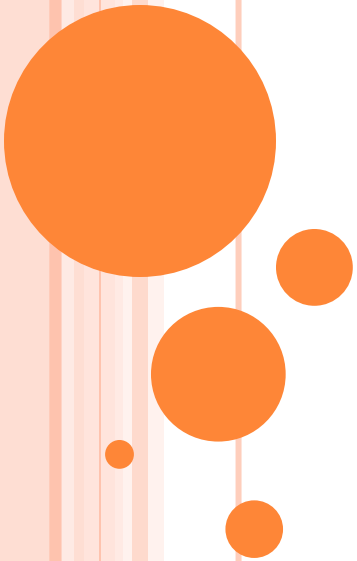


Network Programming: Message-Oriented Middleware

COMP 30220: Distributed Systems

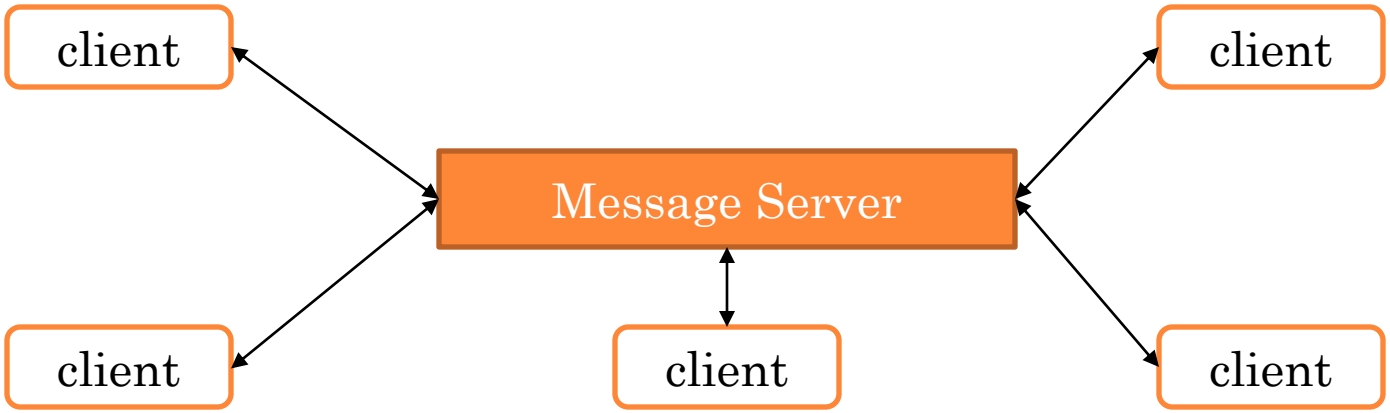
Lecturer: Rem Collier

Email: rem.collier@ucd.ie

Message-Oriented Middleware

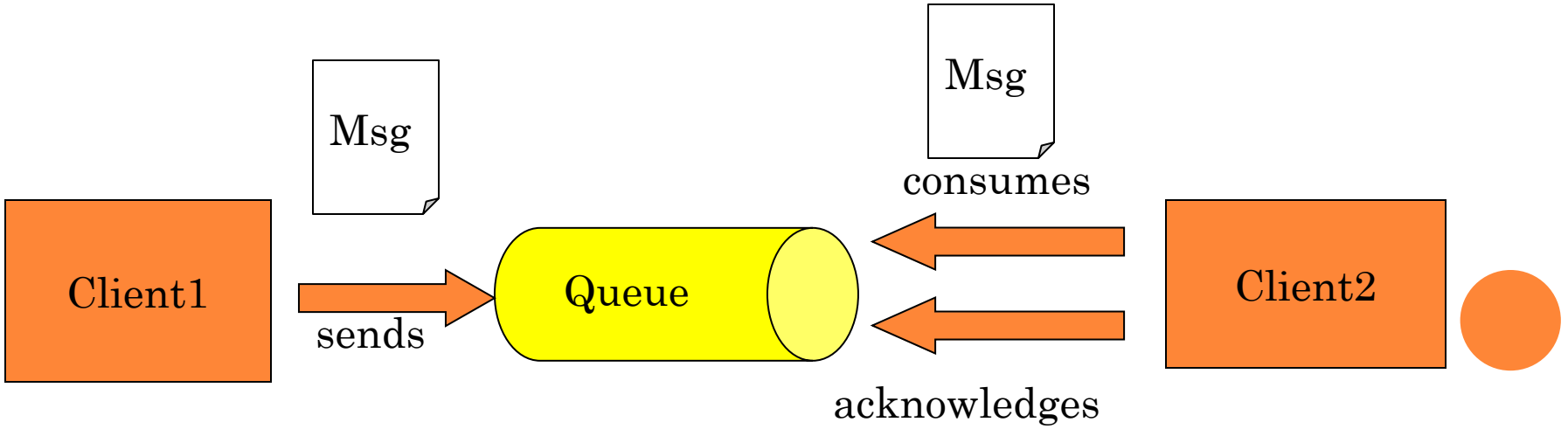
- Message-Oriented Middleware (MOM):
  - *a software or hardware infrastructure supporting sending and receiving messages between distributed systems.*

