

Introduction to the Java Message Service (JMS) Standard

Martin Erzberger
Softwired Inc.

What you will learn

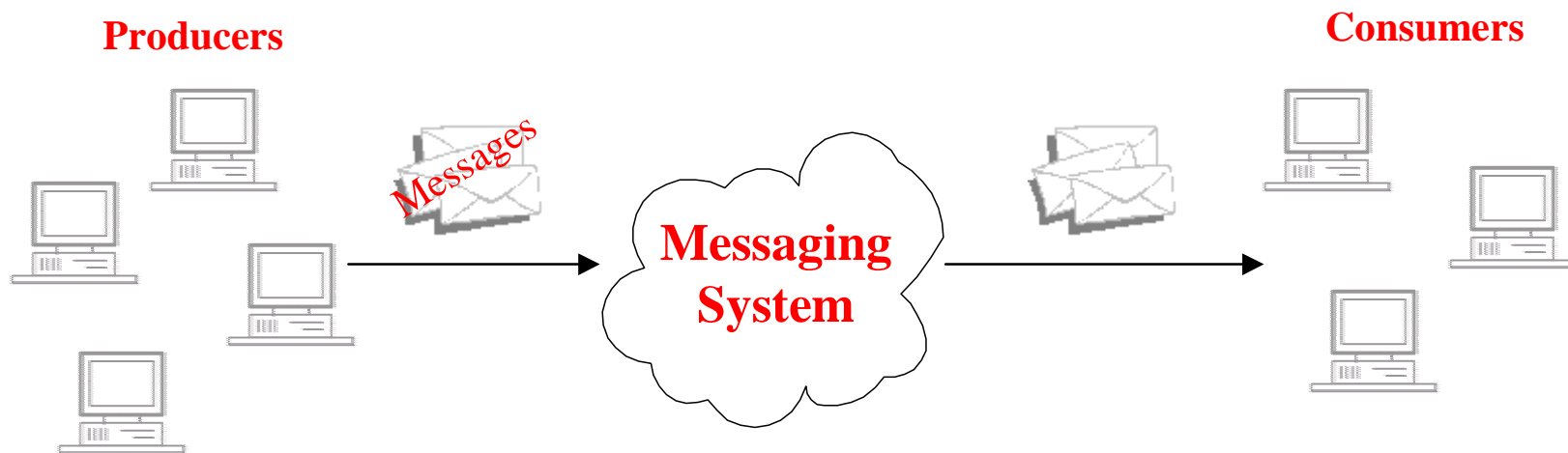
- What is JMS?
- How does JMS work?
- Why should I use JMS?
- How does a JMS application look like?

What is JMS?

- JMS stands for "Java Message Service"
- Sun's Definition: *JMS is an API for accessing enterprise **messaging systems** from Java programs.*
- JMS is for **messaging systems** what JDBC is for *database systems*: A standardized API to access them.

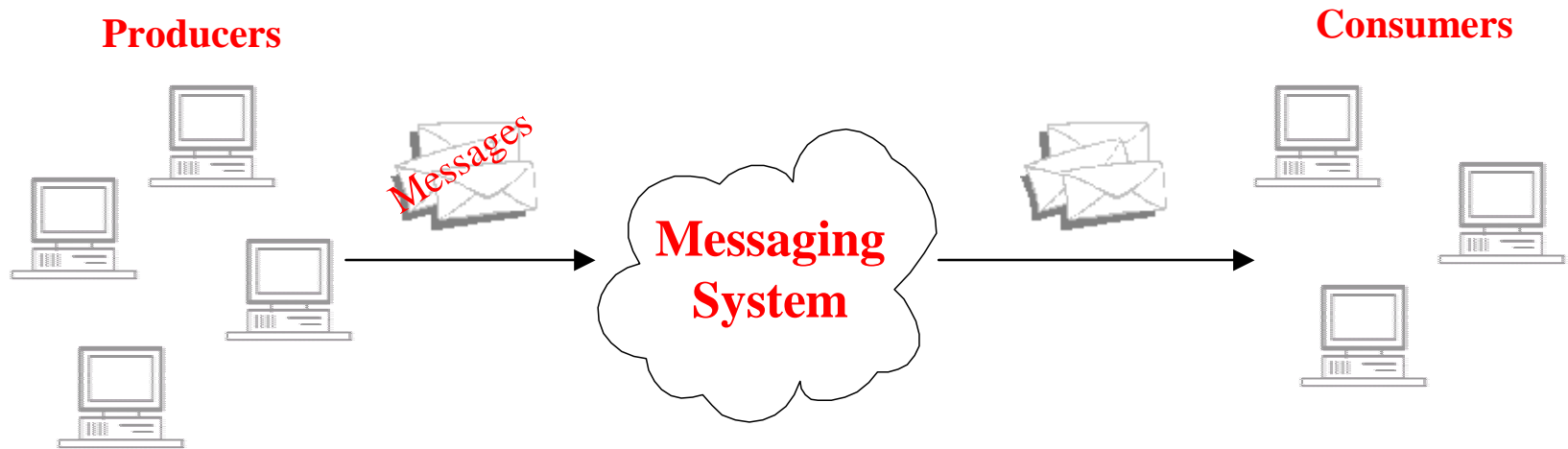
What are Messaging Systems?

