Asynchronous Messaging

Unil | HEC | dop lab | Benoît Garbinato
distributed object programming lab

Fundamental idea

- Provide a communication abstraction that decouples collaborating distributed entities
  - Time decoupling ⇒ asynchrony
  - Space decoupling ⇒ anonymity
- Asynchrony ⇒ persistence of messages

Message-Oriented Middleware

- A Message-Oriented Middleware (MOM) is a software layer acting as a kind of "middle man" between distributed entities
- Most software companies offer middleware products that fall in the MOM category, e.g., IBM MQSeries, Oracle AQ, Sun Java System Message Queue, Microsoft Message Queueing, etc..

Broker & client library

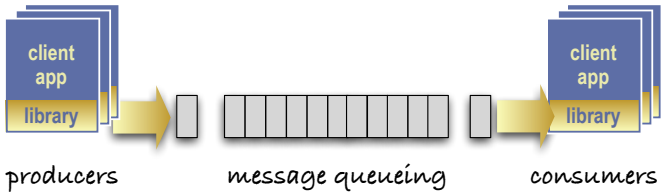
- A MOM is often based on a message broker and a client library.

Communication models

- Point-to-point model
  One-to-one communication between message producers and consumers, where each message is consumed by one and only one consumer
- Publish/Subscribe (pub/sub) model
  One-to-many communication where producers publish messages and all consumers that have subscribed receive them
- In both models, the notion of message is key

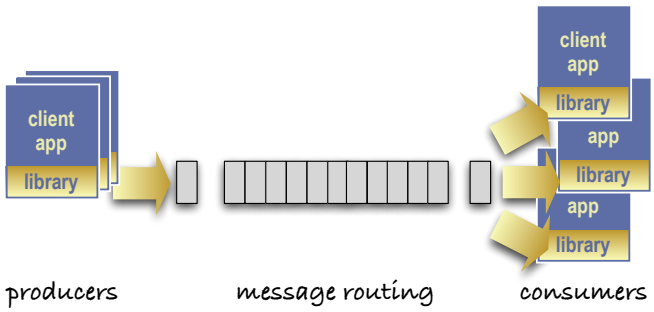
Point-to-Point

- Each message is received by only one consumer
- Messages are placed in a queue and are persisted until they are consumed
- This model can be used to load-balance tasks
  Caveat: fifo processing cannot be guaranteed



Publish/Subscribe

- Each message is received by all subscribers
- Messages are not persisted by default
- There exists various message routing variant:
  - topic-based
  - content-based
  - location-based
  - ...



J2EE overview

- J2EE stands for Java 2 platform, Enterprise Edition
- J2EE is the specification of a distributed multitiered application model for enterprise applications, presented as a coherent set of programming APIs
- Implementations of the J2EE specification are usually proposed in the form of application servers

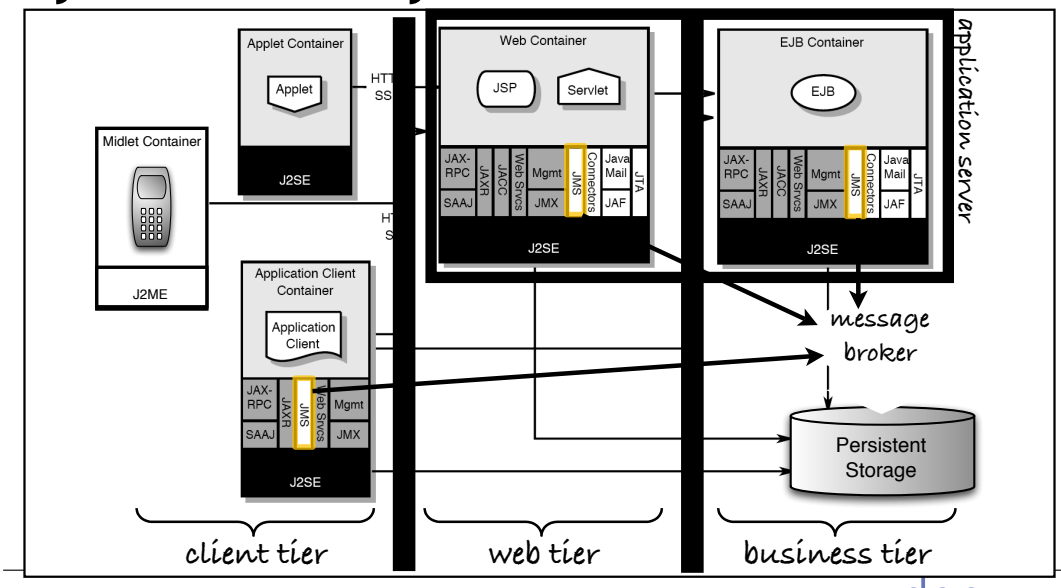
Enterprise Edition (J2EE)
Standard Edition (J2SE)
Micro Edition (J2ME)