- Key components:
  - **MOM server(s)**: also known as **message brokers**.
  - **MOM clients**: connect to the server in order to send and receive messages.



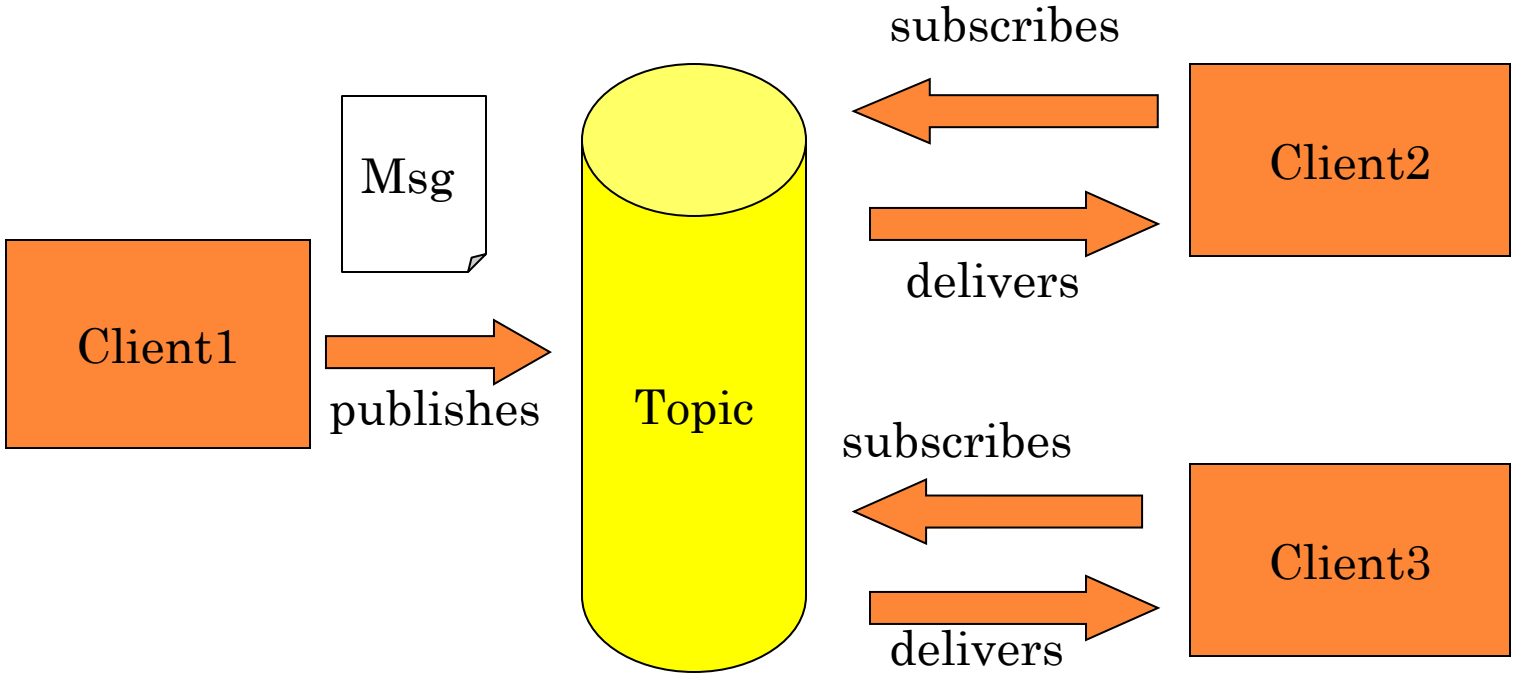
Messaging Models

- Point-to-Point (PTP):
  - Based on FIFO message queues.
  - Multiple clients (**senders**) can post messages on the queue.
  - Sent messages are stored in a persistent message queue until the receiver reads them.
  - Multiple **receivers** can share a queue but each message is delivered to only one receiver.
  - The receiver must acknowledge the receipt of the message.



Messaging Models

- Publish-Subscribe (pub/sub):
  - Topical message queue.
  - Fully decouples sender (publisher) and receiver (subscriber).
  - Delivered to ALL active subscribers.

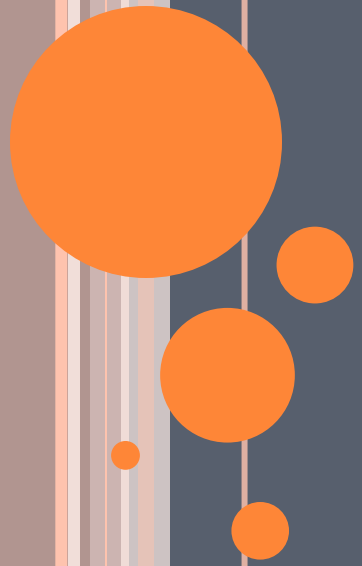


Implications of MOM

- Load Balancing:
  - In P2P mode, clients submit messages to a queue on the message broker.
  - The message broker delivers each message to exactly one receiver.
  - Multiple clients can be assigned to a queue allowing messages to be delivered in parallel.
  - The client does not know which receiver handles the message
  - The client does not wait for the receiver to respond.
- Parallelisation:
  - In Pub/Sub mode, a publisher can submit a message to many subscribers.
  - All subscribers process **the same message** in parallel!
  - If necessary, a queue can be used send responses back to the publisher

Benefits of MOM

- Loose-coupling:
  - Asynchronous messaging means the client no longer waits for a response.
- Reliability:
  - Message / network failure is handled by message persistence.
- Scalability:
  - Support for queuing of messages / introduction of extra components to handle high-demand.
- Availability:
  - Resulting from improved reliability / scalability.
- Language Independent:
  - MOM Server can work with clients written in different languages as long as the message format is language independent.



Java Messaging Services (JMS)

Java Messaging Service (JMS)

