Part VI

Message-Oriented Middleware

Communication via RMI / RPC

- causes tight coupling of communicating systems
- e.g. Java RMI, CORBA, EJB, DCOM
- enables type checking at compile (or run) time
- little overhead (for marshaling)
- modification of large systems is complex
- sender blocked, if receiver is temporarily unavailable

Communication via Messages

- loose coupling
- no type checking (in general)
- more overhead (e.g. for queueing, metadata)
- possibly with guaranteed delivery

  (even if receiver is temporarily unavailable)
- sender can continue, even if receiver is unavailable
- modification relatively easy

Communication Models

- Synchronous Communication
  - sender sends message and waits, until answer arrives
  - receiver receives message, processes it, and sends an answer
- Asynchronous Communication
  - sender sends a message and continues to work (without waiting)
  - the receiver processes the message

Communication Variants

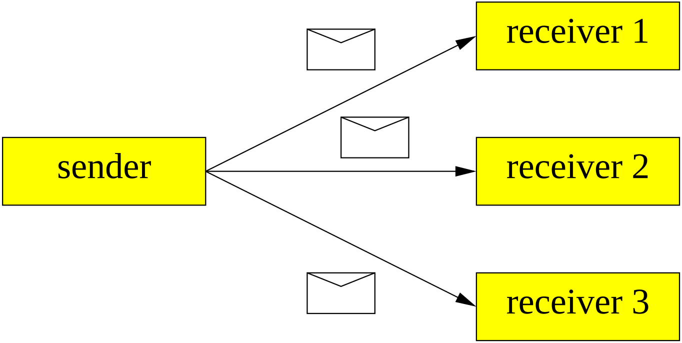
Synchronous Polling

- sender sends a message and continues to work
- receiver processes received message
- sender aks periodically for the result
- if the result is not yet available,
  the sender continues with other work
- if result is available, it is transmitted and the sender processes it

Communication Variants

Asynchronous Broadcasting

- sender sends message to several receivers and continues to work
- every receiver receives the message and processes it

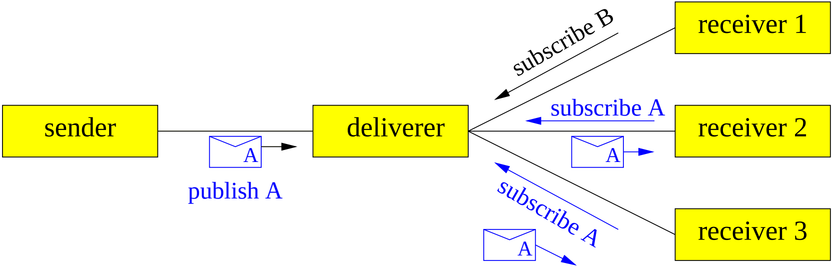


Communication Variants

Asynchronous Publish / Subscribe

- similar to broadcasting,

  but receivers subscribe to subjects at deliverer
- only subscribers receive a message



Message-Oriented Middleware

- propagates messages from sender to receiver(s)
  (often via message server, message broker)
- services w.r.t. messages:
  - creation, propagation, delivery
  - storage
  - transaction handling

Properties of Message-Oriented Middleware

- Advantages
  - asynchronous communication allows the emulation of other models
  - high interoperabilty between heterogeneous systems
  - appropriate for loosely coupled systems
- Disdvantages
  - (in general) not type-safe
  - overhead for queueing, metadata, marshaling and demarshaling (often XML-based)
  - (in general) no distributed object model
  - message broker is single point of failure (→ high availability!)
  - testing and debugging difficult

Market Overview: Message-Oriented Middleware

- IBM Websphere MQ (offers APIs: JMS, MQI, AMI, CMI)
- Sun Java System Message Queue (offers: JMS)
- MSMQ Microsoft Message Queue Server
  (interoperable with IBM MQ)
- ObjectWeb JORAM (open source, offers: JMS)
- TIBCO Enterprise Message Service (supports JMS and .NET)
- HornetQ (open source JMS compliant messaging by JBoss)
- . . .

Java Message Service

- specifies API and protocols for messaging
- Communication Variants:
  - asynchronous point-to-point-communication
  - asynchronous publish / subscribe
  - asynchronous request / reply
  - synchronous request / reply (blocking)
  - synchronous unidirectional communication (with ack.)

