JBoss Messaging

Ovidiu Feodorov
Project Lead

March 1st 2005

Agenda

- Why rewriting JBossMQ?
- The current project status
  - ✓ Goals
  - ✓ The Messaging Core
  - ✓ The JMS Facade
- What is next?

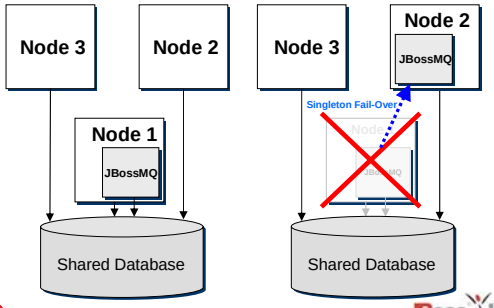
Why rewriting JBossMQ?

- JBossMQ evolved from SpyderMQ
  - ✓ A separate project which was integrated into JBoss
  - ✓ Originally designed as a standalone JMS provider
  - ✓ No HA support available
- The current JBossMQ has
  - ✓ Integration with the AS (the MDB Container)
  - ✓ A certain degree of HA support (HASingleton)

HA support in JBossMQ

- The JBossMQ HA support relies on the HASingleton mechanism
  - ✓ Only one running JBossMQ instance is available on a cluster
  - ✓ If the JBossMQ master node fails, the JBossMQ instance fails-over to the first available node
  - ✓ Features
    - Lossless recovery after fail-over for persistent messages targeted to *queues* and *topic durable subscribers*
    - Client notification via connection's ExceptionListener on fail-over
  - ✓ ... and what this actually means:
    - No in-memory replication, the HA failover will work only for PERSISTENT messages!
    - No transparent client fail-over

High Availability with HASingleton



The current status of JBossMQ

- JBossMQ entered maintenance mode since JBoss version 3.2.6
  - ✓ Only bug fixes will be applied on the JBossMQ's 3.2 and 4 branches
  - ✓ The new feature development effort is directed towards JBoss Messaging
  - ✓ JBoss Messaging scheduled to be released for Q2 2005

JBoss Messaging project goals

- Provide a fully compatible *JMS 1.1 implementation*
  - ✓ JBoss AS integration
  - ✓ Standalone
- Improve the *performance*
  - ✓ Benchmarking infrastructure
- Provide complete *load balancing* and *HA features*
  - ✓ In-memory replication
  - ✓ Distributed destinations
  - ✓ Transparent client fail-over
- Provide a *standard JMS API to JGroups*

7

JBoss Messaging project goals

- We intend to achieve the project goals by:
  - ✓ Creating from ground up an architecture based on reliable hardware multicast (JBoss Messaging Core)
  - ✓ Using existing JBoss subsystems:
    - Unified interceptors/AOP
    - JBoss/Remoting
    - JBoss/Persistence

8

High-level architectural overview

- JBoss Messaging is based on a generic Messaging Core
- The Core allows various facades to be installed in top of it
  - ✓ JMS
  - ✓ SMTP
  - ✓ etc.

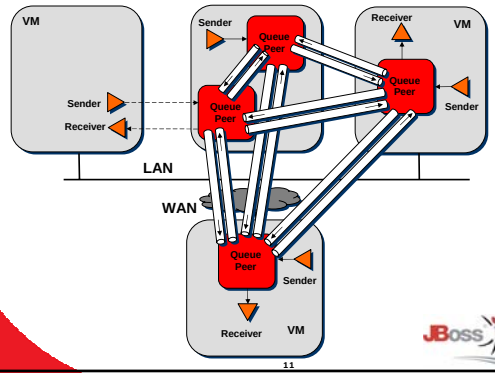
9

JBoss Messaging Core

- Framework built in top of JGroups
- Used to create *reliable* and *distributed* messaging transport systems
  - ✓ *Reliable* – supports guaranteed (once-and-only-once) delivery
  - ✓ *Distributed (Serverless)* – does not require a central server (single point of failure) – relies on distributed peers
- Uses a general messaging idiom, independent of the JMS API

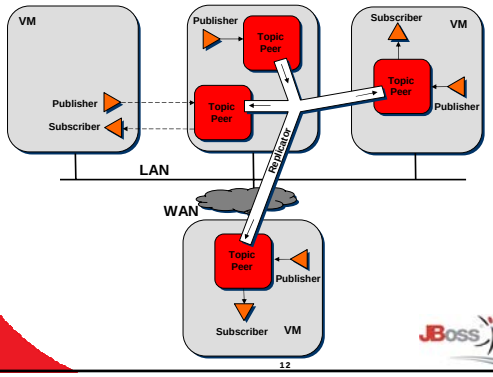
10

A Point-to-Point messaging domain



11

A Publish/Subscribe messaging domain



12

JBoss Messaging Core (contd.)