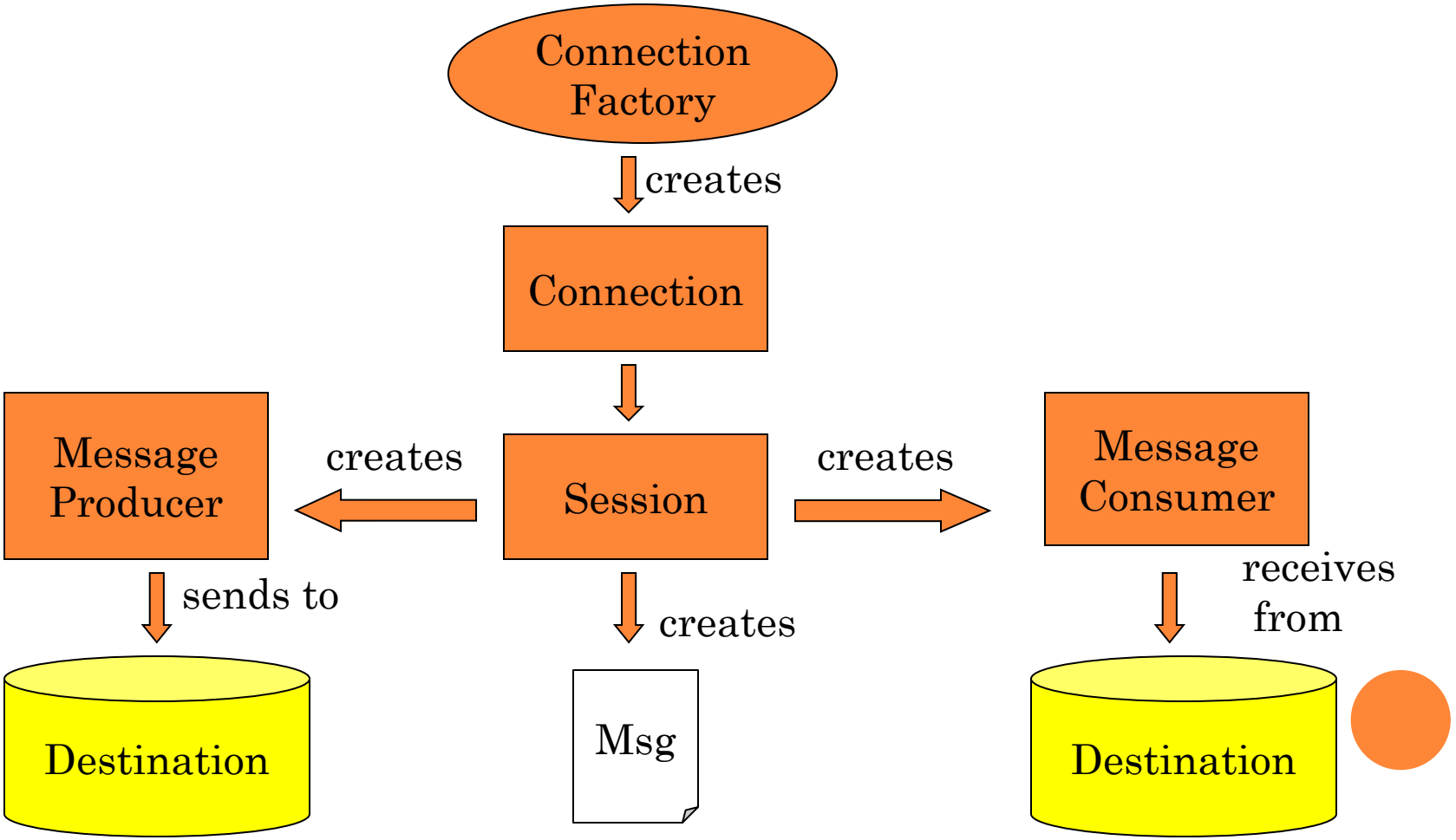
- JMS is an Application Programming Interface (API) for Java-based MOM.
  - Acts as an enabling layer between the client and the provider.
  - Standardises how to interact with the provider.
  - Clear separation of concerns – provider can be changed with minimal impact on the client.
- Many JMS Providers exist:
  - Apache ActiveMQ, Apache Qpid, Weblogic, OpenJMS, …
- We will use ActiveMQ for this class:
  - http://activemq.apache.org/

Setup

- Download ActiveMQ from the website
  - Look in the "bin" folder for "activemq.bat" or equivalent to start the server.
  - Connect your web browser to: http://localhost:8161/admin
  - Username: admin / Password: admin

- Create an Eclipse project:
  - Download **javax.jms.jar** and **jms.jar** from Moodle.
  - copy the **activemq-all-XXX.jar** file from the ActiveMQ root folder into the project and add it to the build path.
  - Lets code some JMS clients!

