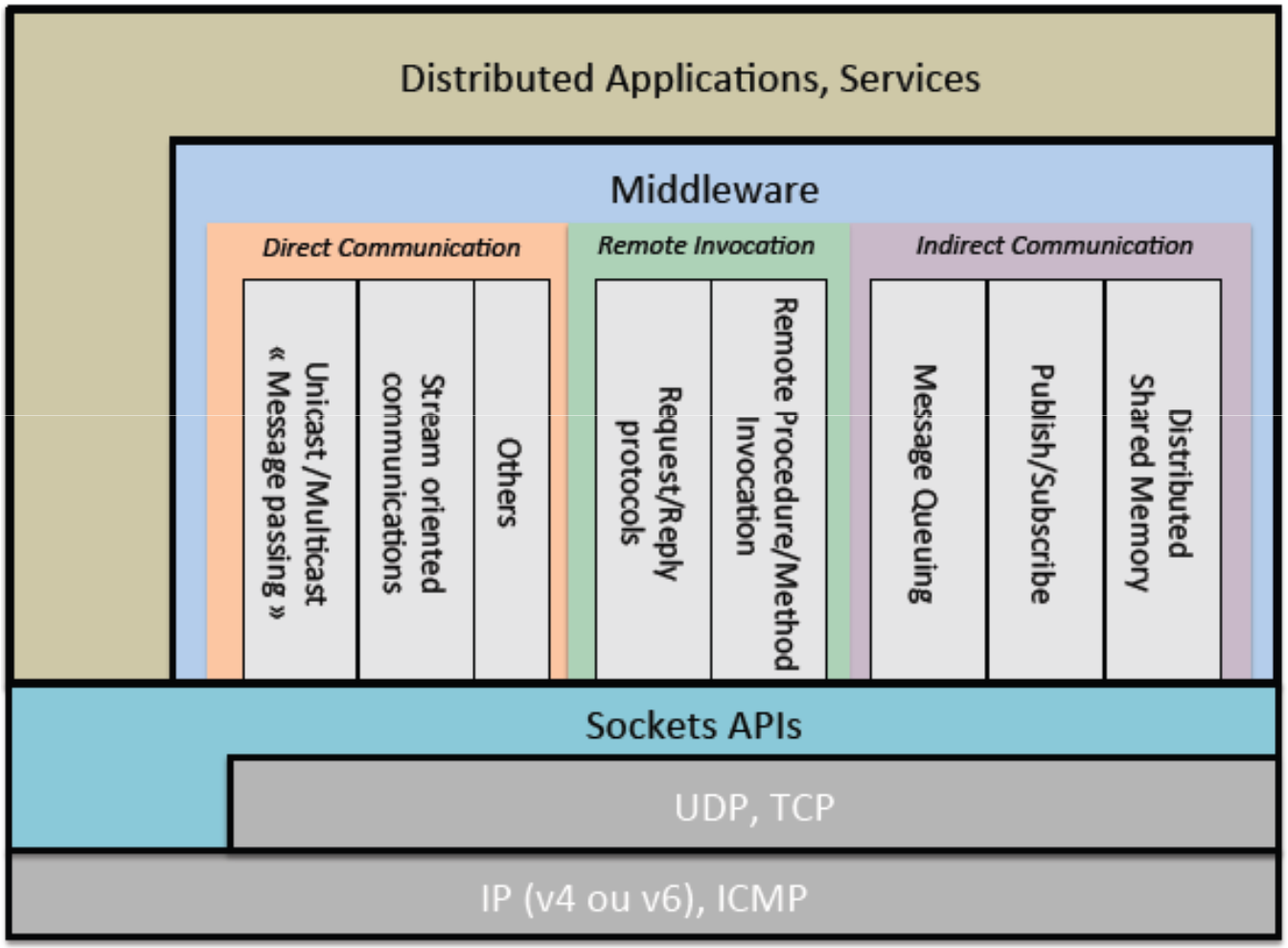
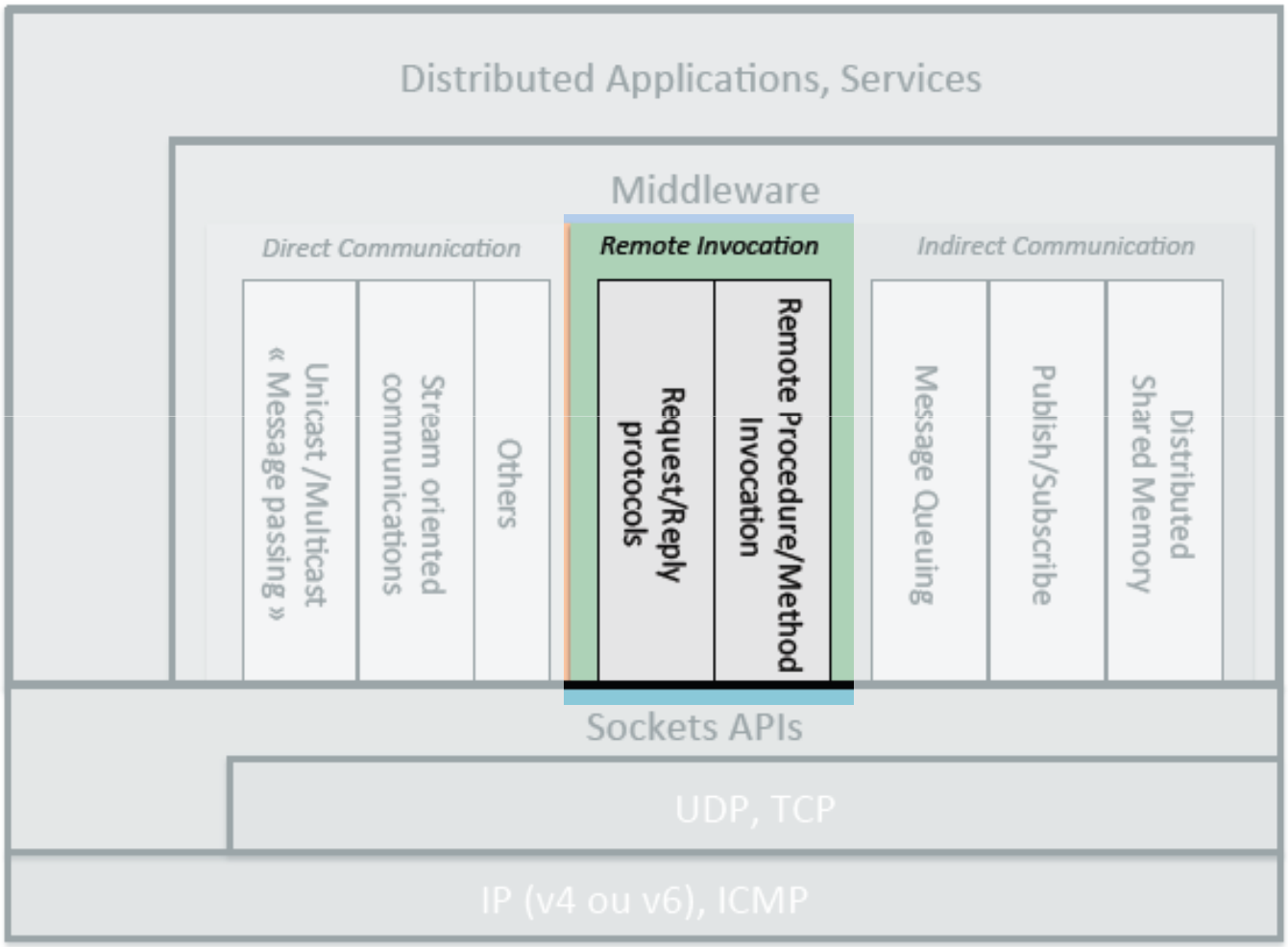