- Crude analogy: E-mail systems for applications

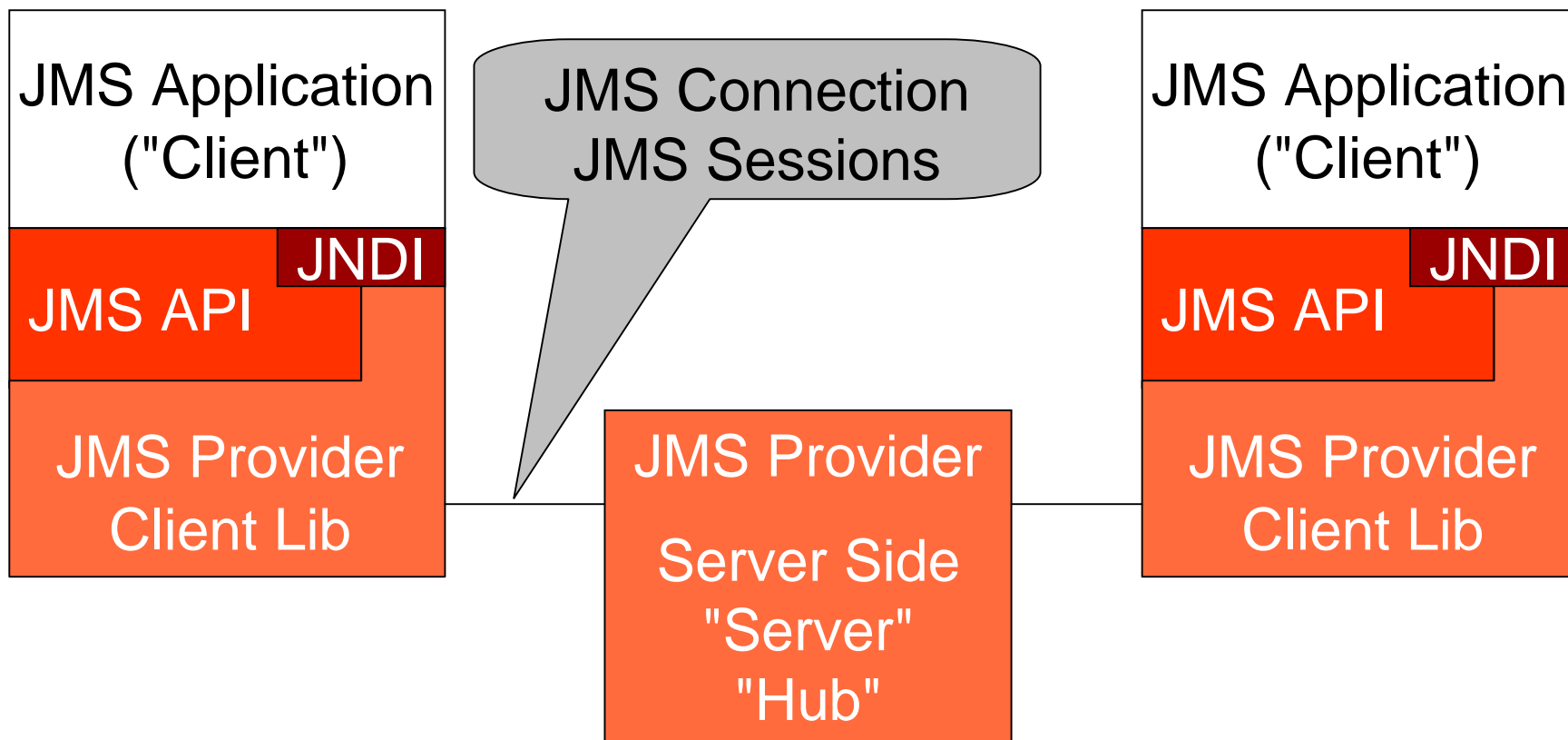


Formal Definition

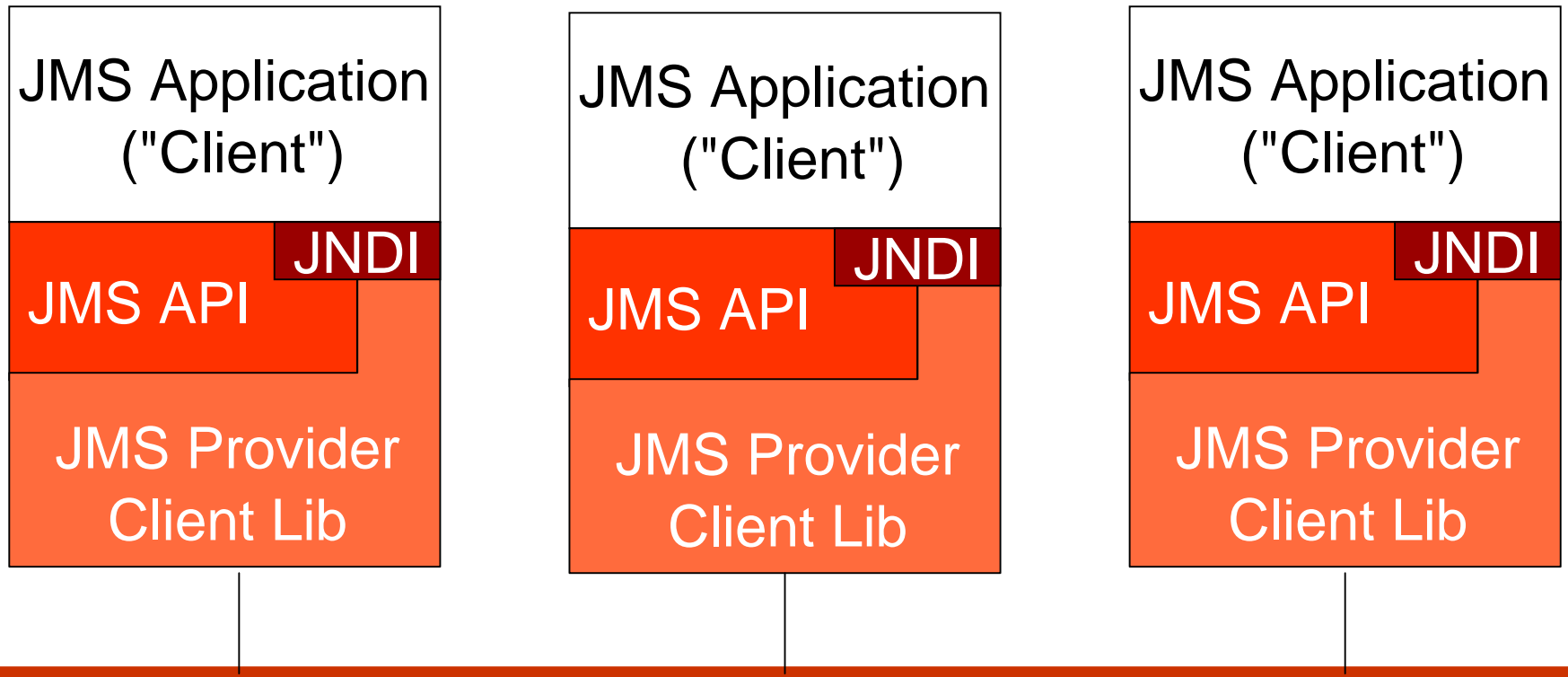
- Messaging is a model in which applications are *loosely coupled* through the exchange of *self-describing messages*.



JMS Components



JMS Components, Serverless



JMS Providers

- Pure JMS Providers
 - Softwired iBus
 - SonicMQ
 - FioranoMQ
 - SwiftMQ
 - BEA Weblogic
 - Sun iPlanet
- Legacy JMS Providers
 - IBM MQ Series
 - Oracle
 - Spiritsoft
 - Talarian (announced)

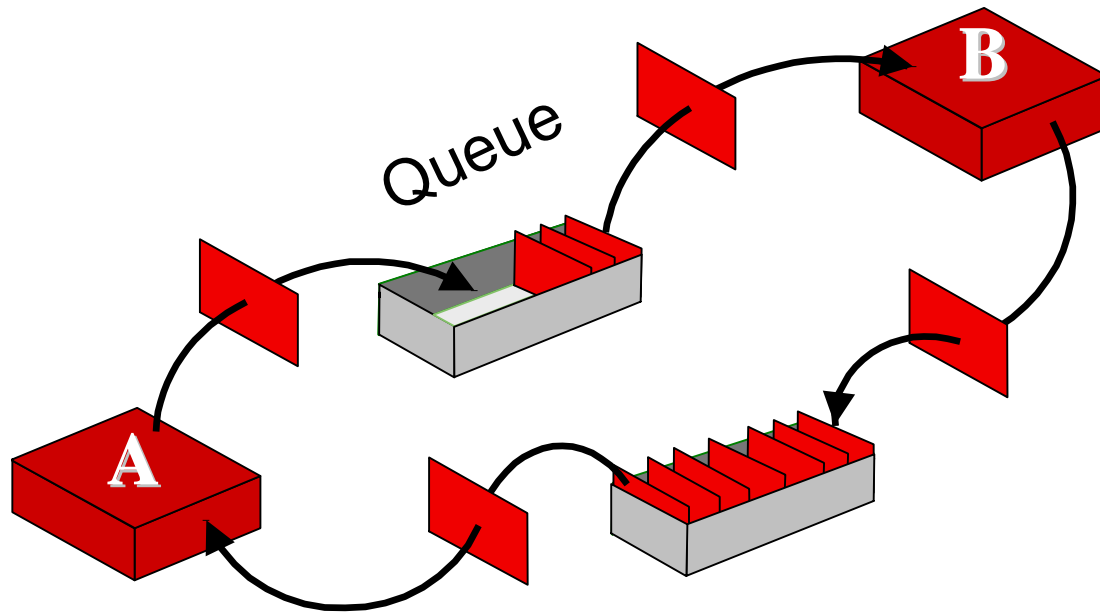
Areas of Use

- Enterprise Application Integration
- Geographic Dispersion:
 - Central office and subsidiaries exchange data
- Business-to-business:
 - EDI
 - XML
- Push-Model-Applications:
 - Online Auctions
 - Stock Quotes
- Mobile Applications

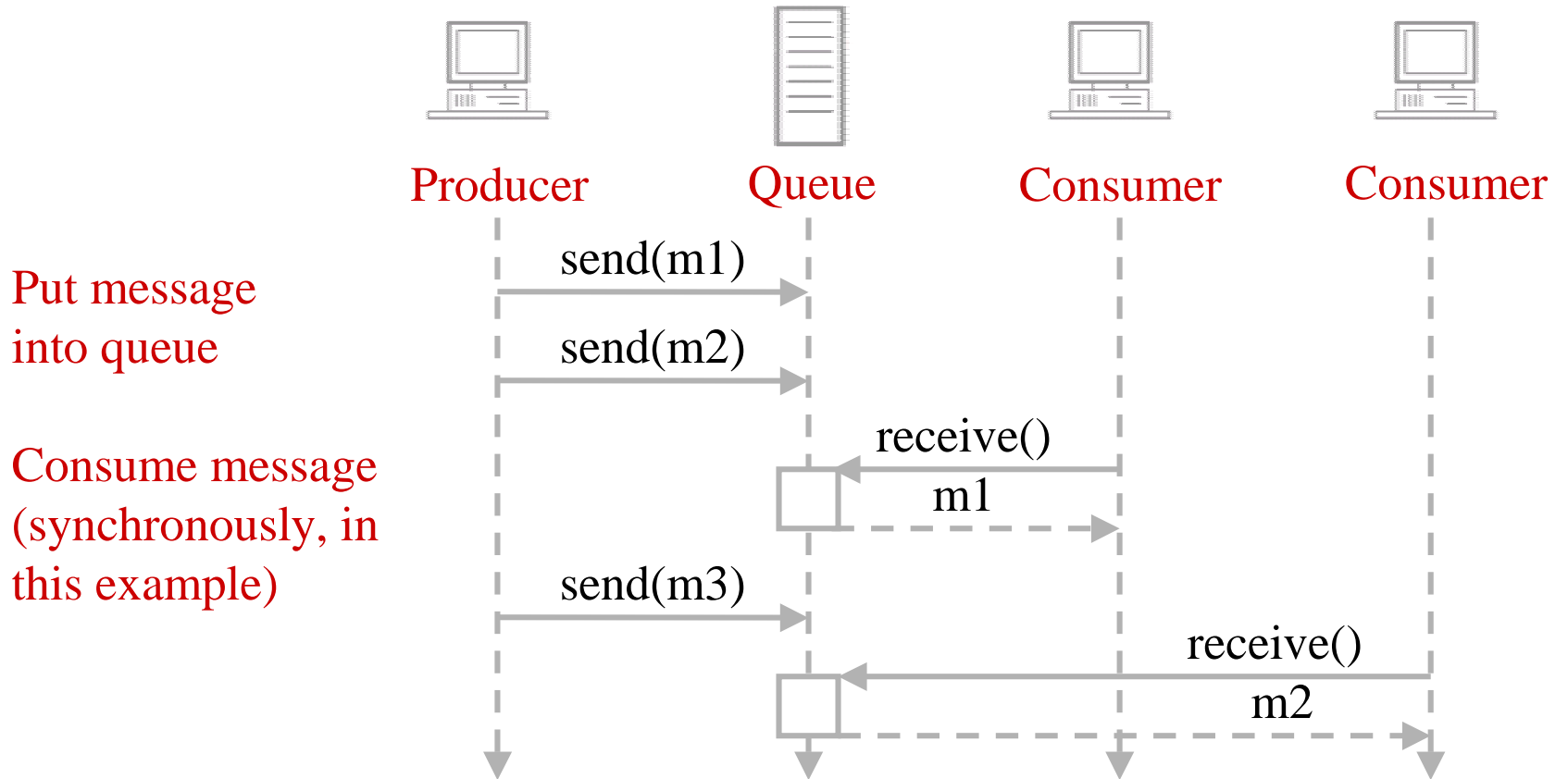
How does JMS work?