- The essence of the JBoss Messaging Core could be reduced to:
  - ✓ *Acknowledgment* (or handling of …) – the interface responsible with this aspect is org.jboss.messaging.core.Receiver
  - ✓ *Synchronicity/Asynchronicity* – the interface that handle this is org.jboss.messaging.core.Channel
  - ✓ *Persistence* – The org.jboss.messaging.core.MessageStrore and AcknowledgmentStore interfaces

Messaging Core Design Elements

- Messaging Core Interfaces:
  - ✓ *Routable/Message/MessageReference*
  - ✓ *Receiver*
  - ✓ *Channel*
  - ✓ *Router*
  - ✓ *MessageStore/AcknowledgmentStore*
- In-VM Primitives
  - ✓ LocalPipe
  - ✓ LocalDestination/LocalQueue/LocalTopic
- Distributed Primitives
  - ✓ Pipe
  - ✓ Replicator
  - ✓ Distributed Destinations (Queues and Topics)

Routable

- *Routable* – defines an atomic, self contained unit of data that flows though the messaging system
- The Core *routes* Routables (and hence the name)
- Must be serializable
- Defined by the org.jboss.messaging.core.Routable interface
- Can be declared *reliable* or *unreliable*

Routable (contd.)

```
public interface Routable extends Serializable
{
   public Serializable getMessageID();
   public boolean isReliable();
   public long getExpirationTime();
   public void putHeader(String name, Serializable value);
   public Serializable getHeader(String name);
   public Serializable removeHeader(String name);
   public Set getHeaderNames();
}
```


Message/MessageReference

- *Message* – a Routable that has a payload

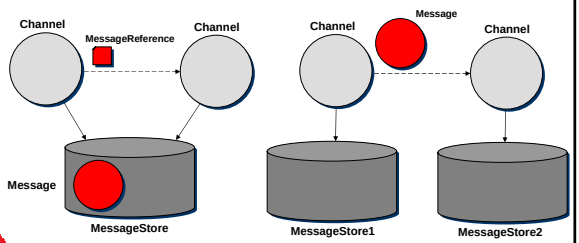
```
public interface Message extends Routable
{
   public Serializable getPayload();
}
```

- *MessageReference* – a "lightweight representative" of a message

```
public interface MessageReference extends Routable
{
    public Serializable getStorageID();
}
```


Message/MessageReference

- The Core prefers to handle MessageReference instead of Messages, whenever possible



Receiver

- *Receiver* – a component that handles Routables
- Its only concern is to provide a positive or negative, implicit or explicit acknowledgement
- Negative acknowledgments can then be stored and the delivery retried (reliable delivery)
- Defined by the org.jboss.messaging.core.Receiver

19

Receiver

```
public interface Receiver
{
   public Serializable getReceiverID();

   public boolean handle(Routable routable);
}
```

- A Receiver can consume the message or forward it
- The handle() return value represents the explicit acknowledgment
- Handle() may throw unchecked exceptions, as an implicit negative acknowledgment; these exceptions must must be dealt with

20

Channel

- *Channel* – abstraction that defines a message delivery mechanism which forwards a message from a sender to one or more Receivers
- The keyword: *synchronicity* (or the lack of it thereof)
- A Channel main concern is synchronously/asynchronously deliver messages to its Receivers
- A Channel is also a Receiver

21

Channel

```
public interface Channel extends Receiver
{
  public boolean isSynchronous();
  public boolean setSynchronous(boolean b);

  public boolean deliver();
  public boolean hasMessages();
  public Set getUnacknoweldged();

  public void setMessageStore(MessageStore ms);
  public MessageStore getMessageStore();
  public void setAcknowledgmentStore(AcknowledgmentStore as);
  public AcknowledgmentStore getAcknowledgmentStore();

}
```

22

Channel

- The Channel's responsibilities
  - ✓ To decide the Receiver(s) to forward the message to
  - ✓ To effectively forward the message, synchronously or asynchronously
- A channel can have zero, one or more *output* Receivers
- A Receiver never explicitly pulls a message from the Channel
- The Receiver doesn't even know it is "associated" with a Channel
- Unidirectional flow of messages

23

Synchronous Channel

- Always attempts synchronous delivery
- Uses the same thread that initiated the delivery to the Channel
- A delivery either succeeds or fails, the Channel doesn't hold messages (in memory or otherwise)
- Channel.handle() returning true means positive acknowledgment, false means *explicit* negative acknowledgment
- A synchronous Channel always provides *reliable delivery* (regardless whether the Routable was declared reliable or unreliable)

