Java wybrane technologie

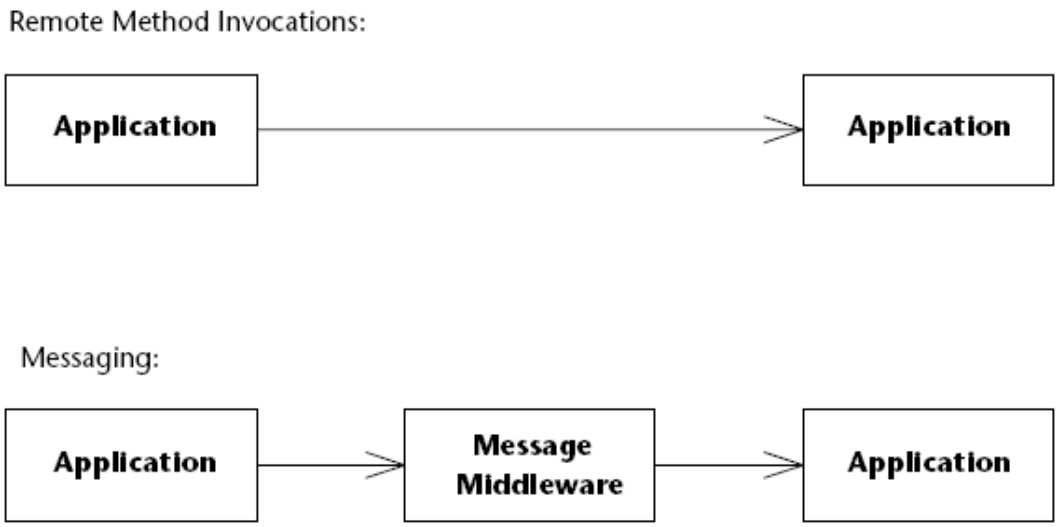
spotkanie nr 9

Java Message Service

i

Message-Driven Beans

Alternatywa dla RMI-IIOP



- asynchroniczność (*asynchrony*)
  - brak blokowania
  - daje się fire-and-forget
- rozprężenie (*decoupling*)
  - klienci nie muszą znać serwera
- niezawodność (*reliability*)
  - serwer nie musi cały czas działać
  - guaranteed message delivery
    - certified message delivery
    - store and forward
  - trwali/nietrwali konsumenci (*durable*)
- wielu odbiorców i nadawców

Message oriented middleware (MOM)

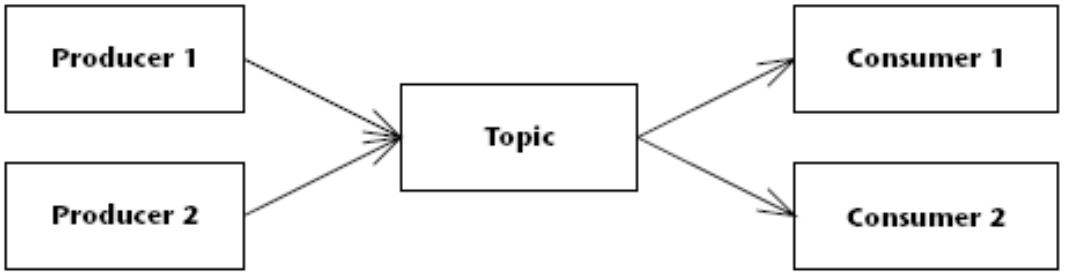
- Przykładowe MOM:
  - Tibco Rendezvous,
  - IBM Web-Sphere MQ,
  - BEA Tuxedo/Q,
  - Sun Java System Messaging Server,
  - Microsoft MSMQ,
  - Sonic Software SonicMQ i
  - FioranoMQ
- Niektóre funkcje
  - gwarantowanie dostarczenia wiadomości
  - odporność na błędy
  - równoważenie obciążenia
  - wykrywanie nieaktywnych odbiorców
  - SOAP po JMS

Java Message Service (JMS)

