Event-Based Systems

Sara Bouchenak

Sara.Bouchenak@imag.fr
http://membres-liglab.imag.fr/bouchenak/teaching/

Agenda

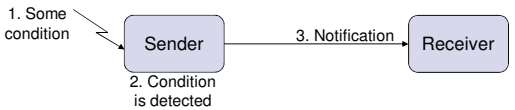
Lecture, Tuesday, 09:45 – 12:45	Lab, Tuesday, 09:45 – 12:45
Introduction to distributed systems	
	Distributed applications with RMI (Part I)
Distributed Web applications	
	Distributed applications with RMI (Part II)
Interruption week	
Event-based systems & MapReduce systems	
	Distributed Web applications with Servlets (Part I)
Cloud computing	
	Distributed Web applications with Servlets (Part II)
Advanced techniques for efficient distributed systems	
	Caching with Memcached
Event-based systems & MapReduce systems	
Interruption week	
Advanced techniques for dependable distributed systems	
	Evaluation

Event-based systems: Outline

- Definitions
- Examples
- Subscription models
- Notification delivery
- Simple JMS examples
- Conclusion

Definitions

- **Event**
  - An occurrence of some kind
  - A detectable condition that can trigger a *notification*
  - Examples
    - Time is 1:00 pm
    - Mouse click
- **Notification**
  - A message informing the recipient that something happened
  - An *event*-triggered signal sent to a run-time-defined recipient



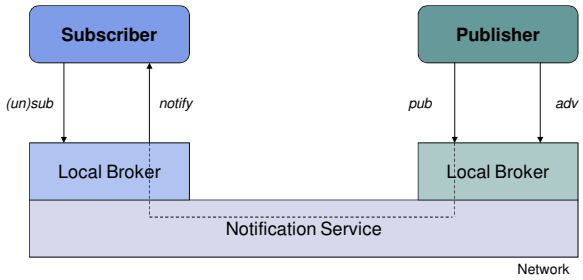
Definitions (2)

- **Event publisher**
  - Entity that is able to detect events
  - A.k.a. *event source*, *event producer*, *sender*
- **Event subscriber**
  - Entity that receives event notifications
  - A.k.a. *event subscriber*, *event consumer*, *event handler*, *notification target*, *receiver*
- **Event-based system**
  - A.k.a. *publish/subscribe system*, *message-oriented middleware* (MOM), *message-based system*, store-and-forward system

Definitions (3)

- **Notification filter**
  - Matches a set of notifications
  - Example: mouse click on a button
- **Content-based publish/subscribe**
  - Subscribers issue subscriptions that contain filters
  - Filters match a set of notifications a subscriber is interested in
- **Clients**
  - Event publishers and event subscribers

Publish-subscribe system



Examples

- Applications
  - Graphical user interface (e.g. Java Swing)
  - System monitoring
  - Workflow systems
  - Active database systems with event-condition-action (ECA) rules
- Event-based system implementations
  - Java Message Service
  - CORBA Event and Notification Service

Outline

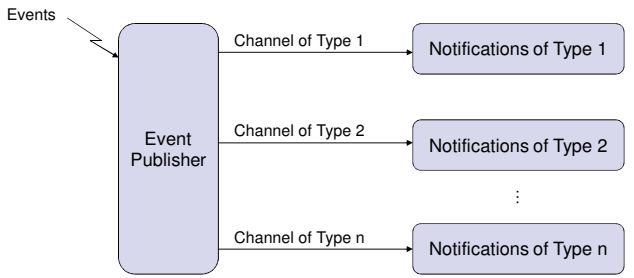
- *Definitions*
- *Examples*
- **Subscription models**
- Notification delivery
- Simple JMS examples
- Conclusion

Subscription models

- Channel
- Type
- Filter
- Group

Subscription models - Channels

Subscription models - Types



Subscription models - Filters

- Content-based filtering
  - Notifications carry payload representing content
  - Subscriptions describe what content is of interest
  - Notification service applies filter to content of all incoming notifications
  - Only notifications that pass filter are delivered to subscribers
  - Example:
    - In a News service, a client subscribes only for news stories that contain a set of words