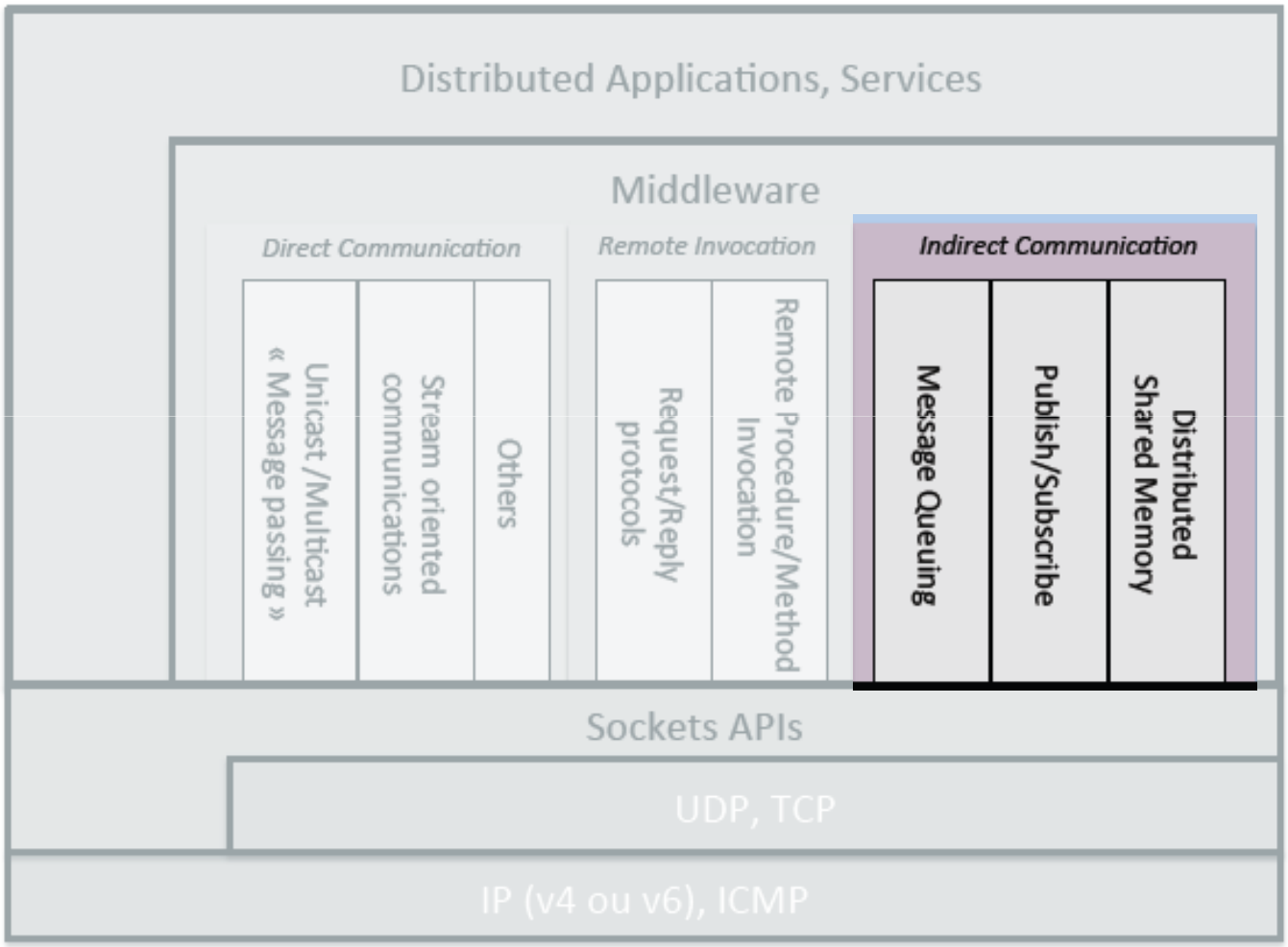
Mise en œuvre du modèle Client/Serveur
Les modèles de communication