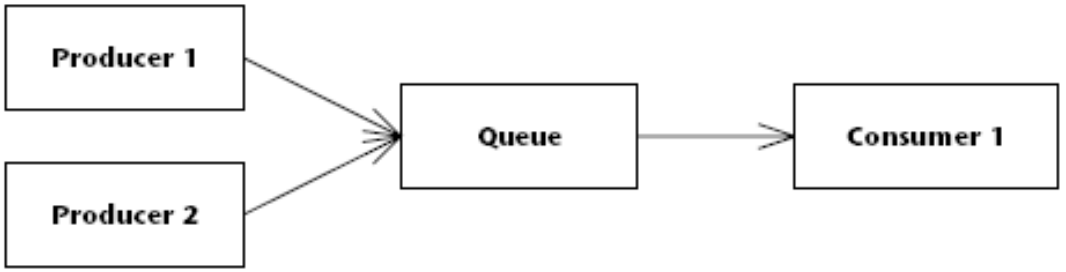
- podobny pomysł do JDBC i JNDI
- API do wysyłania i odbierania komunikatów
- Service Provider Interface (SPI)

Style komunikacji

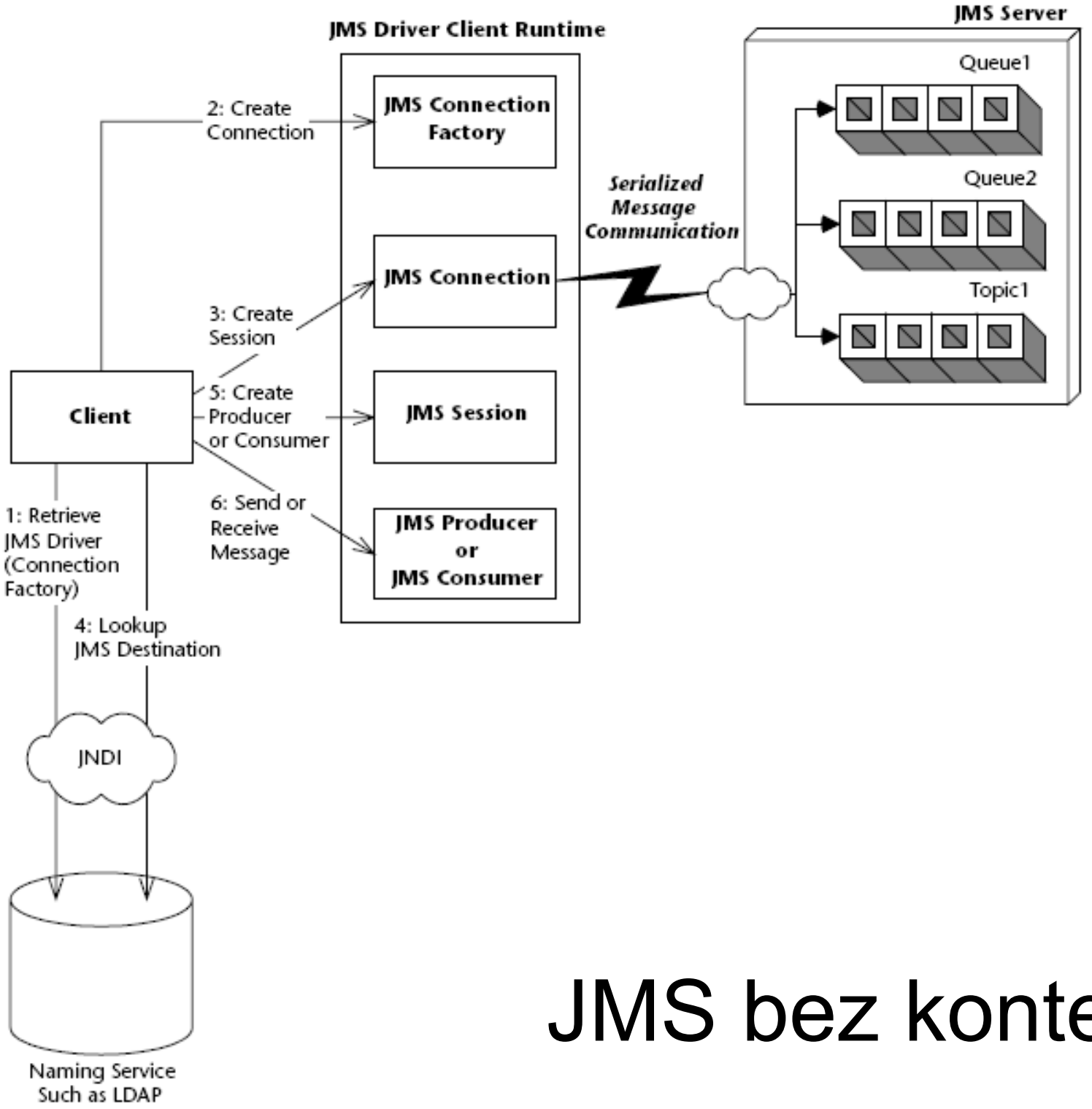
Publish/Subscribe:



Point-to-Point:



- Publish/Subscribe
  - jak radio/telewizja
- Point-to-Point
  - producenci/konsumenci
- Request-Reply
  - asynchroniczne wywoływanie procedur
  - rzadziej spotykane
  - implementowane przy pomocy poprzednich



JMS bez kontenera

Przykład

```
import javax.jms.*;
import javax.naming.InitialContext;
public class Producent {
  public static void main(String[] args) throws Exception {
    InitialContext ctx = new InitialContext(...);

    TopicConnectionFactory factory =
      (TopicConnectionFactory) ctx.lookup("jms/mojaFabrykaT");

    TopicConnection connection = factory.createTopicConnection();
    TopicSession session =
      connection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);

    Topic topic = (Topic)ctx.lookup("jms/mojT");

    TopicPublisher publisher = session.createPublisher(topic);

    TextMessage msg = session.createTextMessage();
    msg.setText("To jest komunikat.");
    publisher.send(msg);

    publisher.close();
  }
}
```

Rodzaje interfejsów

- ConnectionFactory
  - QueueConnectionFactory
  - TopicConnectionFactory
- Connection
  - QueueConnection
  - TopicConnection
- Destination
  - Queue
  - Topic
- Session
  - QueueSession
  - TopicSession
- MessageProducer
  - QueueSender
  - TopicPublisher
- MessageConsumer
  - QueueReceiver, QueueBrowser
  - TopicSubscriber

Rodzaje wiadomości

- `BytesMessage`
- `ObjectMessage`
- `TextMessage`
- `StreamMessage`
- `MapMessage`

Wiadomości mogą mieć nagłówki.

