Order)objectMessage.getObject();

            // przetworzenie zamówienia...

            Context ctx = new InitialContext();

            // przekazanie zamówienia do kolejek
            QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup("jms/QCF");
            QueueConnection queueConnection = qcf.createQueueConnection();
            Queue sendPackageQueue = (Queue) ctx.lookup("jms/SendPackageQueue");
            Queue invoiceRequestQueue = (Queue) ctx.lookup("jms/InvoiceRequestQueue");
            QueueSession session = null;
            QueueSender sendPackageQueueSender = null;
            QueueSender invoiceRequestQueueSender = null;

            queueConnection.start();

            session = queueConnection.createQueueSession(false, 0);
            sendPackageQueueSender = session.createSender(sendPackageQueue);
            invoiceRequestQueueSender = session.createSender(invoiceRequestQueue);

            ObjectMessage packageMessage = session.createObjectMessage(order);
            ObjectMessage invoiceMessage = session.createObjectMessage(order);

            sendPackageQueueSender.send(packageMessage);
            invoiceRequestQueueSender.send(invoiceMessage);

            sendPackageQueueSender.close();
            invoiceRequestQueueSender.close();
            session.close();
            queueConnection.close();

            // aktualizacja statusu zamówienia
            DataSource ds=(DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS");
            Connection conn = ds.getConnection();
```

```
            updateOrderStatus(order, conn);
            conn.close();

            // przekazanie zamówienia do tematu
            TopicConnectionFactory tcf = (TopicConnectionFactory) ctx.lookup("jms/TCF");
            TopicConnection topicConnection = tcf.createTopicConnection();
            Topic processedOrdersTopic = (Topic)ctx.lookup("jms/processedOrdersTopic");
            TopicPublisher processedOrdersTopicPublisher = null;

            topicConnection.start();
            TopicSession topicSession = topicConnection.createTopicSession(false, 0);
            processedOrdersTopicPublisher =
                                   topicSession.createPublisher(processedOrdersTopic);
            ObjectMessage processedOrderMessage =
                                   topicSession.createObjectMessage(order);
            processedOrdersTopicPublisher.publish(processedOrderMessage);

            topicSession.close();
            topicConnection.close();
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage(), e);
        }
    }
}
```

Do właściwego działania komponent MDB wymaga jeszcze odpowiedniego opisu, który instruuje serwer jak komponent powinien się zachowywać oraz jakich elementów wymaga do swojego działania. We wcześniejszych wersjach specyfikacji J2EE opis taki był dostarczany w postaci pliku `ejb-jar.xml`, w nowszych zamiast niego można stosować adnotacje – szczegóły w specyfikacji EJB. Poniżej przykład deskryptora do pokazanego wyżej komponentu:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar id="ejb-jar_1">

    <description><![CDATA[JMS Transcation Demo]]></description>
    <display-name>jms-transaction-demo</display-name>

    <enterprise-beans>
        <message-driven id="MessageDriven_ProcessOrder">
            <description><![CDATA[Process order bean]]></description>

            <ejb-name>ProcessOrderMDB</ejb-name>
            <ejb-class>pl.epoint.transactiondemo.ejb.ProcessOrderMDB</ejb-class>

            <transaction-type>Container</transaction-type>
            <acknowledge-mode>Auto-acknowledge</acknowledge-mode>

             <message-driven-destination>
                <destination-type>javax.jms.Queue</destination-type>
            </message-driven-destination>

             <resource-ref id="ResRef_DS_1">
                  <description>Transaction Demo Datasource</description>
                  <res-ref-name>jdbc/TransactionDemoDS</res-ref-name>
                  <res-type>javax.sql.DataSource</res-type>
                  <res-auth>Container</res-auth>
                  <res-sharing-scope>Shareable</res-sharing-scope>
             </resource-ref>

            <resource-env-ref>
                <resource-env-ref-name>jms/InvoiceRequestQueue</resource-env-ref-name>
                <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
            </resource-env-ref>

            <resource-env-ref>
                <resource-env-ref-name>jms/SendPackageQueue</resource-env-ref-name>
```

```
                <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
            </resource-env-ref>

            <resource-env-ref>
                <resource-env-ref-name>jms/ProcessedOrdersTopic</resource-env-ref-name>
                <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
            </resource-env-ref>
        </message-driven>
    </enterprise-beans>

   <assembly-descriptor >
   </assembly-descriptor>
</ejb-jar>
```

Kilka słów wyjaśnienia wymaga zachowanie transakcyjne komponentu MDB. W komponentach MDB możemy wybrać dwa rodzaje zachowania transakcyjnego (znacznik `transaction-type` w deskryptorze komponentu):

- `Container` - transakcje zarządzane przez kontener EJB (Container Managed Transaction).
- `Bean` – transakcje zarządzane samodzielnie przez programistę (Bean Managed Transaction) w ramach kodu komponentu MDB.

