
Kapitola 1. Přednáška 10 - Systémy řízení zpráv, Java Messaging Service (JMS) API

Obsah

Systémy řízení zpráv, Java Messaging Service (JMS) API	1
Systémy řízení zpráv - motivace	1
Systémy řízení zpráv - výhody	2
Systémy řízení zpráv - principy	2
Systémy řízení zpráv - standardy	2
JMS - kdy použít?	2
JMS - co nabízí	3
JMS - architektura	3
JMS - domény	4
Point-to-point Messaging	4
Publish/Subscribe	5
Komponenty JMS API	5
Zpracování zprávy	6
Demo JMS s použitím Sun Java System Message Queue	6
Použití Sun Java System Message Queue	7
Co je třeba pro překlad klienta	7
Spuštění serveru (Message Brokeru)	7
Test běhu SJSMQ	7
Kostra jednoduché aplikace	8

Systémy řízení zpráv, Java Messaging Service (JMS) API

Systémy řízení zpráv - motivace

Svět podnikových (ale i jiných) IS často vyžaduje integrovat stávající systémy.

Možností, jak integrovat je více, liší se např. v míře/těsnosti vazby mezi jednotlivými systémy. Extrémy jsou:

- těsná vazba (tight coupling) - systémy "o sobě ví všechno" (API), jsou na sobě závislé, často se musí výrazně modifikovat, aby mohly spolupracovat

- volná vazba (loose coupling) - systémy "o sobě neví skoro nic", jsou nezávislé

Systémy řízení zpráv - výhody

Typickým příkladem a jedním směrem moderních integrujících technologií jsou *systémy řízení zpráv*.

Oproti jiným integračním technologiím mají tyto přednosti:

- nevyžadují příliš přebudovávat stávající aplikace proto, aby mohly spolupracovat;
- nezpůsobují úzkou vazbu integrovaných systémů (systémy jsou tzv. *loosely coupled*);
- jako odesílatelé zprávy nemusíme vůbec tušit, jak ji příjemce zpracuje, stačí dohodnout:
 - dohodnout formát
 - znát cílovou adresu zprávy

Systémy řízení zpráv - principy

Systémy řízení zpráv slouží ke správě výměny zpráv mezi softwarovými systémy nebo komponentami.

- komunikace je peer-to-peer, systém řízení zpráv zajišťuje infrastrukturu
- komponenty (systémy), které chtějí zasílat/přijímat zprávy, k tomu využívají služeb agentů (kteří jsou součástí API systémů řízení zpráv)

Systémy řízení zpráv - standardy

Javovou podobou rozhraní k systému řízení zpráv je *Java Messaging Service (JMS) API*, jehož historie sahá do roku 1998, současná verze je 1.1. (z r. 2002).

JMS je poměrně malé rozhraní, které nenutí programátora studovat příliš mnoho konceptů - je jednoduché.

JMS umožňuje komunikaci, která je:

- | | |
|-------------|---|
| asynchronní | zpráva je odeslána konzumentovi, který nemusí být stále "on-line"; vybere si zprávu, až se připojí |
| spolehlivá | systém zajistí perzistentní ukládání zpráv do doby, než jsou příjemcem přečteny; zajistí, že jsou přečteny právě jednou |

JMS - kdy použít?

Možné důvody, proč JMS:

- chceme propojovat komponenty bez znalostí jejich API
- chceme robustní systém: je možný nezávislý souběžný provoz komponent
- zasílání zpráv a čekání na odpověď je asynchronní - proces čekat nemusí (není to tedy request/response výměna)

JMS - co nabízí

Současná verze JMS nabízí:

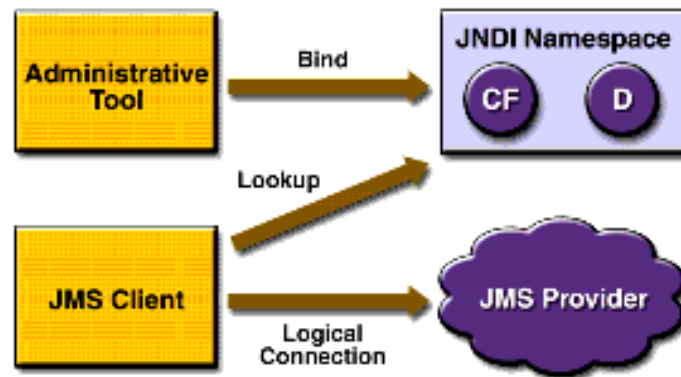
- rozhraní pro tvorbu klientů k (i jiným) systémům řízení zpráv
- rozhraní Message-driven Beans

JMS - architektura

Aplikace JMS sestává z těchto relevantních entit:

JMS Provider	poskytovatel infrastruktury zasílání zpráv vč. administrace
JMS Clients	klienti produkující nebo konzumující zprávy
Messages	objekty - zprávy
Administered objects	typickými reprezentanty jsou předkonfigurované tovární objekty (factories) na vytváření objektů <i>destinations</i> a <i>connections</i> . Vyhledání a zpřístupnění těchto servisních objektů je zajištěno přes JNDI.