} Java 2 platform

Java Messaging Service

- The Java Messaging Service (JMS) defines the asynchronous messaging standard of the J2EE platform
- JMS follows the general J2EE philosophy:
  - JMS is a specification
  - JMS implementations rely on existing products (IBM MQ Series, Oracle AQ, Sun Java System Message Queue, etc.)
  - JMS-based applications are portable across any JMS-compliant implementation

JMS in the J2EE architecture

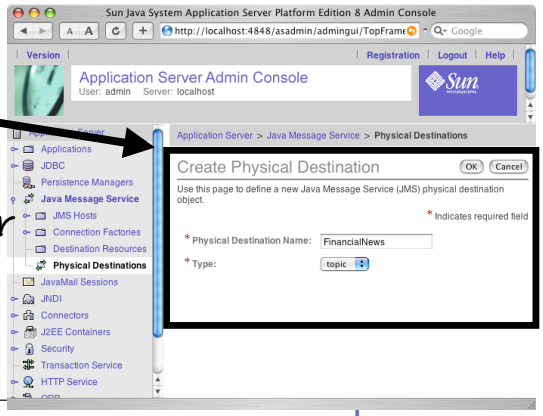


Execution time

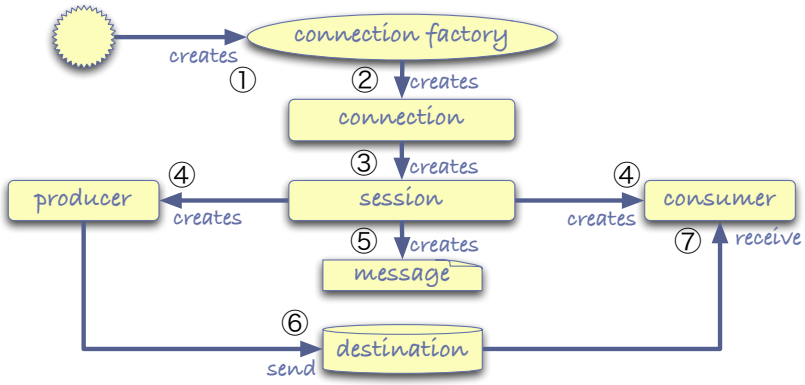
- A producer creates messages & sends them via the JMS API, specifying a message destination
- A consumer receives messages via the JMS API, specifying a message destination and an optional message selector
- A JMS-compliant product provides an implementation of the JMS API in the form of a client library that knows how to communicate natively with the message broker

Deployment time

- Start the message broker (usually via the J2EE application server)
- Create the adequate destinations
- Install the JMS client library on the producer & the producer, and start them



Unified programming model



Two communication models:

- point-to-point (destination = queue)
- publish/subscribe (destination = topic)

Development: publisher

```
public class NewsPublisher {
    static boolean moreNews= true;
    public static void main(String[] args) {
        String topicName= args[0];  String fileName= args[1];
    ①   TopicConnectionFactory connectionFactory = new com.sun.messaging.TopicConnectionFactory();
        TopicConnection connection= null;
        try {
    ②       connection= connectionFactory.createTopicConnection();
    ③       TopicSession session= connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
            Topic topic= session.createTopic(topicName);
    ④       TopicPublisher publisher = session.createPublisher(topic);
    ⑤       TextMessage message = session.createTextMessage();
            BufferedReader newsFeed = new BufferedReader(new FileReader(fileName));
            while (moreNews) {
                String theNews= getNextNews(newsFeed);
                message.setText(theNews);
                System.out.println("Publishing \"" + message.getText() + "\"");
    ⑥           publisher.publish(message);
            }
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.toString()); System.exit(1);
        }
    }
    ...
```