- JMS Domains
 - Publish / Subscribe (pub/sub)
 - Point to Point (p2p)
- JMS API
 - Destinations
 - Message Producers / Consumers
 - Message Structure / Message Types

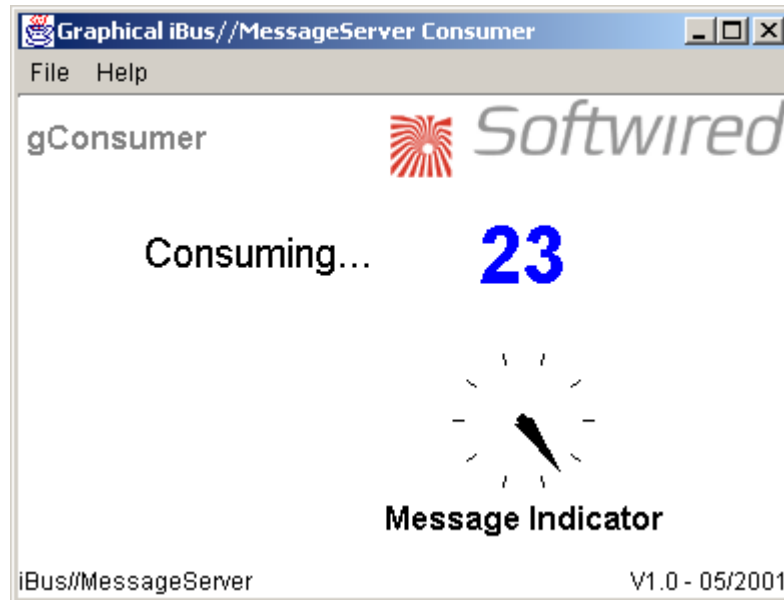
JMS Point to Point



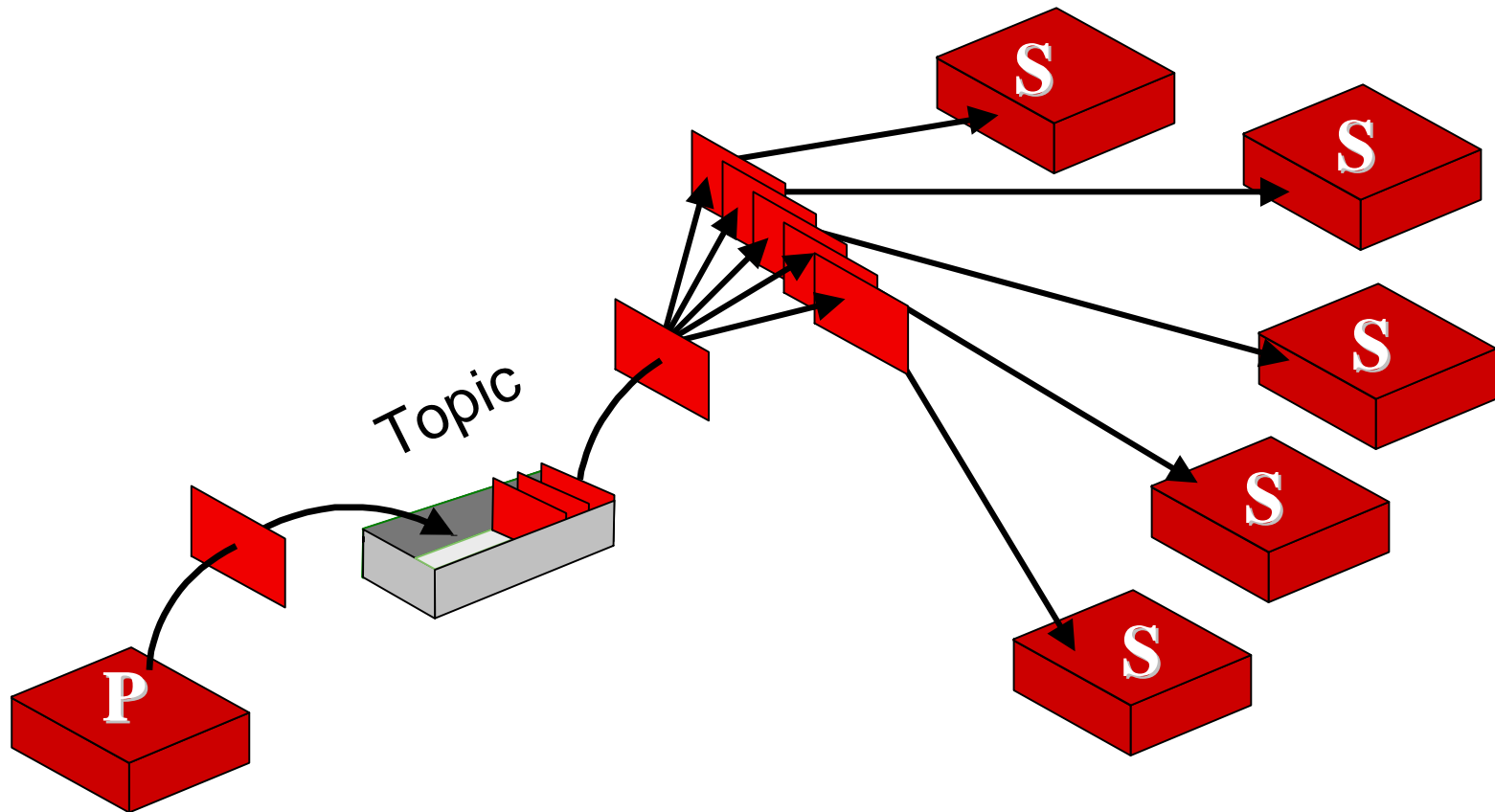
JMS Point to Point (2)



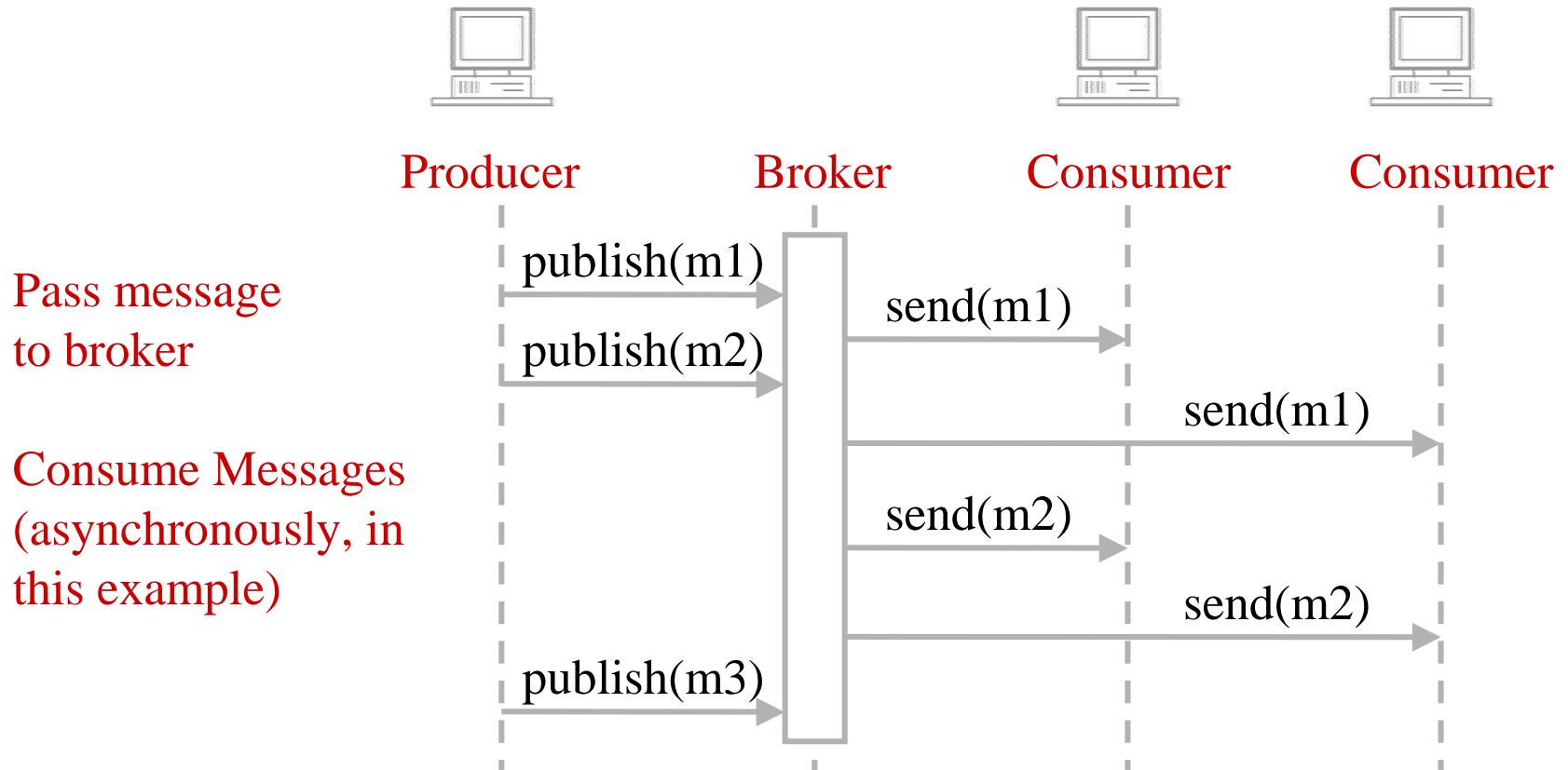
Queue Demo



JMS Publish / Subscribe



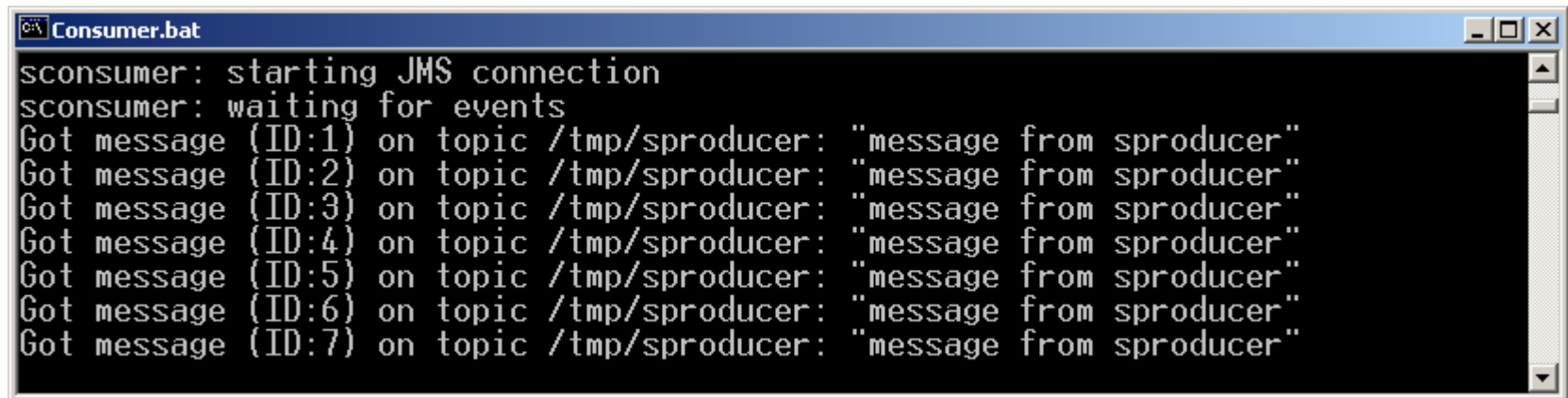
JMS Publish / Subscribe (2)