Subscription models – Filters (2)

- Attribute-based filtering (*topic-based* or *subject-based* filtering)
  - A system is classified using a set of attributes
  - Subscriptions identify notifications of interest by specifying constraints in the attribute space
  - Only notifications that fulfill attribute constraints are delivered to subscribers
  - Example:
    - In a News service, a client subscribes for news stories with the attribute constraints
      [news.date: today] and [news.location: Haiti]

Subscription models – Filters (3)

- Sequence-based filtering (*composite events*)
  - Subscribers describe a certain sequence of events of interest
  - When events occur that fit the sequence, a notification is sent to subscribers
  - Example:
    - In a Weather and traffic news service, a client subscribes for the following composite events
      [trafficAlert.route: 66] and
      [trafficAlert.date: today] and
      [precipitationAlert.location: California] and
      [precipitationAlert.date: today]

Subscription models – Filters (4)

- Translation-based filtering (*message transforming*)
  - Notification content:
    filter converts content from one language to another, from one format to another
  - Notification type:
    filter converts content from one type to another

Subscription models – Groups

- Group
  - Set of subscribers for the same events and same filters
  - Simplify subscription and notification-delivery process
  - A subscriber belonging to a group receives all notifications the group is subscribed to
  - A subscriber may belong to multiple groups

Subscription models – Groups (2)

- Predefined groups
  - Groups defined by an event publisher or notification service
  - Example:
    - A notification service of a banking system predefines groups: administrator, supervisor, and technician.
- Implicit groups
  - Groups set up automatically when two or more subscribers request the same subscription (for notification optimization purposes)
  - Subscribers are generally not aware that they belong to an implicit group

Subscription models – Groups (3)

- Explicit groups
  - Created at runtime via a system-dependent operation
  - Subscribers can then join the group
- Location groups
  - A group associated with the the location of the subscribers

Notification delivery

- Delivery order
  - Event notifications may reach final destination in an order different from the order in which they were sent
  - Possible causes: system loading, propagation latency
- Causal order
  - Notifications reach their destination ordered in the same way as events that triggered them

Outline

- *Definitions*
- *Examples*
- *Subscription models*
- *Notification delivery*
- **Simple Java Messaging Service (JMS) examples**
- Conclusion

A simple point-to-point JMS example

- The sending program, *SimpleQueueSender.java*, performs the following steps:
  1. Performs a Java Naming and Directory Interface (JNDI) API lookup of the QueueConnectionFactory and queue
  2. Creates a connection and a session
  3. Creates a QueueSender
  4. Creates a TextMessage
  5. Sends one or more messages to the queue
  6. Sends a control message to indicate the end of the message stream
  7. Closes the connection in a finally block, automatically closing the session and QueueSender

SimpleQueueSender

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

public class SimpleQueueSender {

   /**
    * Main method.
    *
    * @param args     the queue used by the example and, optionally, the number of messages to send
    */
   public static void main(String[] args) {
      String              queueName = null;
      Context             jndiContext = null;
      QueueConnectionFactory  queueConnectionFactory = null;
      QueueConnection         queueConnection = null;
      QueueSession            queueSession = null;
      Queue               queue = null;
      QueueSender             queueSender = null;
      TextMessage             message = null;
      final int           NUM_MSGS;

      if ( (args.length < 1) || (args.length > 2) ) {
         System.out.println("Usage: java SimpleQueueSender " +  "<queue-name> [<number-of-messages>]");
         System.exit(1);
      }
      queueName = new String(args[0]);
      if (args.length == 2){
         NUM_MSGS = (new Integer(args[1])).intValue();
      } else {
         NUM_MSGS = 1;
      }
```


SimpleQueueSender (2)

```
      /*
       * Create a JNDI API InitialContext object if none exists yet.
       */
      try {
         jndiContext = new InitialContext();
      } catch (NamingException e) {
         System.out.println("Could not create JNDI API " + "context: " + e.toString());
         System.exit(1);
      }