24

Asynchronous Channel

- First attempts synchronous delivery
- If not possible, acts as a middle man: holds the message and retries the delivery (the deliver() method)
- Channel.handle() returning true doesn't mean that all Receivers got the message …
- … it only means the Channel accepted the responsibility to deliver the message
- What about reliability? … next slide

25

Asynchronous Channels and Reliability

- Asynchronous Channels *must* be prepared to deal with the possibility of failure
- Must deterministically handle the situation when they hold message that have not been acknowledged, and a failure occurs
- For *unreliable* messages
  - ✓ No special precautions (risky but fast)
  - ✓ … or in-memory replication among peers
    - asynchronous (slower)
    - synchronous (even slower)
- For *reliable* messages
  - ✓ The Channel *must not* acknowledge the message unless the message is stored in a reliable store (slowest)
- An asynchronous Channel can reliably handle reliable messages only if has access to a MessageStore and AcknowledgmentStore

26

MessageStore

```
public interface MessageStore
{
   public Serializable getStoreID();

   public MessageReference store(Message m) throws Throwable;

   public Message retrieve(MessageReference r);
}
```

- A MessageStore stores (potentially large) Messages and returns (lightweight) MessageReferences

27

AcknowledgmentStore

```
public interface AcknowledgmentStore
{
 public Serializable getStoreID();

 public void storeNACK(Serializable messageID,
                       Serializable receiverID)throws Throwable;
 public void forgetNACK(Serializable messageID,
                        Serializable receiverID)throws Throwable;
 }
```

- The AcknowledgmentStore is a reliable repository for negative acknowledgments
- Negative acknowledgment: <messageID – receiverID>

28

Router

- A local (non-distributed) component that *synchronously* sends messages to zero, one or several of its Receivers
- Introduced to encapsulate the concept of *routing policy*
- A Router is also a Receiver, but not a Channel (does not hold messages)
- Channels use them as *routing delegates*, to decide whom to send messages.
- All Router's Receivers live in the same address space

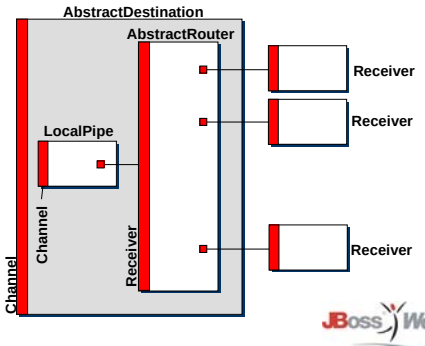
29

In-VM Primitives

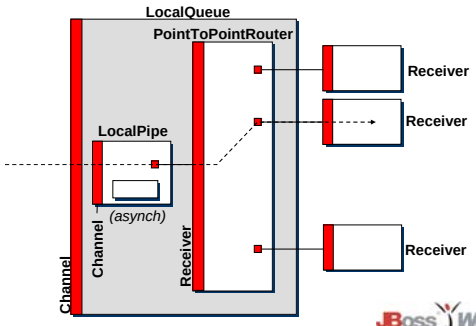
- LocalPipe
  - ✓ A Channel with only one output
  - ✓ Can be configured to be synchronous/asynchronous
- Local Destinations
  - ✓ LocalQueue
    - In-VM Point-to-Point Channel
    - Delivers the message to one and only one Receiver
    - Holds the messages if there are no Receivers
  - ✓ LocalTopic
    - In-VM Publish/Subscribe Channel
    - Delivers the message to all connected Receivers
    - Does not hold messages (synchronous behaviour)