Pub/Sub: Properties

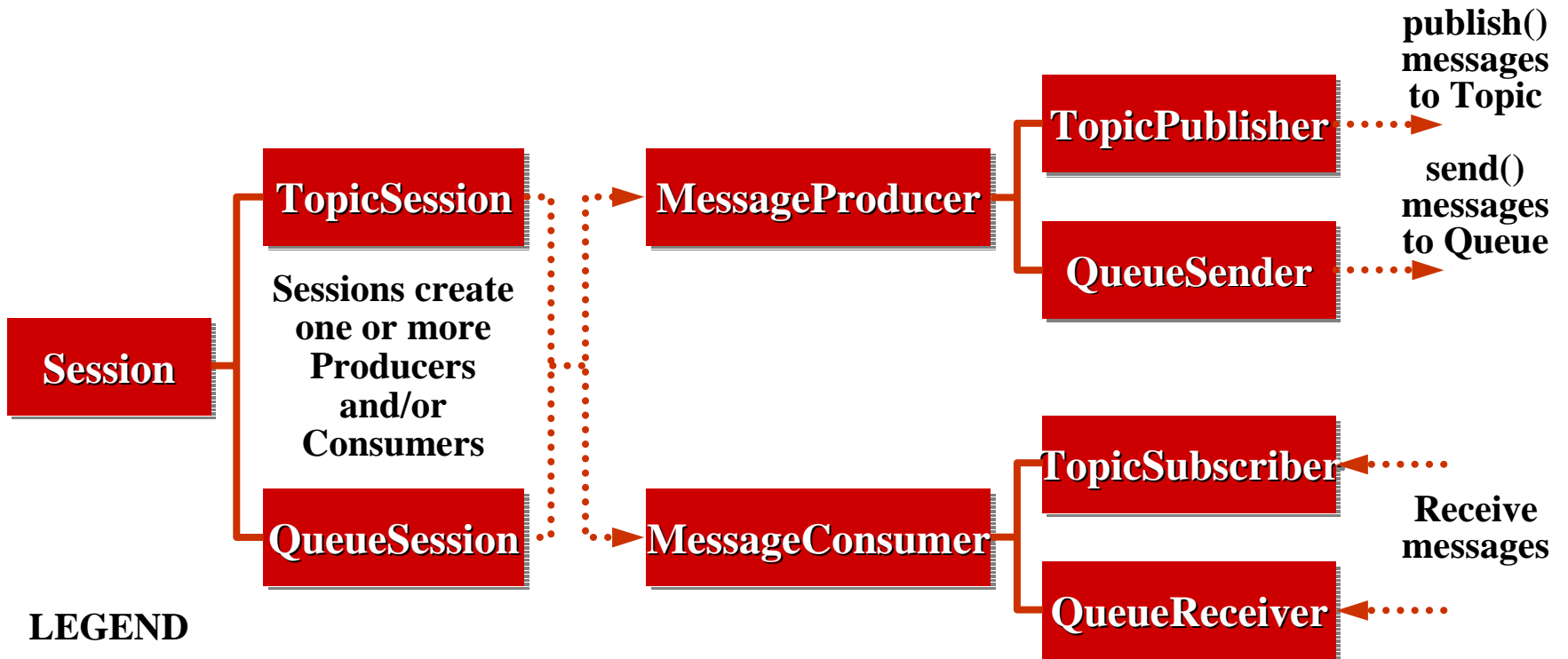
- Push Model: Consumers are *delivered* messages without having to request them.
- *Dynamical addition* of publishers and subscribers at runtime (grow and shrink in complexity over time).
- Each subscriber receives *own copy* of message.

Topic Demo



```
C:\> Consumer.bat
sconsumer: starting JMS connection
sconsumer: waiting for events
Got message (ID:1) on topic /tmp/sproducer: "message from sproducer"
Got message (ID:2) on topic /tmp/sproducer: "message from sproducer"
Got message (ID:3) on topic /tmp/sproducer: "message from sproducer"
Got message (ID:4) on topic /tmp/sproducer: "message from sproducer"
Got message (ID:5) on topic /tmp/sproducer: "message from sproducer"
Got message (ID:6) on topic /tmp/sproducer: "message from sproducer"
Got message (ID:7) on topic /tmp/sproducer: "message from sproducer"
```

JMS API

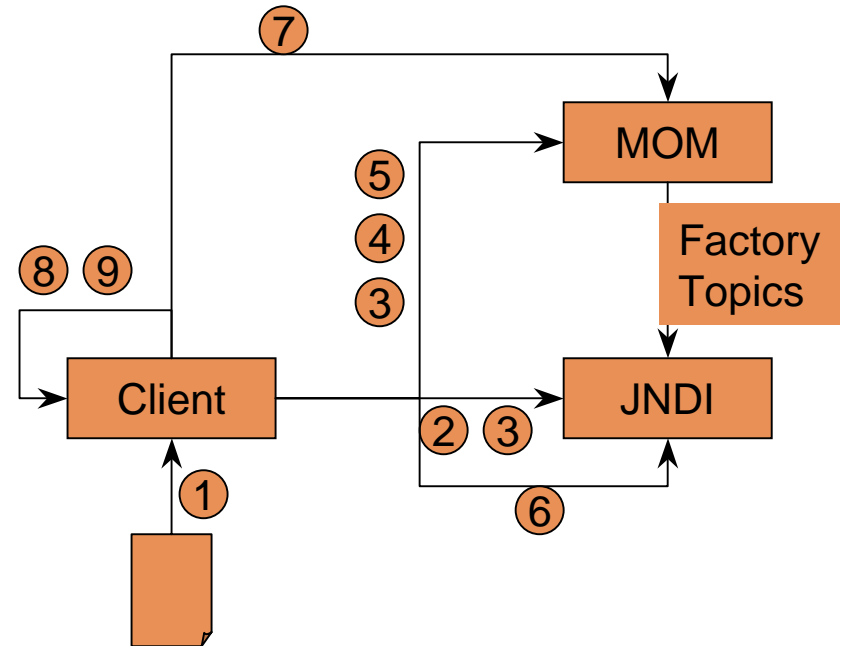


LEGEND

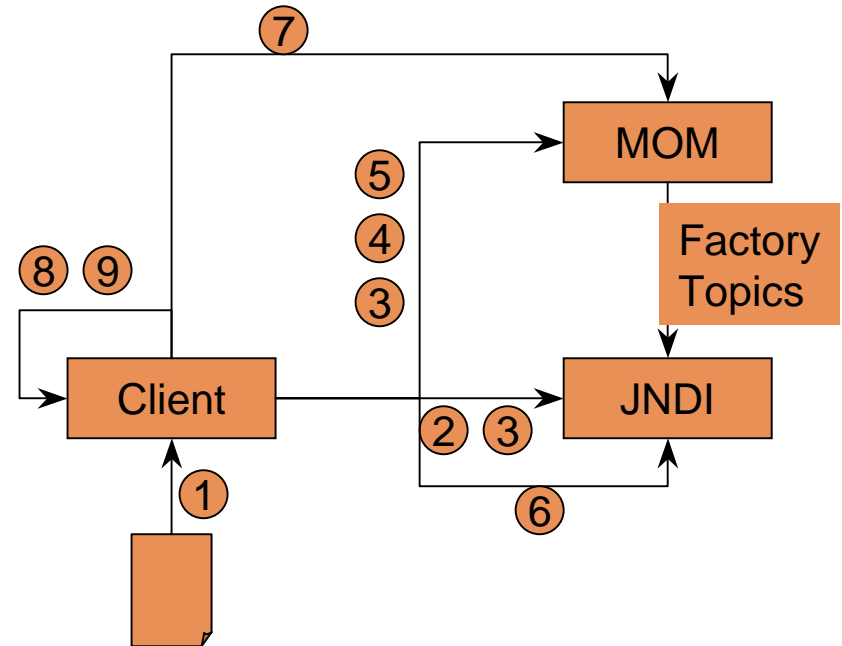
— Inheritance (Extends)

←.....→ Message Flow

Boilerplate of a JMS Application



Boilerplate of a JMS Application

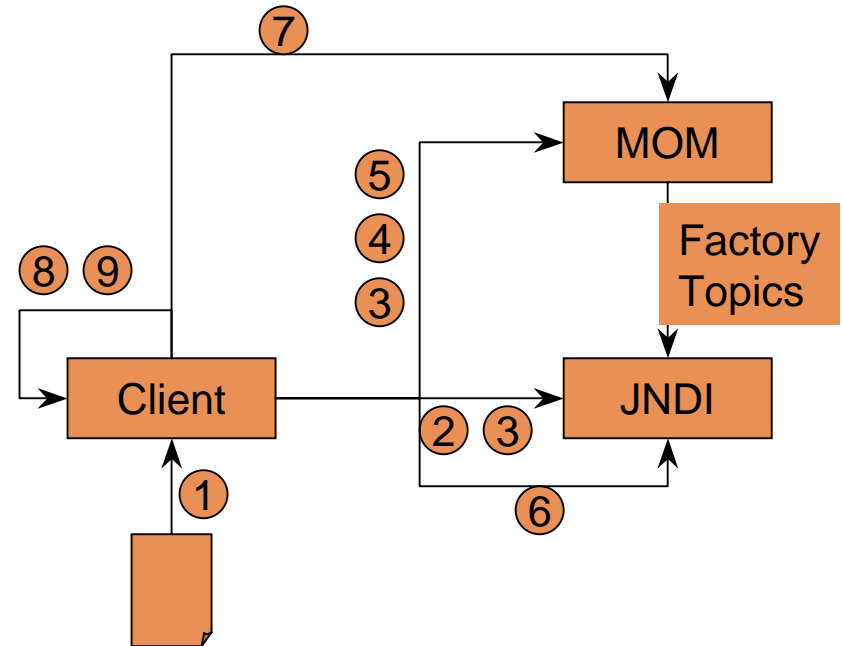


Properties, Context (JNDI)