Development: subscriber

```
public class NewsSubscriber implements MessageListener {
    public static void main(String[] args) {
        String topicName= args[0];
        TopicConnectionFactory connectionFactory = new com.sun.messaging.TopicConnectionFactory();
        TopicConnection connection = null;
        try {
            connection = connectionFactory.createTopicConnection();
            TopicSession session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
            Topic topic= new com.sun.messaging.Topic(topicName);
            TopicSubscriber subscriber = session.createSubscriber(topic);
            MessageListener listener= new NewsSubscriber();
            subscriber.setMessageListener(listener);
            connection.start();
            synchronized (listener) { listener.wait(); }
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.toString()); System.exit(1);
        }
    }
⑦   public void onMessage(javax.jms.Message message) throws Exception {
        String theNews = ((TextMessage) message).getText();
        System.out.println("Learning that \"" + theNews + """);
        if (theNews.endsWith("There are no more news."))
            synchronized (this) { this.notify(); }
    }
    ...
```


Development: producer

```
public class OrderProducer {
    public static void main(String[] args) {
        String queueName= args[0];
        ConnectionFactory connectionFactory = new com.sun.messaging.ConnectionFactory();
        Connection connection= null;
        try {
            connection= connectionFactory.createConnection();
            Queue queue= new com.sun.messaging.Queue(queueName);
            Session session= connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer producer = session.createProducer(queue);
            BufferedReader kbdIn = new BufferedReader(new InputStreamReader(System.in));
            TextMessage message = session.createTextMessage();
            while (true) {
                String order= askForOrder(kbdIn, 3);
                message.setText(order);
                System.out.println("Sending order [" + message.getText() + "]");
     ⑥          producer.send(message);
            }
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.toString()); System.exit(1);
        }
    }
    ...
```

Development: consumer

```
public class OrderConsumer implements MessageListener {
    public static void main(String[] args) {
        String queueName = args[0];
        ConnectionFactory connectionFactory = new com.sun.messaging.ConnectionFactory();
        Connection connection = null;
        try {
            connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            Queue queue= new com.sun.messaging.Queue(queueName);
            MessageConsumer consumer = session.createConsumer(queue);
            MessageListener listener= new OrderConsumer();
            consumer.setMessageListener(listener);
            connection.start();
            synchronized (listener) { listener.wait(); }
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.toString()); System.exit(1);
        }
    }
⑦   public void onMessage(javax.jms.Message message) throws Exception {
        String order = ((TextMessage) message).getText();
        System.out.println("Passing order " + order + " on the market");
        if (order.equals("quit"))
            synchronized (this) { this.notify(); }
    }
    ...
```


Synchronous consumer

```
public class OrderSynchronousConsumer {
    public static void main(String[] args) {
        String queueName = args[0];
        ConnectionFactory connectionFactory = new com.sun.messaging.ConnectionFactory();
        Connection connection = null;
        try {
            connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            Queue queue= new com.sun.messaging.Queue(queueName);
            MessageConsumer consumer = session.createConsumer(queue);
            connection.start();
            while (true) {
⑦               Message m = consumer.receive();
                ...
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.toString()); System.exit(1);
        }
    }
}
```

Message format & types

- A JMS message is composed of three parts:
  - a header holding required fields for the client library and the message broker, e.g., priority, time-to-live, etc.
  - a list of optional properties, which act as meta-data used by the message selection mechanism
  - a body containing the actual data of the message

- There exists various types of messages, which differ in the type of data they carry in their body, e.g., **Message**, **TextMessage**, **ObjectMessage**, etc.

```
...
Message message = session.createMessage();
...
```


Message selectors

- By default, JMS provides topic-based pub/sub
- Thanks to message properties, JMS also support content-based pub/sub via message selectors
- A message selector is a string whose syntax is a subset of the SQL92 conditional expression syntax

```
...
Message message = session.createMessage();
message.setStringProperty("name", "Bob");
message.setIntProperty("age", 30);
message.setStringProperty("address", "Lausanne");
...
String selector= "name LIKE 'Max' OR (age > 18 OR address LIKE 'Lausanne')";
TopicSubscriber subscriber = session.createSubscriber(topic, selector, false);
...
```

Quality of Service (QoS)

- Parameterized Quality of Service (QoS) is usually offered by MOM products
- In JMS, the level of QoS depends on the following parameters:
  - message ordering, time-to-live & priorities
  - acknowledgement modes
  - durable subscriptions
  - delivery modes
  - transactions

Order, priority & time-to-live

- JMS specifies that messages are received in the order in which they were sent with respect to a given session and a given destination (commonly called FIFO order)
- JMS specifies no order across destinations or across sessions sending to the same destination
- The notion of priority allows programmers to have finer control over ordering, via the **`send()`** method
- Programmers can also specify how long the message broker should keep a message, via a time-to-live parameter passed to the **`send()`** method

priority time-to-live (in ms)

```
...
producer.send(aMessage, DeliveryMode.NON_PERSISTENT, 3, 5000);
...
```