Message-Driven Beans

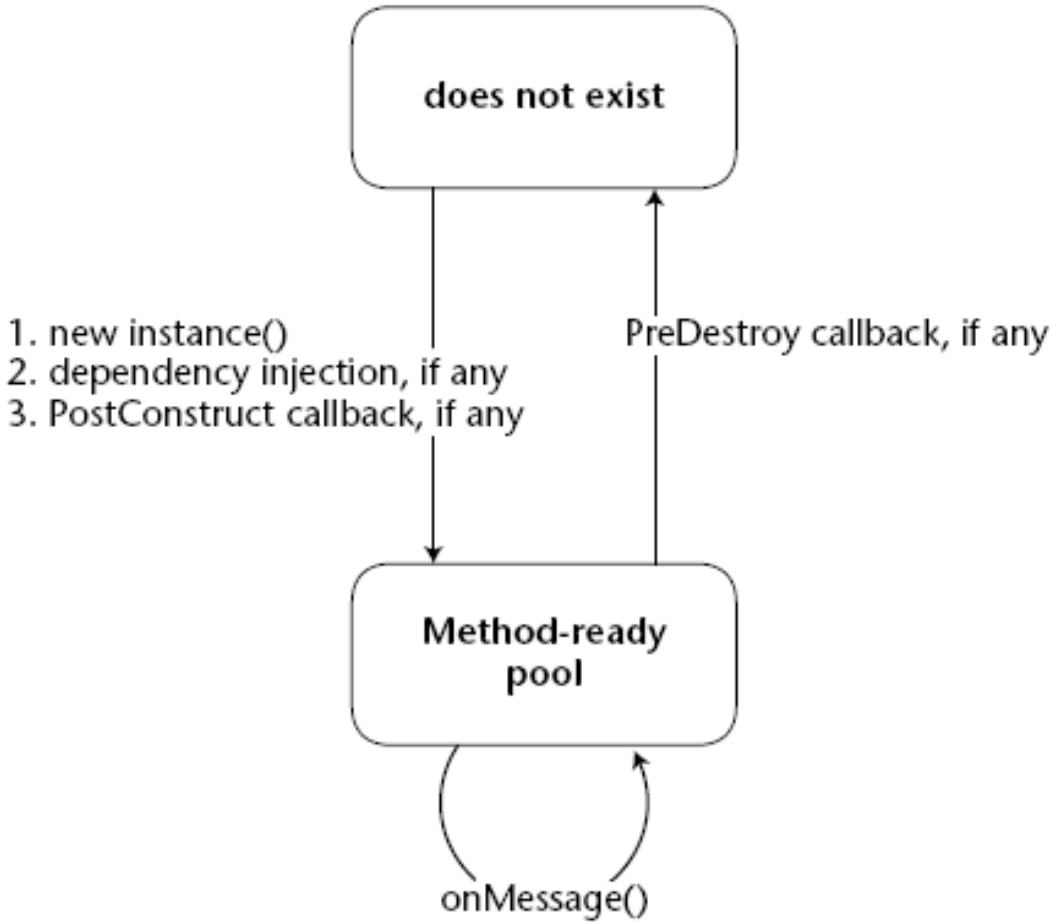
MDB

- Niedostępny dla klientów
- Zazwyczaj implementuje `javax.jms.MessageListener` i odbiera wiadomości JMS
  - dostaje wszystkie wiadomości (chyba, że używamy selectora)
  - trzeba się nagimnastykować żeby odpowiedzieć
  - wyjątki zgłaszane podczas przetwarzania wiadomości nie dotrą do klienta
  - nie ma stanu
  - kontener odpowiada za równoległe przetwarzanie (nie można nic zakładać o kolejności obsługi wiadomości)
- Od EJB 2.1 konektory Java EE Connector Architecture mogą stanowić źródło komunikatów

Alternatywy

- Własne obiekty odpalają Session Beany
  - Nie trzeba pisać kodu, który zarejestruje nasz obiekt jako konsumenta.
  - Nie trzeba pisać wielowątkowej aplikacji.
  - Nie trzeba się martwić startowaniem konsumentów.
  - Możemy korzystać z usług kontenera.
- Session Bean jako konsument
  - Bean jest jednowątkowy i jeżeli obsługuje właśnie żądanie, nie będzie mógł obsłużyć wiadomości.
  - Co jak nie ma beanów w chwili nadejścia wiadomości?

Wymagania



- Wymagania
  - bezargumentowy konstruktor
  - dla JMS: `javax.jms.MessageListener`
    - ma metodę `onMessage(Message m)`
- a

Przykład

```
@MessageDriven(activationConfig =
{ @ActivationConfigProperty
  (
    ropertyName = "destinationType",
    propertyValue = "javax.jms.Topic")
} )
public class MyFirstMDB implements MessageListener {
  public MyFirstMDB() {
    System.out.println("MyFirstMDB()");
  }

  public void onMessage(Message msg) {
    if (msg instanceof TextMessage) {
      TextMessage tm = (TextMessage) msg;
      try {
        String text = tm.getText();
        System.out.println("Odebrany komunikat : " + text);
      } catch (JMSException e) {
        e.printStackTrace();
      }
    }
  }

  @PreDestroy
  public void remove() {
    System.out.println("MyFirstMDB.remove()");
  }
}
```