JMS Queues and Topics

- Queue: message queue for n:m communication (one receives)
- Topic: publish/subscribe channel for n:m communication (all subscribers receive)
- queues and topics are incompatible

Queues and Topics

- Queue:
  - every message is delivered only once
  - a message is stored, until the receiver fetches it
  - the order of messages is not guaranteed
- Topic:
  - variant 1: non-durable
    - only current subscribers receive messages
    - posibbly messages are not delivered at all (if no subscriptions)
  - variant 2: durable
    - also subscriptions after publication are taken into account
    - this requires messages to be stored (e.g. until they expire)

Structure of a JMS Message

- a JMS message consists of header, properties, and body
- the header contains metadata (receiver, expiration time, . . .)
- the properties contain additional, freely structured information (e.g. primitive data, strings)
- the body contains the content

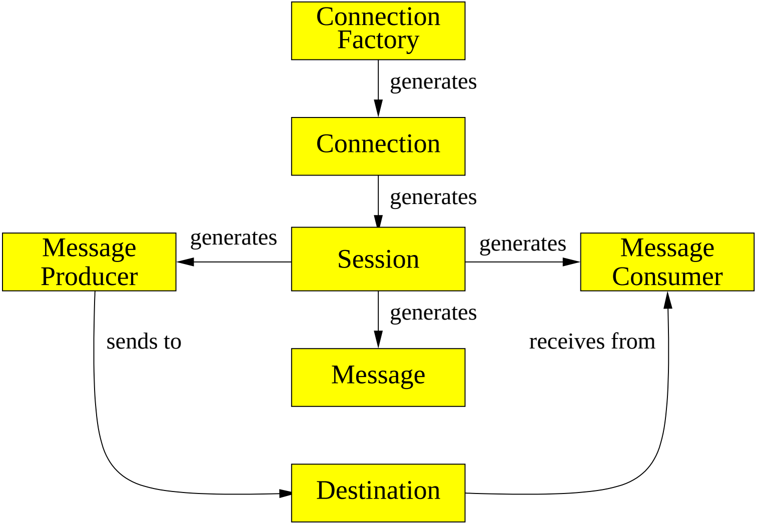
JMS Message Types

- `TextMessage:` transmits strings
- `MapMessage:` for name-value pairs of primitive data types
- `ObjectMessage:` transmits serialized object
- `BytesMessage:` delivers a byte stream
- `StreamMessage:` for a stream of primitive values

Interfaces for Sending and Receiving Messages

- `javax.jms.ConnectionFactory:`
  constructs connection between JMS client and JMS provider
- `javax.jms.Connection:` encapsulates connection
- `javax.jms.Session:` session for sending or receiving
- `javax.jms.Destination:`
  destination of a message (queue or topic)
- `javax.jms.MessageProducer/MessageConsumer:`
  sender / receiver of a message
- destinations may exist only temporarily

JMS Communication: Sequence of Operations



Sending a Message to a Queue

```
ConnectionFactory cf =
   (ConnectionFactory) ic.lookup("ConnectionFactory");
Connection con = cf.createConnection();
Session session =
   con.createSession(false,Session.AUTO_ACKNOWLEDGE);

con.start();
Queue queue = (Queue) ic.lookup("queue/testQueue");
MessageProducer sender = session.createProducer(queue);
MapMessage message = session.createMapMessage();
// fill message with content
sender.send(message);
con.close();
```

Fill MapMessage with Content

```
message.setInt("quantity",a.length);
for(int i=0; i<a.length;i++)
   message.setDouble("arg"+i,a[i]);

// include reference to temporary queue for the answer
message.setJMSReplyTo(temporaryQueue);
```

Sending a Message to a Topic

```
ConnectionFactory cf =
   (ConnectionFactory) ic.lookup("ConnectionFactory");
Connection con = cf.createConnection();
Session session =
   con.createSession(false,Session.AUTO_ACKNOWLEDGE);

con.start();
Topic topic = (Topic) ic.lookup("topic/testTopic");
MessageProducer publisher = session.createProducer(topic);
MapMessage message = session.createMapMessage();
// fill message with content
publisher.send(message);
con.close();
```