Acknowledgement modes

- An acknowledgment informs the MOM (e.g., its underlying message broker) that the client has successfully received a message
- JMS supports three acknowledgment modes:

`AUTO_ACKNOWLEDGE`	the session automatically acknowledges the receipt of each message
`CLIENT_ACKNOWLEDGE`	the client acknowledges programmatically, invoking `acknowledge()` on each message
`DUPS_OK_ACKNOWLEDGE`	more efficient variant of `AUTO_ACKNOWLEDGE` that can result is duplicate messages in case of failures

```
...
Session session= connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
...
```


Delivery modes

- In JMS, there exists two delivery modes:

`NON_PERSISTENT`	most efficient but less reliable, since messages are guaranteed to be delivered at most once, i.e., some might be lost, e.g., due to some failure (power outage)
`PERSISTENT`	most reliable, since messages are guaranteed to be delivered once and only once; this is usually achieved by persisting sent messages on stable storage and keeping them until they are acknowledged

- The delivery mode can be specified at the producer level or each time a messages is sent:

```
...
MessageProducer producer = session.createProducer(queue);
producer.setDeliveryMode(DeliveryMode.PERSISTENT);
producer.send(aMessage, DeliveryMode.NON_PERSISTENT, 0, 0);
...
```

Durable subscriptions

- With pub/sub, messages are only received by subscribers present at the time of the publication
- A durable subscriber is one that wants to receive all messages published on a topic, even those published when the subscriber is inactive, i.e., when it has no associated subscriber object
- In order to tell the message broker what messages are still to be received by a durable subscriber, the latter must provide a unique name

```
...
TopicSubscriber subscriber= session.createDurableSubscriber(topic, "Bob");
session.unsubscribe("Bob");
...
```

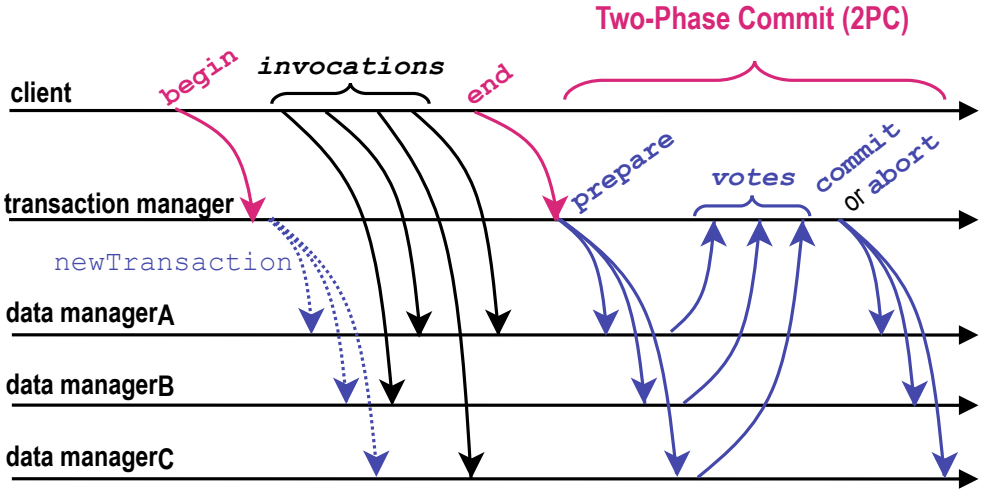

Transactions (1)

- A transaction allows a group of messages to be managed as a single unit of work
- In JMS, transactions are managed by the session
- The decision to have a session transacted must be taken at creation time:

```
...
Session session= connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
...
```

- As soon as messages are sent or received via a transacted session, the transaction starts, i.e., sent/received messages are grouped as a one unit of work

Transactions (2)



Transactions (3)

- When method `commit()` or method `rollback()` is called on the transacted session, the current transaction terminates and a new one is started
- Transaction termination affects producers and consumers in the following manner:

  Producer - what happens to messages sent during the transaction?
  - Commit all grouped messages are effectively sent
  - Rollback all grouped messages are disposed

  Consumer - what happens to messages received during the transaction?
  - Commit all grouped messages are disposed
  - Rollback all grouped messages are recovered, i.e., they might be received again in the next transaction

Questions?