Les middlewares d'appel à distance



Les middlewares de communication orientés messages (MOM)



Communication indirecte orientée messages

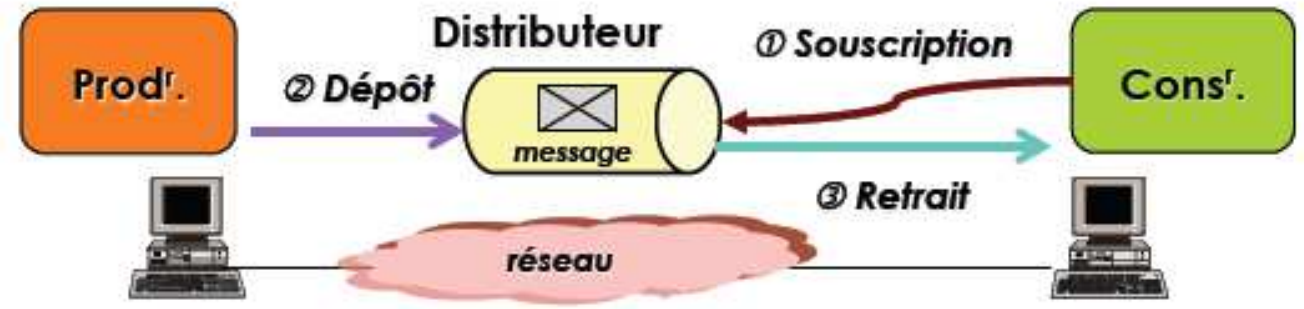
- Transmission de messages entre un émetteur (rôle de producteur) et un récepteur (rôle de consommateur)
- Communication asynchrone
- Les rôles sont définis par une intéraction particulière (une entité peut occuper les deux rôles)

- 2 modèles d'exécution:
  - Message Queuing (Files de messages)
  - Publish / Subscribe (Abonnements)

- La plupart des MOM offrent ces 2 modèles

Publish/Subscribe (par abonnement)

- Transfert indirect de messages applicatifs ou d'évènements via un service intermédiaire de distribution.
- Le producteur envoie un message
  - Basé sur un sujet (subject-based)
  - Basé sur un contenu (content-based)
- Le consommateur s'abonne (à un sujet ou un contenu)
- Communication 1-N (plusieurs consommateurs peuvent s'abonner)
- Mode **Push** : le retrait est à l'initiative du producteur