Goal: Set-up JNDI by setting its root context

```
final String factoryClassName=  
"ch.softwired.jms.naming.IBusContextFactory";  
  
Hashtable env=new Hashtable();  
  
env.put(Context.INITIAL_CONTEXT_FACTORY,  
factoryClassName);  
  
Context context = new InitialContext(env);
```

Boilerplate of a JMS Application



ConnectionFactory, Topic

Goal: Retrieve Administered Objects from JNDI

```
topicConnectionFactory =  
(TopicConnectionFactory)context.lookup("TopicCo  
nnectionFactory");
```

```
topic = (Topic)context.lookup("Quotes");
```

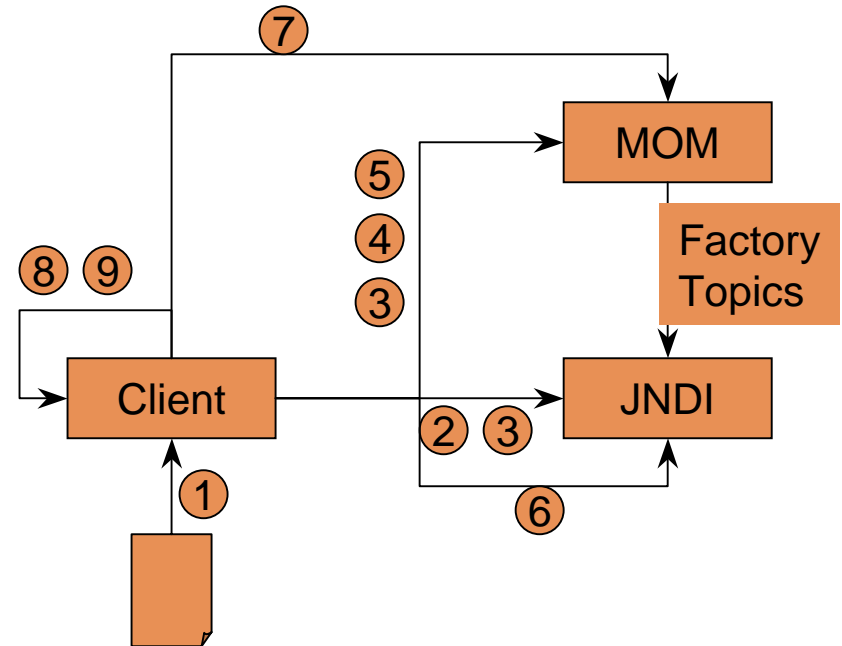
ConnectionFactory

- Sole Purpose: Create JMS Connections
- The ConnectionFactory can be customized before it is bound with JNDI:
 - Server to connect to (IP Address or DNS name)
 - Port to connect to
 - Protocol to use (TCP, SSL, UDP, HTTP)
 - Reconnection-Strategy to use, etc.

Topic

- Created and administrated by System Admin.
- Handle for the physical implementation of topic in JMS server.
- Encapsulates vendor specific name of physical topic.
- Provides a volatile m:n connection of publishers and subscribers.

Boilerplate of a JMS Application



Connection, Session

Goal: Create the Session with the JMS Provider

```
TopicConnection topicConnection =  
topicConnectionFactory.createTopicConnection();
```

```
TopicSession topicSession =  
topicConnection.createTopicSession(false,  
    Session.AUTO_ACKNOWLEDGE);
```

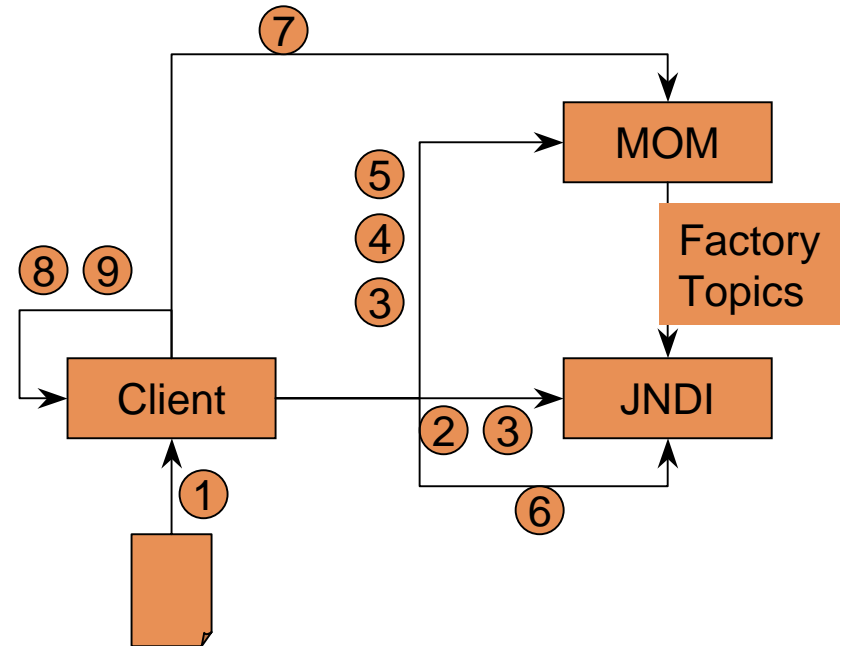
TopicConnection

- Unique, direct connection to the message server
- Factory for TopicSessions
- Expensive!
 - Avoid using more than one connection per client
 - Except if you want to communicate with more than one JMS provider (Bridge application)

TopicSession

- Factory for objects:
 - Messages (including their type).
 - Topic subscribers.
 - Topic publisher.
- Defines transactional behaviour.
- Defines acknowledgement behaviour.
- Assigns messages to topics.
- Not Thread-safe!

Boilerplate of a JMS Application



Publisher, Session Start

Goal: Get Ready to Publish Messages

```
TopicPublisher topicPublisher =  
    topicSession.createPublisher(topic);  
  
topicConnection.start();
```

TopicPublisher

- Helper Object to assist in publishing messages

Create and Publish a Message

Goal: Publish a Message

```
String quoteStr = "SUNW, 19.71";
```

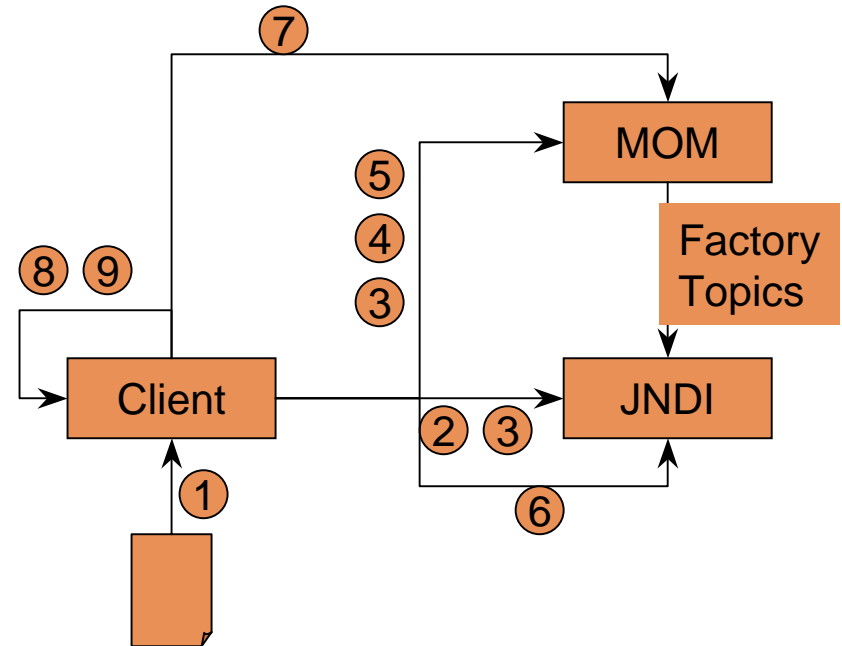
```
TextMessage quote =  
    topicSession.createTextMessage(quoteStr);
```

```
topicPublisher.publish(quote);
```

Demo

Producer.java

Boilerplate of a JMS Application



Receive Messages

Goal: Receive one message synchronously

```
TopicSubscriber topicSubscriber =  
    topicSession.createSubscriber(topic);  
  
Message msg = topicSubscriber.receive();  
String message = ((TextMessage)msg).getText();  
System.out.println("Got message: " + message);
```

TopicSubscriber

- Helper Object to assist in receiving messages
- Asynchronous usage needs a MessageListener
- Synchronous usage via receive()
- Also used to register ExceptionListener, to dispatch asynchronous exceptions

Demo

Receiver.java

Receive Messages

Goal: Set-up Asynchronous Message Receipt

```
TopicSubscriber topicSubscriber =  
    topicSession.createSubscriber(topic);
```

```
topicSubscriber.setMessageListener(new  
    Listener());
```

```
topicConnection.setExceptionHandler(new  
    ErrorListener());
```

MessageListener

Goal: Process Messages

```
class Listener implements MessageListener {
    public void onMessage(Message m) {
        String message = null;
        try {
            message = ((TextMessage)m).getText();
        } catch (JMSEException e) {
            System.out.println("Error");
        }
        System.out.println("Got message: " + message);
    }
}
```