30

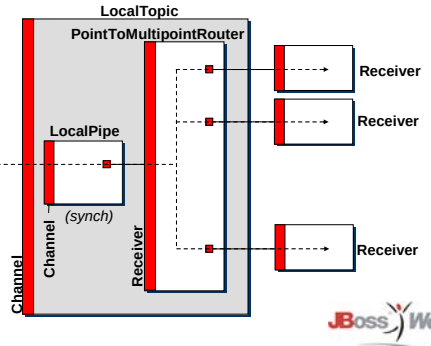
Abstract Destination



LocalQueue



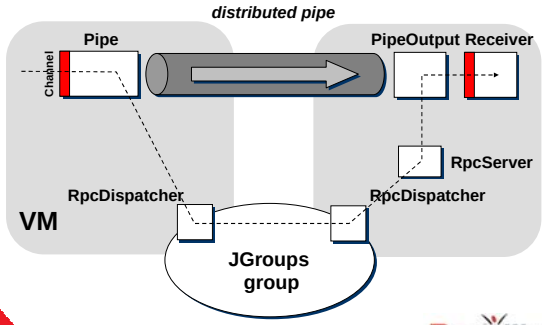
LocalTopic



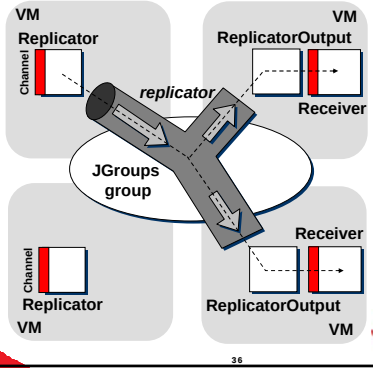
Distributed Primitives

- Pipe
  - ✓ Distributed Channel with only one output
  - ✓ Spans two (or more) address spaces
  - ✓ Can be configured to be synchronous/asynchronous
- Replicator
  - ✓ Distributed Channel that replicates a message to multiple receivers in different address spaces
  - ✓ Can be configured to be synchronous/asynchronous
  - ✓ Mostly used synchronously (otherwise it need access to MessageStore/AckStore)

Pipe



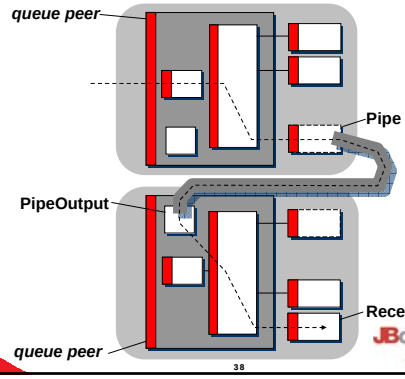
Replicator



Distributed Destinations

- Distributed Queue
  - ✓ LocalQueue extension
  - ✓ Multiple Queue *peers* coordinate into creating a *distributed queue*
- Distributed Topic
  - ✓ LocalTopic extension
  - ✓ Multiple Topic *peers* coordinate into creating a *distributed topic*

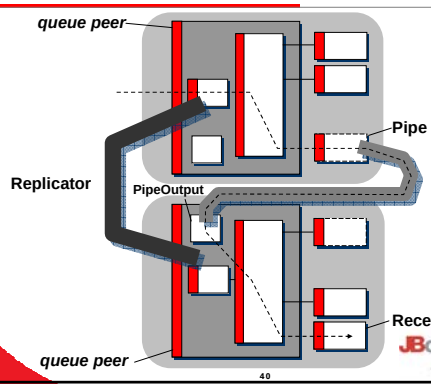
Distributed Queue



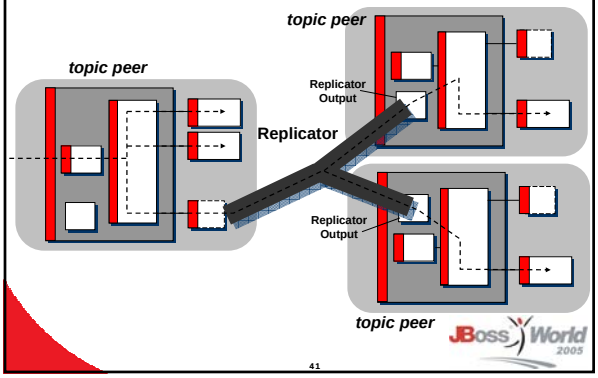
Replicated Queue

- Use case (for a LocalQueue):
  - ✓ unreliable messages are sent to a queue with no receivers
  - ✓ messages keep accumulated in the peer's buffer
  - ✓ the peer crashes
  - ✓ Result: messages are lost
- Even for a distributed queue, the situation is similar, because messages are buffered only on the peer that received them
- A distributed queue offers load balancing but not HA for unreliable message
- Solution: a replicated queue

Replicated Queue (contd.)



Distributed Topic



The JMS Facade

- Uses the the JBoss Unified Interceptors
- Uses a delegate pattern
  - ✓ The JMS API is sometimes unnecessarily complicated …
  - ✓ … so we use it as a façade to another façade – the delegates
- Also uses an interceptor pattern
  - ✓ FactoryInterceptor
  - ✓ ConnectionInterceptor
  - ✓ SessionInterceptor
  - ✓ TransactionInterceptor
  - ✓ PersistenceInterceptor
  - ✓ SecurityInterceptor
  - ✓ ClientInterceptor
  - ✓ JMSExceptionInterceptor
  - ✓ CloseInterceptor
- Work in progres

Conclusion

- A new JMS implementation is necessary to address the current HA and performance issues
- Implementation under way
  - ✓ JBoss Messaging Core – almost done
  - ✓ JMS Façade – work in progress
- The project page:
  http://www.jboss.org/wiki/Wiki.jsp?page=JBossMessaging
- Help wanted!

JBoss World 2005

43