JMS API Programming Model



A "Simple" JMS Sender Client

```
public class Sender {
    public static void main(String[] args) {
        ConnectionFactory connectionFactory =
            new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT_BROKER_URL);
        try {
            Connection connection = connectionFactory.createConnection();
            connection.setClientID("sender");
            Session session = connection.createSession(false,
                Session.AUTO_ACKNOWLEDGE);
            Queue queue = session.createQueue("TESTQUEUE");
            MessageProducer messageProducer = session.createProducer(queue);

            TextMessage textMessage = session.createTextMessage("Hello World");
            messageProducer.send(textMessage);
        } catch (JMSException e) { e.printStackTrace(); }
    }
}
```

Deconstructing the Sender

- Creating the connection factory:

```
ConnectionFactory connectionFactory =
    new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT_BROKER_URL);
```

- Getting a connection (and setting the sender name):

```
Connection connection = connectionFactory.createConnection();
connection.setClientID(<client-name>);
```

- Creating a Session:

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

- Creating / Getting a Queue:

```
Queue queue = session.createQueue(<queue-name>);
```

- Creating a Producer:

```
MessageProducer messageProducer = session.createProducer(queue);
```

- Sending a Message:

```
TextMessage textMessage = session.createTextMessage("Hello World");
messageProducer.send(textMessage);
```

A "Simple" JMS Receiver Client

```
public class Receiver {
    public static void main(String[] args) {
        try {
            ConnectionFactory factory = (ConnectionFactory)
                new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT_BROKER_URL);
            Connection connection = factory.createConnection();
            connection.setClientID("receiver");
            Session session = connection.createSession(false,
                Session.CLIENT_ACKNOWLEDGE);
            Queue queue = session.createQueue("TESTQUEUE");
            MessageConsumer consumer = session.createConsumer(queue);
            connection.start();
            Message message = consumer.receive();
            if (message instanceof TextMessage) {
                System.out.println("Received: " + ((TextMessage) message).getText());
                message.acknowledge();
            }
            connection.close();
        } catch (JMSException e) { e.printStackTrace(); }
    }
}
```

Deconstructing the Receiver

- The code is the same – up to creating the message consumer…
- Creating the Consumer:

```
MessageConsumer consumer = session.createConsumer(queue);
```

- Start the connection (for listening):

```
connection.start();
```

- Receive the message:

```
Message message = consumer.receive();
```

- Acknowledge the receipt of the message:

```
message.acknowledge();
```

- Close the connection:

```
connection.close();
```

Switching to Pub/Sub

- To change to pub/sub only 2 changes are necessary:
  - No client id is required (so we do not have to generate unique client id's for each producer/consumer).
  - Change from createQueue to createTopic:

```
Queue queue = session.createQueue(<queue-name>);
```