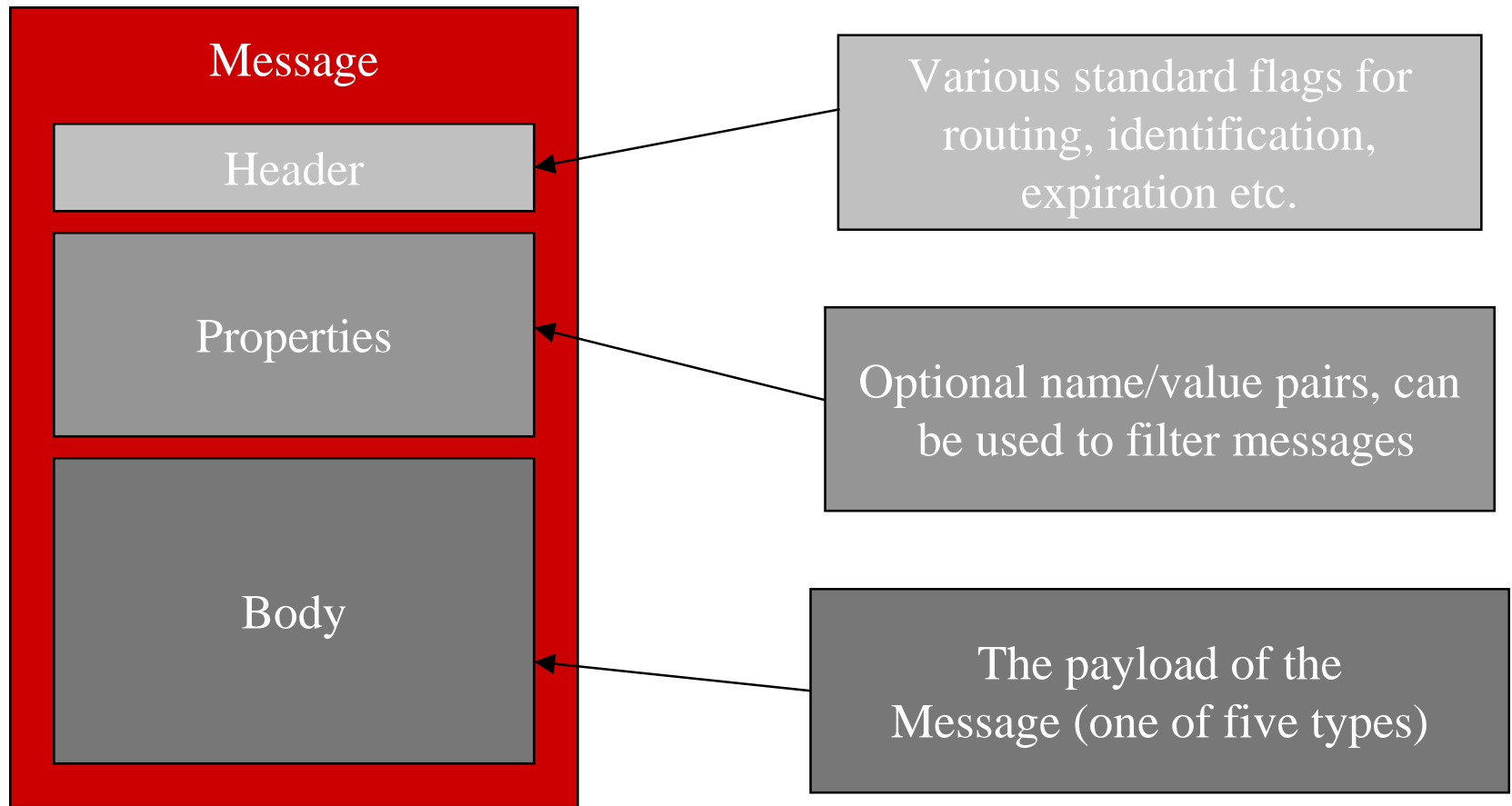
Demo

Consumer.java
ThreadHelper.java

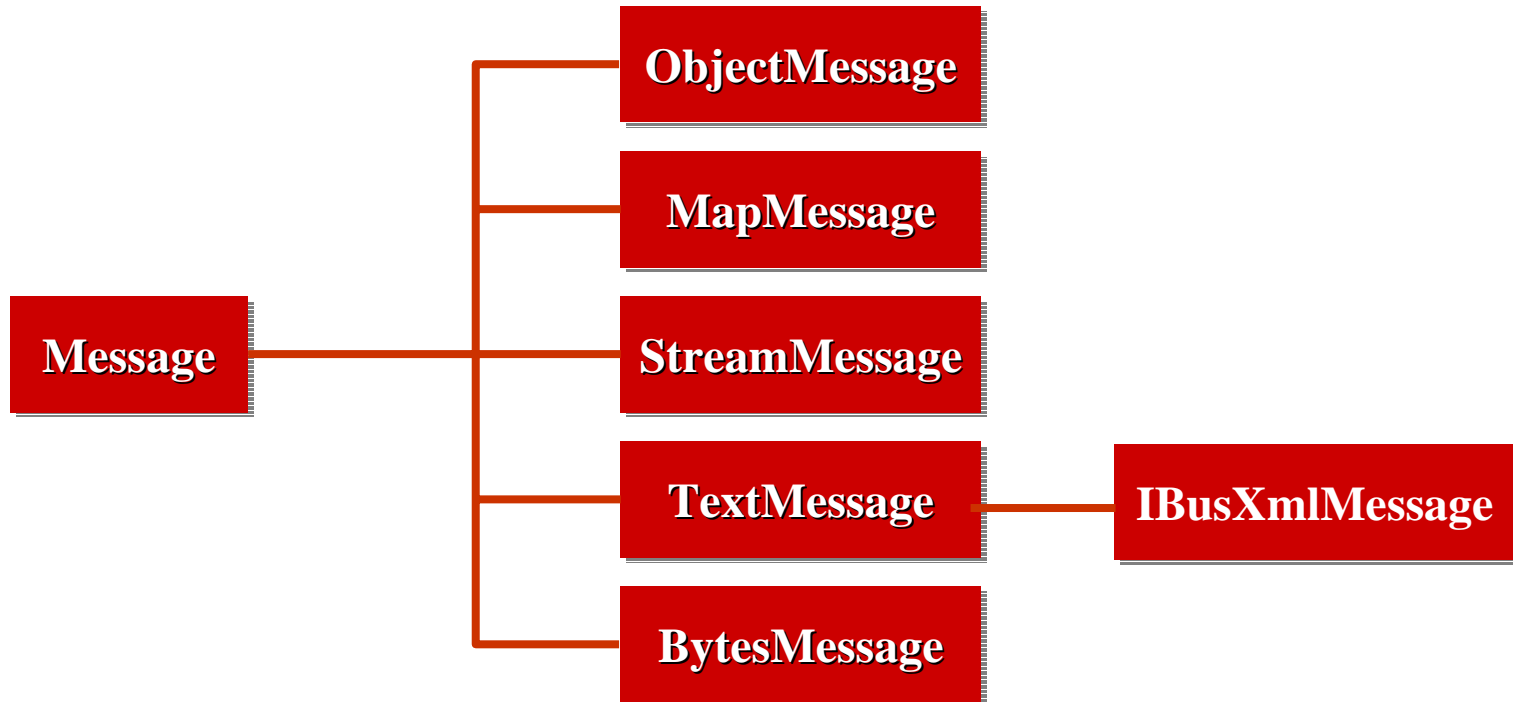
Source Code

- [Producer.java](#)
- [Receiver.java](#)
- [Consumer.java](#)
- [ThreadHelper.java](#)

Structure of JMS Messages



JMS Message Types



LEGEND

— Inheritance (Extends)

Why should I use JMS?

- Properties of a JMS application
- JMS vs. JavaMail
- JMS vs. Distributed Objects

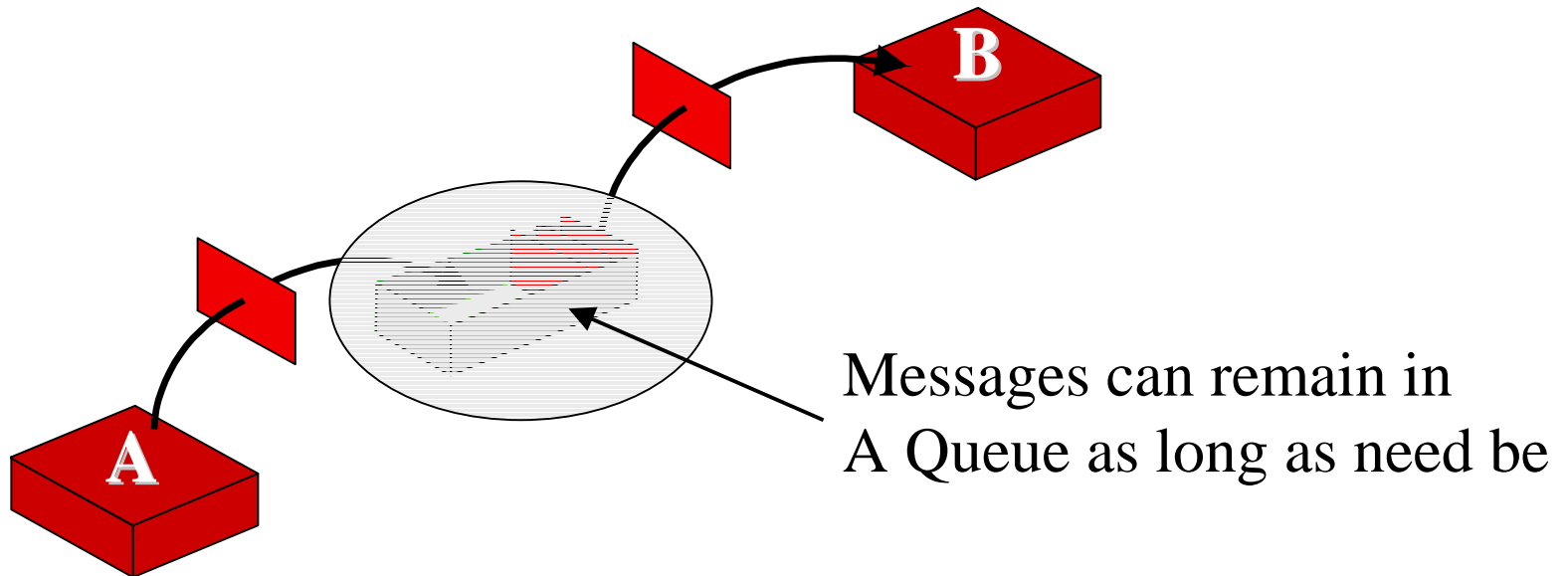
Properties of a JMS application

- Robustness to change*
- Time Independence
- Location Independence
- Latency Hiding (Asynchronous)
- Configurable Quality of Service (QoS)