Asynchronous Reception with Message-Driven Bean

- implement interface `javax.jms.MessageListener` and method `public void onMessage`
- register as message listener

```
public class ExampleListener
             implements javax.jms.MessageListener {
  public void onMessage(Message message) {
    // process message
  }
}
```

Receive Message Synchronously

```
InitialContext ic = new InitialContext();
ConnectionFactory cf =
   (ConnectionFactory) ic.lookup("ConnectionFactory");
Connection con = cf.createConnection();
Session session =
   con.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = (Queue) ctx.lookup("queue/testQueue");
MessageConsumer consumer = session.createConsumer(queue);

con.start();
StreamMessage msg = (StreamMessage) consumer.receive(10000);
// receive further messages, if needed
qc.close();
```

- reception from topic analogously

Message Selectors

- messages can be filtered depending an their properties, e.g.
- Sender:

```
message.setStringProperty("ResultType","Median");
```

- Receiver:

```
consumer = session.createConsumer("ResultType=´Median´");
msg = (StreamMessage) consumer.receive(0);
```

- unfitting messages are ignored by receiver

Message-Driven Bean

- acts as receivers of JMS messages
- can receive from queues or topics
- encapsulates the reception of a message by the `onMessage()` method
- only the message processing has to be implemented

Answering a Message

- MDBs do not automatically send an answer
- this has to be done explicitly (as explained above)
- often, a temporary queue is used for transmitting an answer
- Sender:

```
Queue temp = session.createTemporaryQueue();
// ...
message.setJMSReplyTo(temp);
```

- MDB:

```
Destination replyTo = message.getJMSReplyTo();
MessageProducer producer = session.getProducer(replyTo);
// generate answer message and send it ...
```

Annotations for Configuring a MDB

```
import javax.ejb.ActivationConfigProperty;
import javax.ejb.Messagedriven;
import javax.jms.Message;
import javax.jms.MessageListener;
@MessageDriven(activationConfig={
  @ActivationConfigProperty(propertyName="destinationType",
                propertyValue="javax.jms.Topic"),
  @ActivationConfigProperty(propertyName="destination",
                propertyValue="topic/myTopic"),
  @ActivationConfigProperty(propertyName="messageSelector",
                propertyValue="MessageType=´Cancelation´"),
  @ActivationConfigProperty(propertyName="subscriptionDurability",
                propertyValue="Durable"),
  @ActivationConfigProperty(propertyName="clientId",
                propertyValue="MyId12345"),
  @ActivationConfigProperty(propertyName="subscriptionName",
                propertyValue="MyName123456")})
public class MyMDB implements MessageListener{
  public void onMessage(Message m){...}
}
```

4. Messaging and .NET

- message oriented communication within .NET:
  for example with WCF
- here: commmunication with JMS (HornetQ) via Stomp protocol
- .NET implementation: Apache.NMS.Stomp
- API similar to Java API of JMS
- see tutorial and example on web page

Subscribing to Topic (C#)

```
class Program{
  private static ISession session;
  static void Main(string[] args) {
    Uri connectUri = new Uri("stomp:tcp://localhost:61613");
    IConnectionFactory factory = new NMSConnectionFactory(connectUri);
    using (IConnection connection = factory.CreateConnection())
    using (session = connection.CreateSession()) {
      IDestination destination =
        session.GetDestination("topic://TaskTopic");
      using (IMessageConsumer consumer =
          session.CreateConsumer(destination)) {
        connection.Start();
        consumer.Listener += new MessageListener(OnMessage);
        Console.ReadLine(); } }
  }
  private static void OnMessage(IMessage message) { ... }
}
```

Receiving Messages

```
private static void OnMessage(IMessage message) {
  IBytesMessage msg = (IBytesMessage) message;
  int count = msg.ReadInt32();
  double[] a = new double[count];
  for (int i = 0; i < count; i++)
    a[i] = msg.ReadDouble();

  double result = Median(a);

  ...
}
```

Sending Messages

```
private static void OnMessage(IMessage message) {
  ...

  IDestination replyTo = msg.NMSReplyTo;
  using (IMessageProducer producer =
      session.CreateProducer(replyTo)) {
    IBytesMessage reply = producer.CreateBytesMessage();
    reply.WriteDouble(result);
    reply.Properties.SetString(ResultType, "Median");
    producer.Send(reply);
  }
}
```