  - Becomes:

```
Destination destination = session.createTopic("MYTOPIC");
```

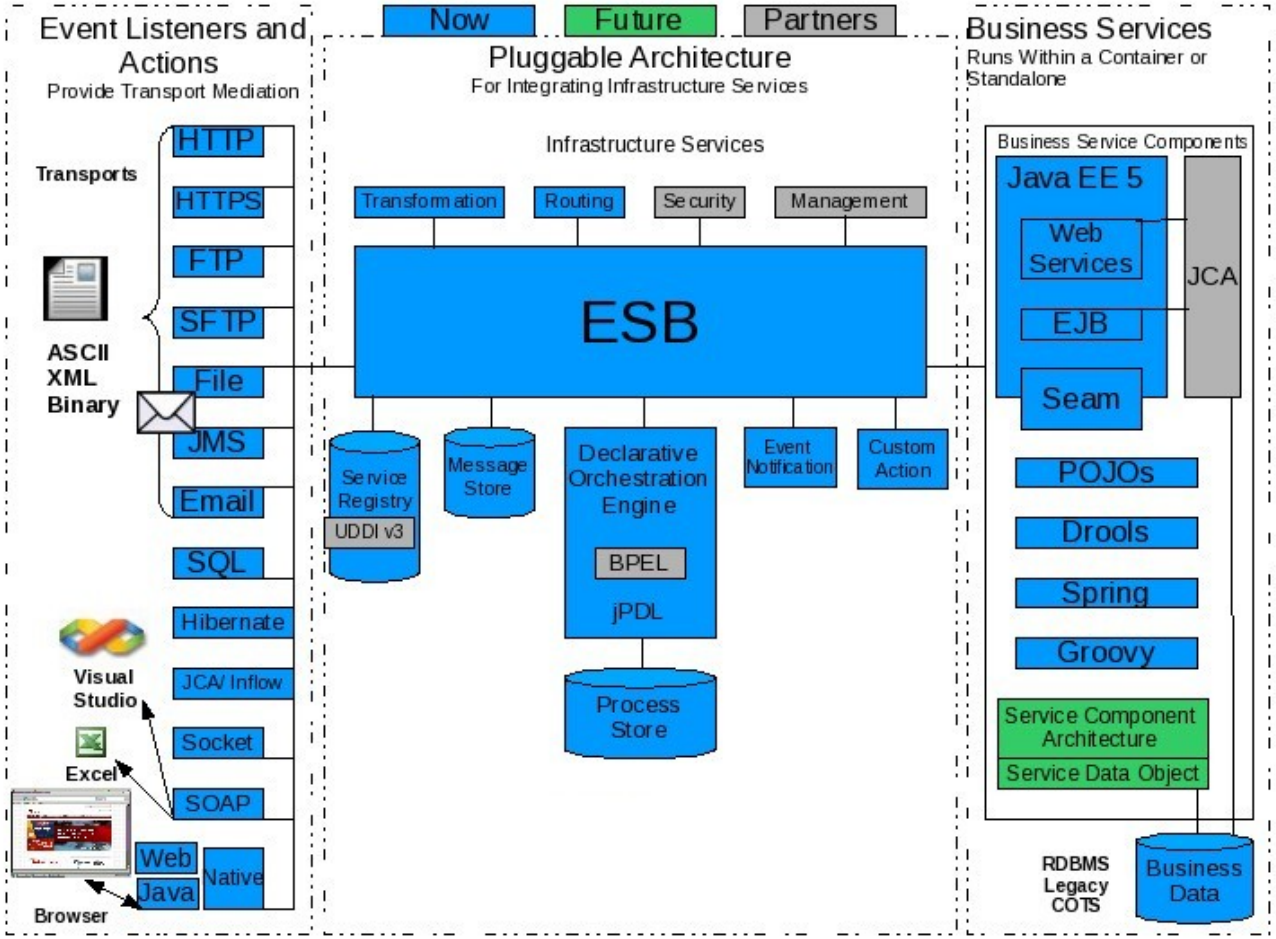
JMS Message Types

Message Type	Contains	Some Methods
TextMessage	String	getText,setText
MapMessage	set of name/value pairs	setString,setDouble,setLong,getDouble,getString
BytesMessage	stream of uninterpreted bytes	writeBytes,readBytes
StreamMessage	stream of primitive values	writeString,writeDouble,writeLong,readString
ObjectMessage	serialize object	setObject,getObject

Discussion

- Okay:
  - Deployments require an additional component (the MOM Server)
  - The code is more complex than the RMI code (but similar complexity to the socket code)
- But:
  - Communication between components is now asynchronous and reliable.
  - One solution can support both unicast (PTP) and broadcast (pub/sub)
  - It is not Java only (unlike RMI)
  - MOM solutions offer out of the box support for scaling the solution.
  - It is an accepted industry standard!

Final Thought: Enterprise Service Bus



Final Thought: Enterprise Service Bus

- MOM is the basis for many ESB architectures
  - Message broker also provides "intelligence":
    - Fault-tolerance
    - Transactions
    - Intelligent Routing
  - "Dump Endpoints & Smart Middleware"
- But, MOM is not ESB!
  - Complexity of ESB architectures cause a development bottleneck:
    - Every service must interact with it
    - Design driven by middleware not project
    - Key logic moved from service to middleware
  - "Smart Endpoints & Dumb Middleware" (Microservices)