* Demo

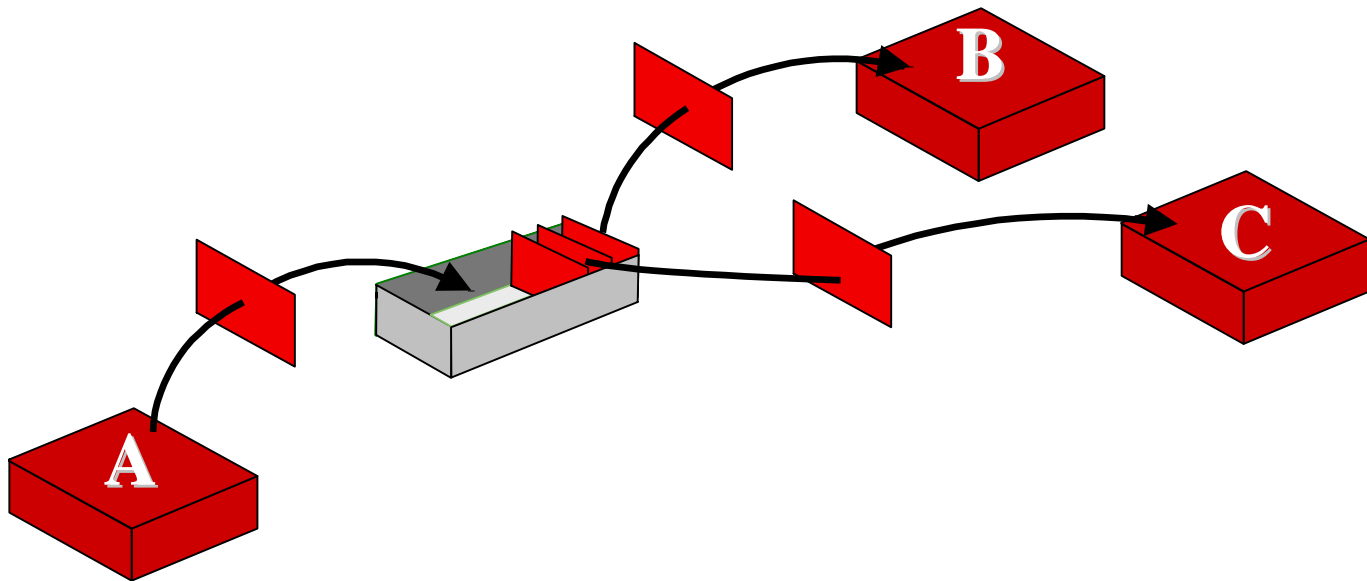
Time Independence

- No need for Producers and Consumers to be available at the same time!



Location Independence

- Messages are addressed to a *Topic* or *Queue*, not to a particular remote object



Configurable Quality of Service

Registered Mail	Bulk Mail
Transactional, persistent, "exactly-once" delivery	Best effort delivery
Financial transactions	Short-lived quote updates
Expensive (Bandwidth, CPU, Disk, Memory)	Resource-optimized (e.g. IP Multicast)

Messaging vs. JavaMail

- What's different from the JavaMail API?
 - Full exception handling
 - Transaction support
 - Delivery order guarantees
 - Efficient binary data support
 - Indirect addressing (Topics and Queues)
 - Consumer load balancing
 - Push all the way through
 - **Much** faster (orders of magnitude)

JMS vs. Distributed Objects

- Distributed Objects
 - RMI
 - CORBA
- Remote Procedure Call
 - OSF/DCE
 - Sun RPC
- Direct Socket Communication

JMS and J2EE

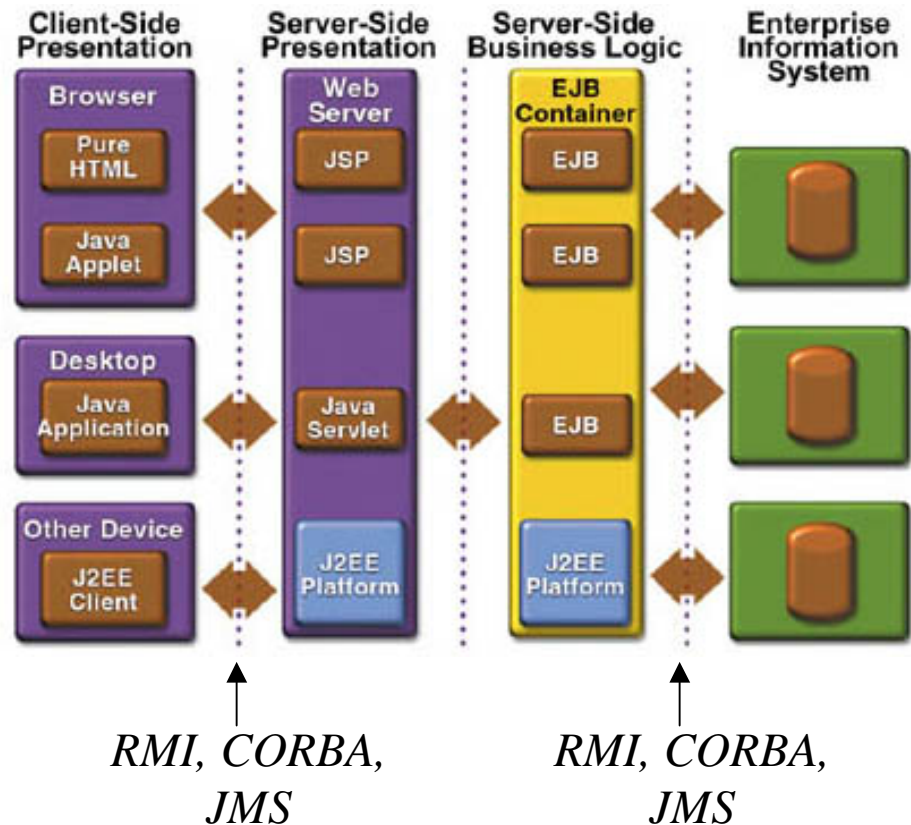
- **Sun:** "JMS is a strategic technology for J2EE. JMS will work in concert with other technologies to provide reliable, asynchronous communication between components in a distributed computing environment."
- New with J2EE version 1.3: Message-Driven Beans (MDB)
- Often, J2EE and JMS provider can be chosen separately, and will work together (example: BEA Weblogic Server / Softwired iBus//MessageServer)

JMS and Enterprise Java Beans

- App server provides EJB, freeing applications from details of threading, transactions, scalability, fault-tolerance.
- JMS plays similar role to RMI: Link clients with EJBs.
- New Message Driven Beans to receive messages (EJB 2.0)
- App server transactions replace/augment JMS transactions.

JMS and J2EE, EJB

- J2EE Family
- Enterprise JavaBeans
- JavaServer Pages
- Servlets
- Java Naming and Directory Interface (JNDI)
- Java Transaction API (JTA)
- CORBA
- JDBC data access
- ... and JMS!



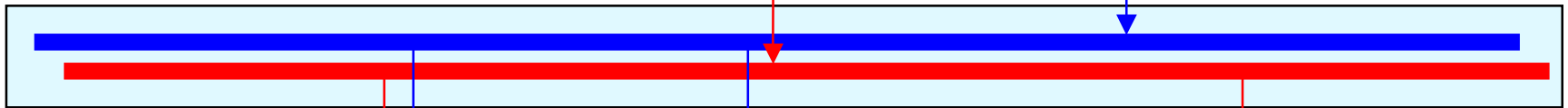
How does a simple JMS application look like?

- A pub/sub example
- A p2p example
- A mobile messaging / mixed example

A pub/sub example

publish ("SUNW", 18.67);
publish ("IBM", 115.27);

publish ("CSGN", 327);
publish ("SRN", 127);



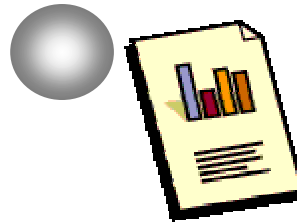
subscribe ("NYSE");
subscribe ("SWX");

subscribe ("SWX");

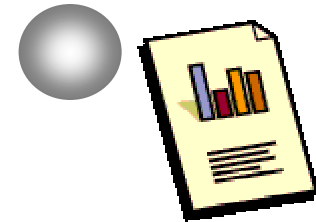
subscribe ("NYSE");