W przypadku obsługi transakcji przez kontener najpierw jest otwierana transakcja przez kontener, następnie w ramach tej transakcji pobierany jest komunikat z kolejki i dopiero wtedy wywoływana jest metoda `onMessage(...)` na komponencie MDB. Jeśli podczas wywołania metody `onMessage(...)` zostanie rzucony wyjątek kontener ma możliwość zwrócenia komunikatu do kolejki.

W przypadku obsługi transakcji przez programistę występuje całkowicie odmiennie zachowanie. Najpierw komunikat jest pobierany z kolejki, następnie wywoływana jest metoda `onMessage(...)` na komponencie MDB, w której programista sam otwiera transakcję. W tym przypadku operacja pobrania komunikatu z kolejki nie uczestniczy w transakcji, a więc kontener nie ma żadnej możliwości zwrócenie komunikatu do kolejki.

Zalecanym sposobem zarządzania transakcjami w przypadku komponentów MDB jest zarządzanie przez kontener, gdyż tylko wtedy błąd w trakcie przetwarzania powoduje, że komunikat wraca do kolejki i może być ponownie przetworzony.

Dla porządku poniżej przedstawiłem przykładową implementację metody `onMessage(...)` dla komponentu MDB, który samodzielnie zarządza transakcją:

```
public class ProcessOrderMDBBMT implements MessageDrivenBean, MessageListener {

    ...

    public void onMessage(Message msg) {
        ObjectMessage objectMessage = (ObjectMessage) msg;

        try {
            Order order = (Order)objectMessage.getObject();

            Context ctx = new InitialContext();
            UserTransaction ut = mdbctx.getUserTransaction();

            ut.begin();

            // przetworzenie zamówienia...

            // przekazanie zamówienia do kolejek
            QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup("jms/QCF");
            QueueConnection queueConnection = qcf.createQueueConnection();;
            Queue sendPackageQueue = (Queue) ctx.lookup("jms/SendPackageQueue");
            Queue invoiceRequestQueue = (Queue) ctx.lookup("jms/InvoiceRequestQueue");
            QueueSession session = null;
            QueueSender sendPackageQueueSender = null;
            QueueSender invoiceRequestQueueSender = null;

            queueConnection.start();

            session = queueConnection.createQueueSession(false, 0);
            sendPackageQueueSender = session.createSender(sendPackageQueue);
```

```
            invoiceRequestQueueSender = session.createSender(invoiceRequestQueue);

            ObjectMessage packageMessage = session.createObjectMessage(order);
            ObjectMessage invoiceMessage = session.createObjectMessage(order);

            sendPackageQueueSender.send(packageMessage);
            invoiceRequestQueueSender.send(invoiceMessage);

            sendPackageQueueSender.close();
            invoiceRequestQueueSender.close();
            session.close();
            queueConnection.close();

            // aktualizacja statusu zamówienia
            DataSource ds=(DataSource)ctx.lookup("java:comp/env/jdbc/TransactionDemoDS");
            Connection conn = ds.getConnection();
            updateOrderStatus(order, conn);
            conn.close();

            // przekazanie zamówienia do tematu
            TopicConnectionFactory tcf = (TopicConnectionFactory) ctx.lookup("jms/TCF");
            TopicConnection topicConnection = tcf.createTopicConnection();
            Topic processedOrdersTopic = (Topic)ctx.lookup("jms/processedOrdersTopic");
            TopicPublisher processedOrdersTopicPublisher = null;

            topicConnection.start();
            TopicSession topicSession = topicConnection.createTopicSession(false, 0);
            processedOrdersTopicPublisher =
                                   topicSession.createPublisher(processedOrdersTopic);
            ObjectMessage processedOrderMessage =
                                   topicSession.createObjectMessage(order);
            processedOrdersTopicPublisher.publish(processedOrderMessage);

            topicSession.close();
            topicConnection.close();

            ut.commit();
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage(), e);
        }
    }
}
```