Obrázek 1.1. Spolupráce JMS a JNDI (z tutoriálu J2EE Sun)



JMS - domény

Většina implementací JMS umožňuje:

point-to-point messaging

zasílání zpráv od jednoho odesílatele k jednomu příjemci

publish-subscribe

více odesílatelů může zprávu "vystavit" na místo, kam se odběratel ("předplatitel") může zapsat a zprávy odebírat

Point-to-point Messaging

Proces P-to-P zaslání zpráv zahrnuje tři základní participující objekty:

klient-odesílatel

vytvoří a pošle zprávu do *fronty* zpráv

fronta zpráv

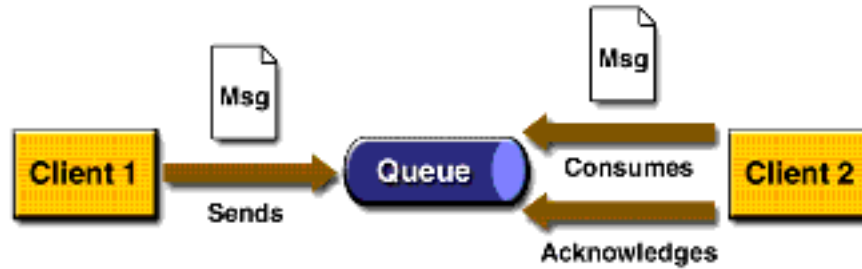
zprávu přijme a zajistí její "přežití" než je zkonsumována nebo zastará (time-out)

klient-příjemce

odebírání zpráv z fronty

Vztah producent-konzument je 1:1.

Obrázek 1.2. Mechanismus Point-to-Point (z tutoriálu J2EE Sun)



Publish/Subscribe

Producent adresuje zprávu tzv. tématu (topic).

Konzument se může přihlásit k odběru zpráv k tomuto tématu.

Vztah producentkonzument tak není obecně 1:1.

Systém zabezpečí uchování zprávy na tak dlouho, dokud ji všichni odběratelé nepřečtou.

Obrázek 1.3. Mechanismus Publish/Subscribe (z tutoriálu J2EE Sun)



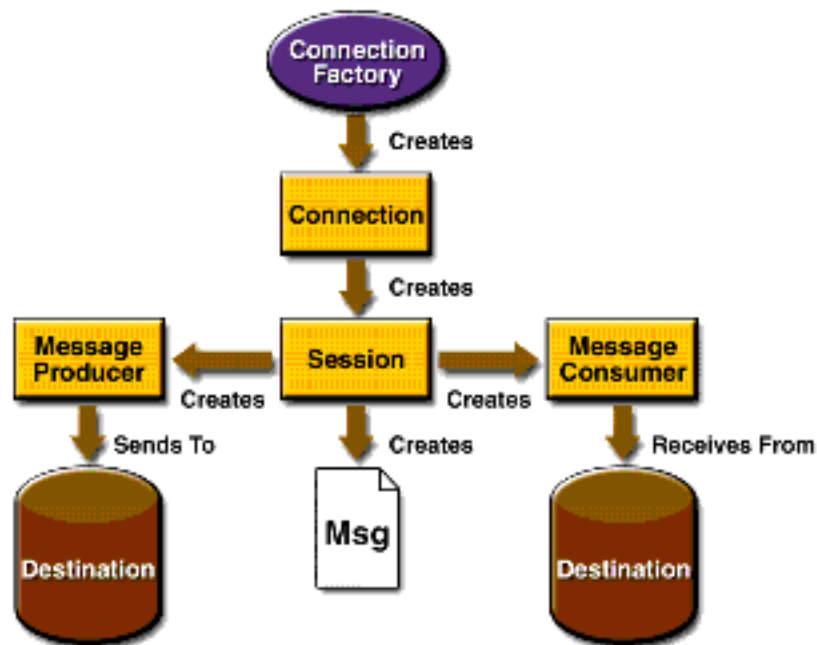
Komponenty JMS API

Komponenty (třídy, rozhraní):