      /*
       * Look up connection factory and queue.  If either does not exist, exit.
       */
      try {
         queueConnectionFactory = (QueueConnectionFactory) jndiContext.lookup("QueueConnectionFactory");
         queue = (Queue) jndiContext.lookup(queueName);
      } catch (NamingException e) {
         System.out.println("JNDI API lookup failed: " + e.toString());
         System.exit(1);
      }
```

SimpleQueueSender (3)

```
/*
 * Create connection.
 * Create session from connection; false means session is not transacted.
 * Create sender and text message.
 * Send messages, varying text slightly.
 * Send end-of-messages message.
 * Finally, close connection.
 */
try {
   queueConnection = queueConnectionFactory.createQueueConnection();
   queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
   queueSender = queueSession.createSender(queue);
   message = queueSession.createTextMessage();
   for (int i = 0; i < NUM_MSGS; i++) {
      message.setText("This is message " + (i + 1));
      queueSender.send(message);
   }

   /*
    * Send a non-text control message indicating end of messages.
    */
   queueSender.send(queueSession.createMessage());
} catch (JMSException e) {
   System.out.println("Exception occurred: " + e.toString());
} finally {
   if (queueConnection != null) {
      try {
         queueConnection.close();
      } catch (JMSException e) {}
   }
}
```


A simple point-to-point JMS example (2)

- The receiving program, *SimpleQueueReceiver.java*, performs the following steps:
  1. Performs a JNDI API lookup of the QueueConnectionFactory and queue
  2. Creates a connection and a session
  3. Creates a QueueReceiver
  4. Starts the connection, causing message delivery to begin
  5. Receives the messages sent to the queue until the end-of-message-stream control message is received
  6. Closes the connection in a finally block, automatically closing the session and QueueReceiver

SimpleQueueReceiver

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

public class SimpleQueueReceiver {

   /**
    * Main method.
    *
    * @param args    the queue used by the example
    */
   public static void main(String[] args) {
      String              queueName = null;
      Context             jndiContext = null;
      QueueConnectionFactory  queueConnectionFactory = null;
      QueueConnection         queueConnection = null;
      QueueSession            queueSession = null;
      Queue               queue = null;
      QueueReceiver           queueReceiver = null;
      TextMessage             message = null;

      /*
       * Read queue name from command line and display it.
       */
      if (args.length != 1) {
         System.out.println("Usage: java " + "SimpleQueueReceiver <queue-name>");
         System.exit(1);
      }
      queueName = new String(args[0]);
```


SimpleQueueReceiver (2)

```
/*
 * Create a JNDI API InitialContext object if none exists yet.
 */
try {
   jndiContext = new InitialContext();
} catch (NamingException e) {
   System.out.println("Could not create JNDI API " + "context: " + e.toString());
   System.exit(1);
}

/*
 * Look up connection factory and queue.  If either does not exist, exit.
 */
try {
   queueConnectionFactory = (QueueConnectionFactory) jndiContext.lookup("QueueConnectionFactory");
   queue = (Queue) jndiContext.lookup(queueName);
} catch (NamingException e) {
   System.out.println("JNDI API lookup failed: " + e.toString());
   System.exit(1);
}
```

SimpleQueueReceiver (3)

```
/*
 * Create connection.
 * Create session from connection; false means session is not transacted.
 * Create receiver, then start message delivery.
 * Receive all text messages from queue until a non-text message is received indicating end of message stream.
 * Close connection.
 */
try {
    queueConnection = queueConnectionFactory.createQueueConnection();
    queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
    queueReceiver = queueSession.createReceiver(queue);
    queueConnection.start();
    while (true) {
        Message m = queueReceiver.receive(1);
        if (m != null) {
            if (m instanceof TextMessage) {
                message = (TextMessage) m;
                System.out.println("Reading message: " + message.getText());
            } else {
                break;
            }
        }
    }
} catch (JMSException e) {
    System.out.println("Exception occurred: " + e.toString());
} finally {
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSException e) {}
    }
}
```


References

- Ted Faison. *Event-Based Programming: Taking Events to the Limit*. Springer-Verlag, 2006.
- Michael Jaeger. *Self-Managing Publis/Subscribe Systems: Foundations, Algorithms and Analysis*. VDM Verlag, 2007.