W deskryptorze takiego komponentu element `transaction-type` musi przyjąć następującą postać:

```
<transaction-type>Bean</transaction-type>
```

W ramach wywołania metody `onMessage()` komponentu MDB możemy również wywoływać inne komponenty EJB, które w zależności od ich konfiguracji mogą się podłączać do trwającej transakcji. Szerzej będę o tym pisał w kolejnej części artykułu.

Podsumowanie

W przykładach dla zwiększenia ich czytelności pominąłem właściwą obsługę wyjątków. W rzeczywistym systemie należy ją oczywiście dodać. Obsługę wyjątków w lokalnych transakcjach JMS można wykonać wzorując się na przykładzie z poprzedniej części artykułu, gdzie pokazałem w jaki sposób zrealizować obsługę wyjątków podczas realizacji lokalnych transakcji JDBC. Z kolei obsługę wyjątków w globalnych (rozproszonych) transakcjach wykonujemy w sposób identyczny jak to pokazałem w pierwszej części artykułu. Szczególną uwagę należy zwrócić na obsługę wyjątków w komponentach MDB. Specyfikacja EJB definiuje jakie powinno być zachowanie komponentów i kontenera w przypadku rzucenia wyjątku w trakcie wykonania metody biznesowej. Temat ten będę poruszał bardziej szczegółowo podczas omawiania transakcji w komponentach EJB.

Jak można zauważyć, dużą część kodu przedstawionych przykładów, stanowią operacje pomocnicze takie jak pobranie obiektów z JNDI, utworzenie połączeń, utworzenie sesji, zamykanie sesji, zamykanie połączeń, właściwa obsługa wyjątków itp. Samego kodu realizującego logikę jest stosunkowo niewiele. W praktyce mając do obsłużenia kilkadziesiąt, czy kilkaset operacji na kolejkach ciężko byłoby je pisać w przedstawiony sposób. Na szczęście można działania pomocnicze stosunkowo łatwo opakować w klasy narzędziowe. W przykładach pozostawiłem je celowo, żeby pokazać pełną złożoność współpracy z serwerem aplikacji.

Transakcje z wykorzystaniem systemów kolejkowania wydają się być idealnym narzędziem do integracji systemów, które muszą pracować transakcyjnie, zachowując jednocześnie zalety komunikacji asynchronicznej. Jak to zwykle bywa diabeł tkwi w szczegółach. A w tym przypadku w szczegółach współpracy serwera aplikacji z systemami kolejkowania. Systemy kolejkowania dostarczane razem z serwerem aplikacji raczej nie sprawiają kłopotów. Natomiast współpraca z zewnętrznymi systemami kolejkowania napotyka na sporo trudności. Szczególnych problemów możemy się spodziewać w zakresie pracy systemów kolejkowania w ramach transakcji globalnych. Niestety specyfikacja JMS nie wymusza na dostawcy dostarczenia systemu wspierającego transakcje rozproszone. Określa ona jedynie, że jeśli dostawca zdecyduje się umożliwić pracę w transakcji rozproszonej, to musi być zrobiona według zasad określonych w specyfikacji JMS (głównie chodzi o implementację szeregu interfejsów XA). Można śmiało powiedzieć, że jest to temat zdecydowanie gorzej dopracowany niż współpraca serwera aplikacji z różnymi bazami danych.

W następnej części artykułu postaram się przedstawić zagadnienie związane z transakcjami w komponentach EJB oraz strategie, które możemy zastosować budując systemy transakcyjne w technologii Java Enterprise.

Literatura:

[1] Java Message Service 1.1
[2] Java 2 Enterprise Edition Specification 1.3 oraz 1.4
[3] EJB Specification 2.0, 2.1, 3.0
[4] Java Transaction API Specification
[5] Mark Little, Jon Maron, Greg Pavlik, Java Transaction Processing, Prentice Hall, 2004
[6] Dokumentacja do serwera aplikacji JBoss
[7] Dokumentacja do serwera aplikacji IBM Websphere