Deskryptor

```
<?xml version="1.0" encoding="UTF-8" ?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="3.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
  <enterprise-beans>

    <message-driven>
      <ejb-name>LogBeanDD</ejb-name>
      <ejb-class>examples.messaging.dd.LogBean</ejb-class>
      <messaging-type>javax.jms.MessageListener</messaging-type>
      <transaction-type>Bean</transaction-type>
      <message-destination-type>javax.jms.Topic</message-destination-type>

      <activation-config>
        <activation-config-property>
          <activation-config-property-name>destinationType</...>
          <activation-config-property-value>javax.jms.Topic</...>
        </activation-config-property>
      </activation-config>
    </message-driven>

  </enterprise-beans>
</ejb-jar>
```

- Mapowanie na konkretny temat/kolejkę wskazujemy w deskryptorze kontenera

Opcjonalne podelementy @MessageDriven oraz <message-driven>

- ```
  @ActivationConfigProperty(
    propertyName = "destinationType",
    propertyValue = "javax.jms.Topic"
  )
  ```

- ```
  @ActivationConfigProperty(
    propertyName="messageSelector",
    propertyValue="JMSType = 'ala' AND
                   ma = 'kota'"
  )
  ```
  - `m.setStringProperty("ma","kota")`
  - składnia wzorowana na SQL

- ```
  @ActivationConfigProperty(
   propertyName="subscriptionDurability
  ",
   propertyValue="NonDurable"
  )
  ```
- o tranzakcjach jeszcze będzie

- ```
  <activation-config-property>
    <activation-config-property-name>
      destinationType
    </activation-config-property-name>
    <activation-config-property-value>
      javax.jms.Topic
    </activation-config-property-value>
  </activation-config-property>
  ```

- ```
  <activation-config-property>
    <activation-config-property-name>
      messageSelector
    </activation-config-property-name>
    <activation-config-property-value>
      JMSType='ala' AND ma='kota'
    </activation-config-property-value>
  </activation-config-property>
  ```

- ```
  <activation-config-property>
    <activation-config-property-name>
      subscriptionDurability
    </activation-config-property-name>
    <activation-config-property-value>
      NonDurable
    </activation-config-property-value>
  </activation-config-property>
  ```

Zagadnienia zaawansowane

- Transakcje
  - konsument i producent nie należą do tej samej transakcji (wiadomość pojawia się dopiero po zacommitowaniu)
- Bezpieczeństwo
  - nie ma standardowego sposobu przekazywania *security identity*
- Równoważenie obciążenia
  - pośrednik i model pull
  - to kontener jest odbiorcą wiadomości, a nie poszczególne egzemplarze
  - wszystkie kontenery w klastrze będą odbiorcami
- Metody `@PreDestroy` są wywoływana rzadko
  - sprzątamy zanim zgłosimy wyjątek

Odpowiadanie

- Kolejka do odpowiedzi
- Kolejka tymczasowa (związana z obiektem `Connection`)
  - `JMSReplyTo`
  - `JMSCorrelationID`
  - tymczasowa kolejka stworzona przez bean może przepaść!

Kiedy stosować/nie stosować komunikatów

Stosować

- kosztowne czynności, których efekty nie muszą być natychmiastowe
- nie trzeba się blokować jeżeli nie oczekujemy odpowiedzi (wartość zwrotna `void`)
- równoważenie obciążenia na zasadzie *pull*, a nie *push*
- łatwo wykryć kiedy jest za mało konsumentów
- łatwa priorytyzacja
- łatwa integracja z systemami zastanymi
- luźne sprzężenie
- większa niezawodność przy słabym łączu sieciowym
- komunikacja wiele do wiele

Nie stosować

- jak oczekujemy wyniku
- jeżeli nie ma pewności, że operacja się powiedzie
- kiedy operacja ma być częścią większej transakcji
- kiedy nie ufamy klientom (można samemu udoskonalić)
- w małych aplikacjach, gdzie pośrednik może być zbyt zasobożerny
- jak potrzeba obiektowości i silnych typów
- system ma być prosty i łatwy w testowaniu i debugowaniu