- Administered objects
- Connections
- Sessions
- Message Producers

- Message Consumers
- Messages

Obrázek 1.4. Programovací model JMS API (z tutoriálu J2EE Sun)



Zpracování zprávy

Klient přijímající zprávu (konzument) ji může přečíst v režimu:

synchronním řekne si o ni metodou *receive*

asynchronním klient je posluchačem události "příjetí zprávy" (metoda *onMessage*); systém událost vyvolá

takovým typickým klientem je Message-driven Bean (EJB komponenta)

Demo JMS s použitím Sun Java System Message Queue

Použití Sun Java System Message Queue

Sun nabízí podrobný tutoriál k použití jeho implementace systému řízení zasílání zpráv: Sun Java System Message Queue [<http://docs.sun.com/app/docs/doc/817-6026>]

Součástí tutoriálu je Quick Start Tutorial [<http://docs.sun.com/source/817-6026/tutorial.html>] s praktickým návodem jak nainstalovat, přeložit a spustit server a jednoduchého klienta.

Co je třeba pro překlad klienta

Pro překlad klienta musíme mít v CLASSPATH tyto soubory:

jar soubory (z adresáře `IMQ_HOME\lib\`) `jms.jar imq.jar jndi.jar` (u Javy 1.4 je to automaticky, není třeba přidávat do CLASSPATH)

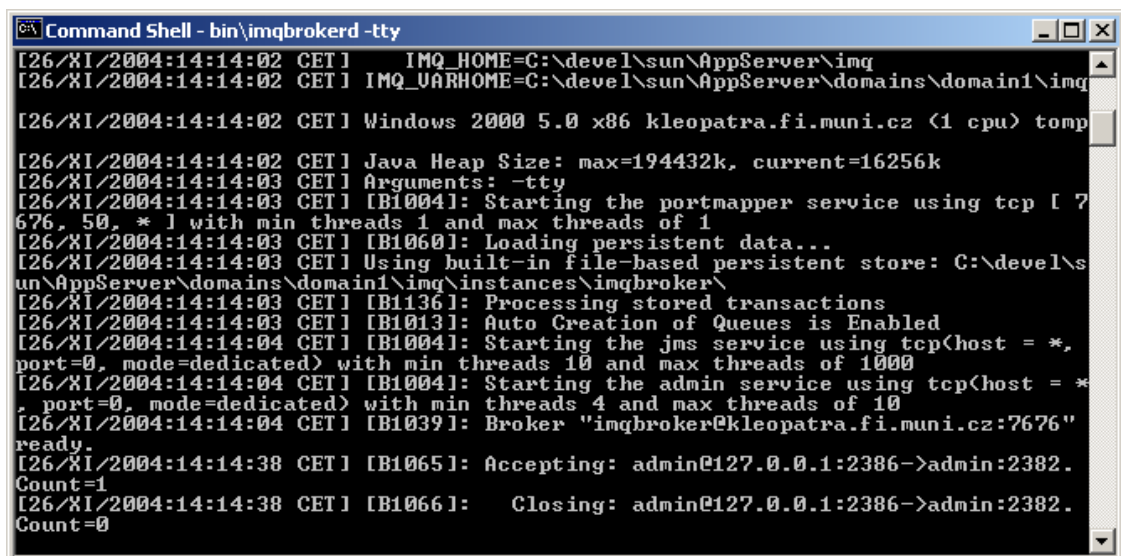
Spuštění serveru (Message Brokeru)

Provedeme spustitelným souborem z `IMQ_HOME/bin/`

- `imqbrokerd -tty`

Server SJS MQ se rozběhne s hláškami:

Obrázek 1.5. Spuštění SJS Message Queue



```
Command Shell - bin\imqbrokerd -tty
[26/XI/2004:14:14:02 CET] IMQ_HOME=C:\devel\sun\AppServer\imq
[26/XI/2004:14:14:02 CET] IMQ_UARHOME=C:\devel\sun\AppServer\domains\domain1\imq
[26/XI/2004:14:14:02 CET] Windows 2000 5.0 x86 kleopatra.fi.muni.cz <1 cpu> temp
[26/XI/2004:14:14:02 CET] Java Heap Size: max=194432k, current=16256k
[26/XI/2004:14:14:03 CET] Arguments: -tty
[26/XI/2004:14:14:03 CET] [B1004]: Starting the portmapper service using tcp [ 7
676, 50, * ] with min threads 1 and max threads of 1
[26/XI/2004:14:14:03 CET] [B1060]: Loading persistent data...
[26/XI/2004:14:14:03 CET] Using built-in file-based persistent store: C:\devel\s
un\AppServer\domains\domain1\imq\instances\imqbroker\
[26/XI/2004:14:14:03 CET] [B1136]: Processing stored transactions
[26/XI/2004:14:14:03 CET] [B1013]: Auto Creation of Queues is Enabled
[26/XI/2004:14:14:04 CET] [B1004]: Starting the jms service using tcp(host = *,
port=0, mode=dedicated) with min threads 10 and max threads of 1000
[26/XI/2004:14:14:04 CET] [B1004]: Starting the admin service using tcp(host = *
, port=0, mode=dedicated) with min threads 4 and max threads of 10
[26/XI/2004:14:14:04 CET] [B1039]: Broker "imqbroker@kleopatra.fi.muni.cz:7676"
ready.
[26/XI/2004:14:14:38 CET] [B1065]: Accepting: admin@127.0.0.1:2386->admin:2382.
Count=1
[26/XI/2004:14:14:38 CET] [B1066]: Closing: admin@127.0.0.1:2386->admin:2382.
Count=0
```

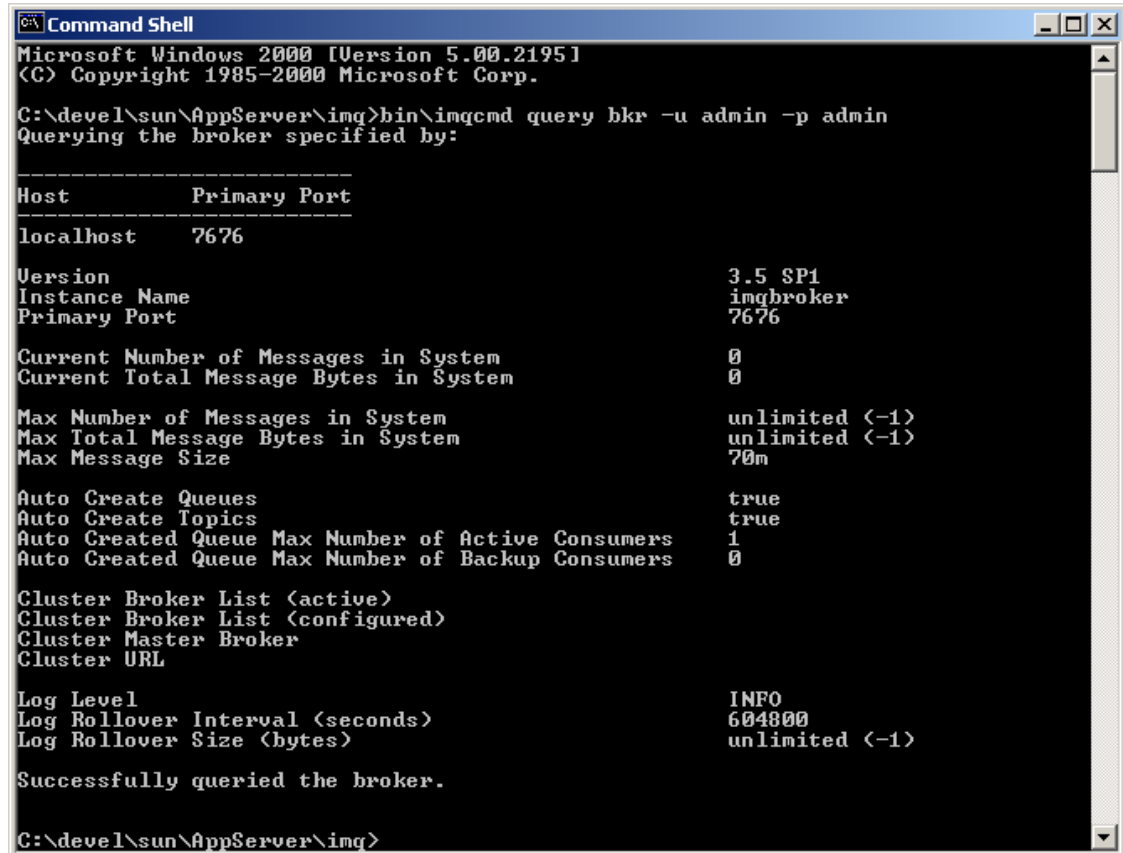
Test běhu SJS MQ

Zda MQ běží v pořádku lze zjistit dotazem:

- `imqcmd query bkr -u admin -p admin`

MQ odpoví:

Obrázek 1.6. Test běhu SJS MQ



```
Microsoft Windows [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\devel\sun\AppServer\imq>bin\imqcmd query bkr -u admin -p admin
Querying the broker specified by:

-----
Host           Primary Port
-----
localhost     7676

Version                3.5 SP1
Instance Name         imqbroker
Primary Port          7676

Current Number of Messages in System      0
Current Total Message Bytes in System     0

Max Number of Messages in System         unlimited (-1)
Max Total Message Bytes in System        unlimited (-1)
Max Message Size                          70m

Auto Create Queues                       true
Auto Create Topics                       true
Auto Created Queue Max Number of Active Consumers 1
Auto Created Queue Max Number of Backup Consumers 0

Cluster Broker List (active)
Cluster Broker List (configured)
Cluster Master Broker
Cluster URL

Log Level                               INFO
Log Rollover Interval (seconds)         604800
Log Rollover Size (bytes)                unlimited (-1)

Successfully queried the broker.

C:\devel\sun\AppServer\imq>
```

Kostra jednoduché aplikace

Demo *HelloWorldMessage* z adresáře `IMQ_HOME/demo/helloworld/helloworldmessage` má tyto hlavní prvky (viz tutoriál):

1. Import the interfaces and Message Queue implementation classes for the JMS API.

The `javax.jms` package defines all the JMS interfaces necessary to develop a JMS client.

```
import javax.jms.*;
```

2. Instantiate a Message Queue `QueueConnectionFactory` administered object. A `QueueConnection-`

Factory object encapsulates all the Message Queue-specific configuration properties for creating QueueConnection connections to a Message Queue server.

```
QueueConnectionFactory myQConnFactory =  
    new com.sun.messaging.QueueConnectionFactory();
```

ConnectionFactory administered objects can also be accessed through a JNDI lookup (see Looking Up ConnectionFactory Objects). This approach makes the client code JMS-provider independent and also allows for a centrally administered messaging system.

3. Create a connection to the message server. A QueueConnection object is the active connection to the message server in the Point-To-Point programming domain.

```
QueueConnection myQConn =  
    myQConnFactory.createQueueConnection();
```

4. Create a session within the connection. A QueueSession object is a single-threaded context for producing and consuming messages. It enables clients to create producers and consumers of messages for a queue destination.

```
QueueSession myQSess = myQConn.createQueueSession(false,  
    Session.AUTO_ACKNOWLEDGE);
```

The myQSess object created above is non-transacted and automatically acknowledges messages upon consumption by a consumer.

5. Instantiate a Message Queue queue administered object that corresponds to a queue destination in the message server. Destination administered objects encapsulate provider-specific destination naming syntax and behavior. The code below instantiates a queue administered object for a physical queue destination named “world”.

```
Queue myQueue = new com.sun.messaging.Queue("world");
```

Destination administered objects can also be accessed through a JNDI lookup (see Looking Up Destination Objects). This approach makes the client code JMS-provider independent and also allows for a centrally administered messaging system.

6. Create a QueueSender message producer. This message producer, associated with myQueue, is used to send messages to the queue destination named “world”.

```
QueueSender myQueueSender = myQSess.createSender(myQueue);
```

7. Create and send a message to the queue. You create a TextMessage object using the QueueSession object and populate it with a string representing the data of the message. Then you use the QueueSender object to send the message to the “world” queue destination.

```
TextMessage myTextMsg = myQSess.createTextMessage();  
myTextMsg.setText("Hello World");
```

```
System.out.println("Sending Message: " + myTextMsg.getText());  
myQueueSender.send(myTextMsg);
```

8. Create a `QueueReceiver` message consumer. This message consumer, associated with `myQueue`, is used to receive messages from the queue destination named "world".

```
QueueReceiver myQueueReceiver =  
    myQSession.createReceiver(myQueue);
```

9. Start the `QueueConnection` you created in Step 3. Messages for consumption by a client can only be delivered over a connection that has been started (while messages produced by a client can be delivered to a destination without starting a connection, as in Step 7).

```
myQueueConn.start();
```

10. Receive a message from the queue. You receive a message from the "world" queue destination using the `QueueReceiver` object. The code, below, is an example of a synchronous consumption of messages (see Message Consumption: Synchronous and Asynchronous).

```
Message msg = myQueueReceiver.receive();
```

11. Retrieve the contents of the message. Once the message is received successfully, its contents can be retrieved.

```
if (msg instanceof TextMessage) {  
    TextMessage txtMsg = (TextMessage) msg;  
    System.out.println("Read Message: " + txtMsg.getText());  
}
```

12. Close the session and connection resources.

```
myQSession.close();  
myQueueConn.close();
```