Risk Management

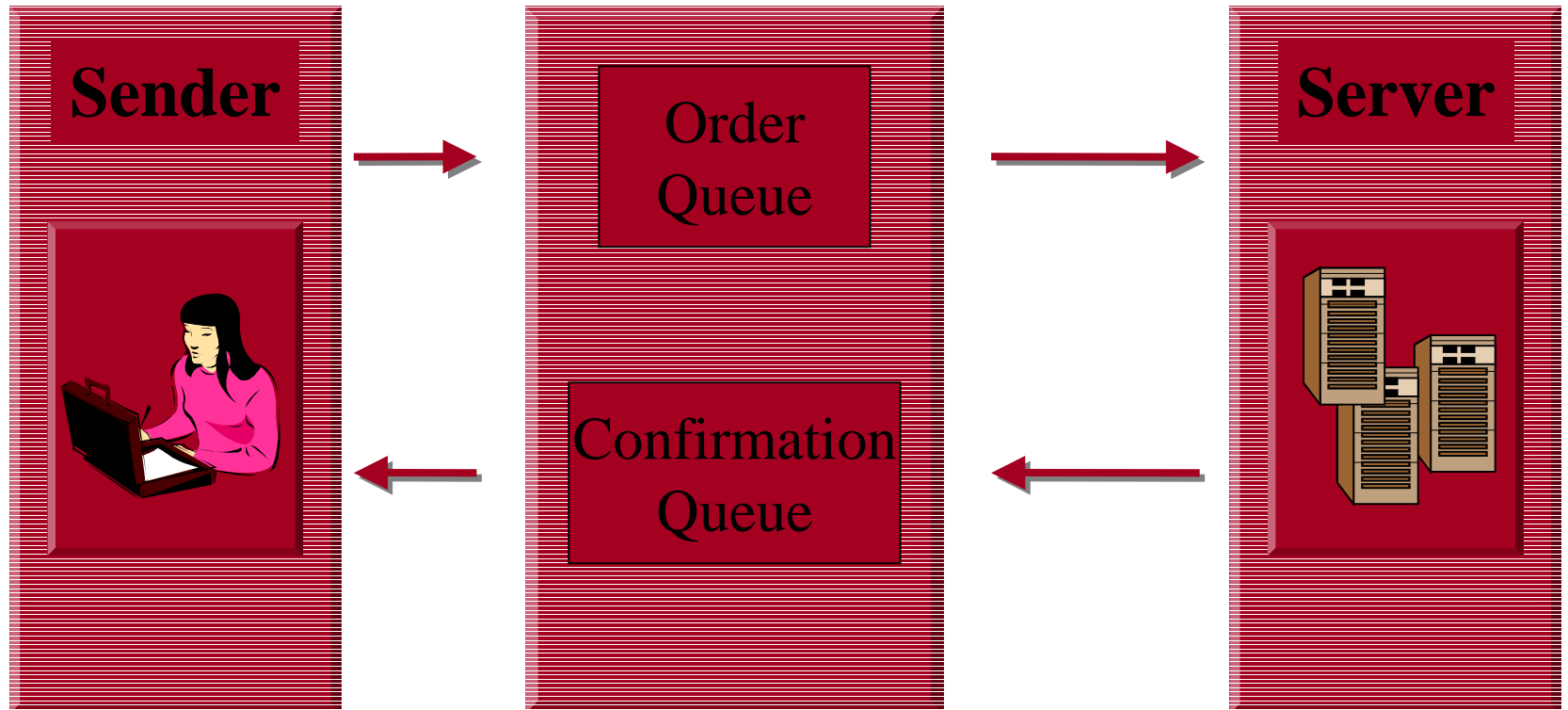


Trade SWX

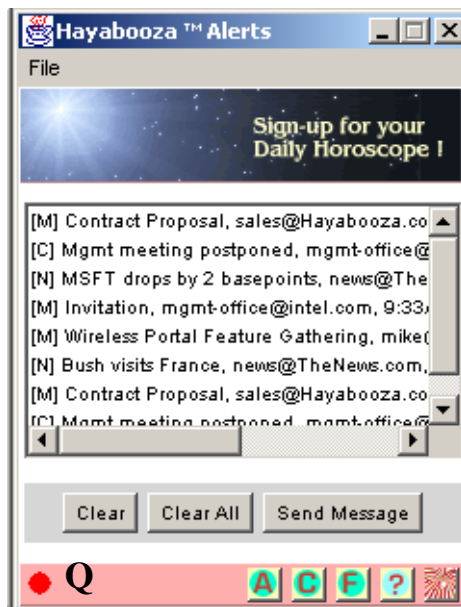


Trade NYSE

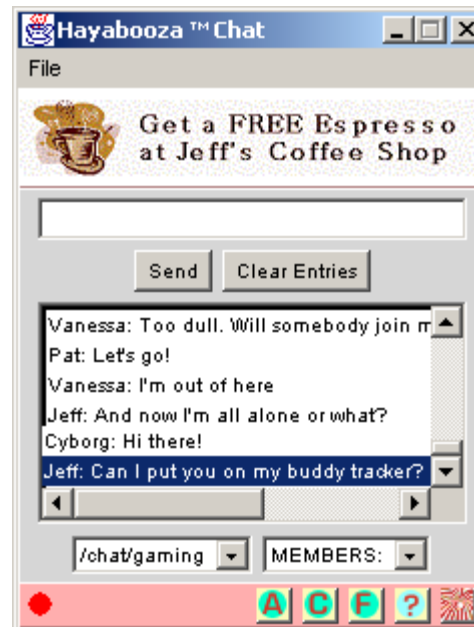
A p2p example



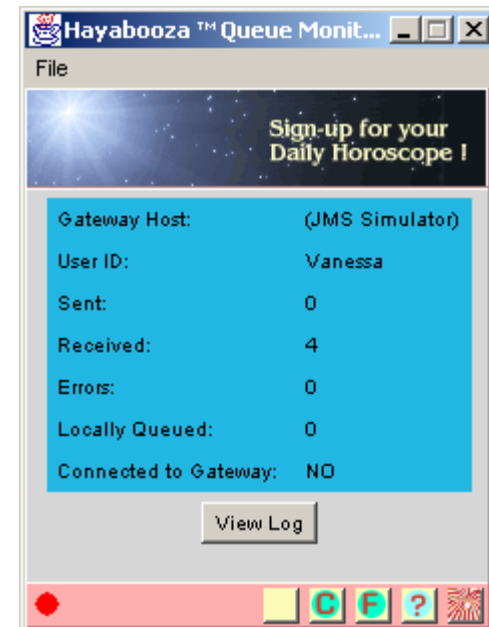
A mobile messaging / mixed example



Alert Notifications
(e-mail, calendar, news)



Wireless Chat



Control Panel

JMS-Product Interoperability

- JMS is an *Interface-Level* specification (i.e. there is no wire protocol specified).
- Compile-time differences:
 - Naming of destinations (to solve: Use JNDI).
 - Naming of initial context (to solve: Use config-file).
 - Different message header and properties contents
- Run-time differences:
 - Connection Scalability
 - Throughput, Latency
 - Deployment architectures (local cluster vs. server-to-server routing)
 - Protocol support
 - Resource consumption etc.

Conclusions

- Messaging is a very compelling alternative to RMI, Corba etc.
- When developing in Java, JMS is the API to use for accessing a messaging service.
- Don't confuse JavaMail and JMS!
- For mobile application data exchange, JMS is almost the only "working" approach.

Thank You!

More information:

Softwired AG
Martin Erzberger
Technoparkstrasse 1
8005 Zurich
Switzerland

martin.erzberger@softwired-inc.com

011 41 445 2370

Advanced JMS Features

- Pub/Sub
- Point to Point
- Message Selectors
- Transactions

Temporary Topics

- Created by the user.
- Living as long as session lives.

Publisher:

- Call `createTemporaryTopic()` on the `TopicSession`.
- Then attach topic to message: `setJMSReplyTo()`.

Subscriber:

- Get topic out of message: `getJMSReplyTo()`.
- Create publisher with that topic using session.

Topic Session: Acknowledgements

- Has to be specified on Consumer side.
- **AUTO_ACKNOWLEDGE (at-most-once-delivery - Default):**
 - Acknowledgement is last operation after message handling.
- **DUPS_OK_ACKNOWLEDGE (at-least-once-delivery):**
 - Provider is allowed to send message more than once
 - Client needs to handle duplicates.
 - Might be faster than once-and-only-once
- **CLIENT_ACKNOWLEDGE (exactly-once-delivery):**
 - JMS client explicitly acknowledges using `message.acknowledge()` in `onMessage()` method.
 - Finer grained control of acknowledging

Durable Subscriptions

- Outlasts client's connection with message server.
- Store-and-forward Messaging:
 - JMS server stores messages on client's behalf.
 - After subscriber becomes available again, the JMS server forwards all un-expired messages.
- Created by using TopicSession's `createDurableSubscriber()` method.
- Explicitly call `TopicSession.unsubscribe()` to really disconnect client – subscriber must be closed first.

QueueBrowser

- Allows peeking messages in Queue without consuming them.
- Created by using QueueSessions `createBrowser()`-method.
- Browsed by using QueueBrowser's `getEnumeration()`-method.
- The view of a queue browser is not stable:
 - Sequence may change due to priorities.
 - Messages may disappear because of expiry or consumption by another consumer.

Temporary Queue

- Created by user.
- Can only be consumed by JMS client that created it.
- Created by calling `createTemporaryQueue()` on the `QueueSession`'s.
- Similar usage as `TemporaryTopic` in `Publish/Subscribe`.

Message Selectors

- Filter messages on receipt.
- Use properties and criteria in conditional expressions (SQL-92, WHERE-clauses) to decide with boolean logic which messages will be sent to client.
- Created by using sessions createSubscriber()-method:
 - `...Message.setStringProperty("UserName"), userName);...`
 - `createSubscriber(topic, "UserName <> 'Tom'", false);`
- P2P: Visible in queue for all but consumers with this selector.

Message Selectors: Syntax

- Identifiers are property or header names.
- Comparison Operators:
 - Result is either TRUE or FALSE.
 - <, >, <=, >=, <> (not equal), = .
 - AND, OR, NOT.
 - IS NULL, IS NOT NULL.
 - LIKE (similarity – „name LIKE ,%ohn““).
 - „_“ is wildcard for 1 char.
 - „%“ is wildcard for any sequence.
 - BETWEEN (range – „age BETWEEN 20 AND 30““).

JMS Transactions

- Local vs. Distributed Transactions
- The optional XASession etc.

Local vs. Distributed Transaction

- Local Transaction:
 - Used to group multiple message send operations, receive acknowledgements, or both
 - Scope: One JMS Session (thus one Thread)
- Distributed Transaction:
 - Used to group messaging operations with other transactional operations (e.g. DB update)
 - Scope: Defined via Transaction Context
 - No-no: Transactional send and ack receipt of one message!

The optional XASession

- JMS Specification Chapter 8.
- Used mainly by App Server Vendors to integrate JMS with MDBs.
- However, the feature can be used in a standalone JMS application.
- Replaces the "simple" JMS transactions.