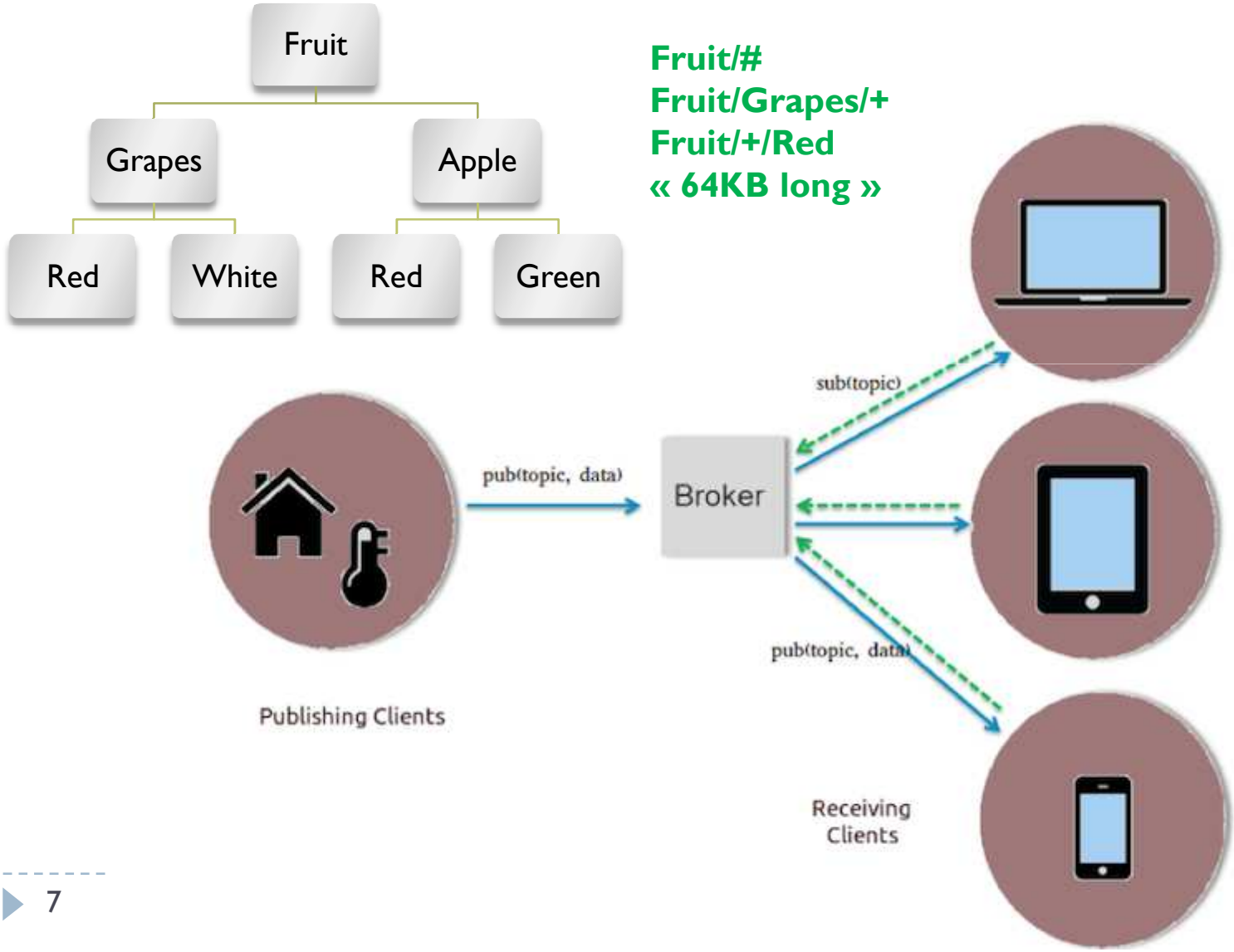


Exemples de MOM Pub/Sub

- Protocoles
  - MQTT: *Message Queuing Telemetry Transport*)
  - XMPP : *Extensible Messaging and Presence Protocol*
  - DPS : *Distributed Publish Subscribe for IoT*

- Exemples de solutions
  - Mosquito
  - HiveMQ
  - Apache ActiveMQ
  - RabbitMQ
  - IBM Webshere MQ
  - OpenDDS

MQTT Pub/Sub


Fruit
Grapes
Apple
Red
White
Red
Green
Fruit/#
Fruit/Grapes/+
Fruit/+/Red
« 64KB long »
sub(topic)
pub(topic, data)
Broker
pub(topic, data)
Publishing Clients
Receiving Clients

MQTT.fx http://mqttfx.jensd.de/

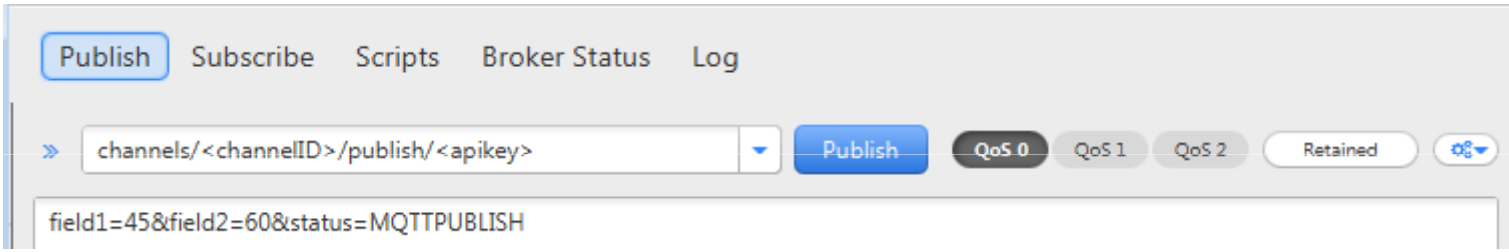
1) Connexion au broker MQTT de Thingspeak:

Broker Address mqtt.thingspeak.com and **Port** 1883

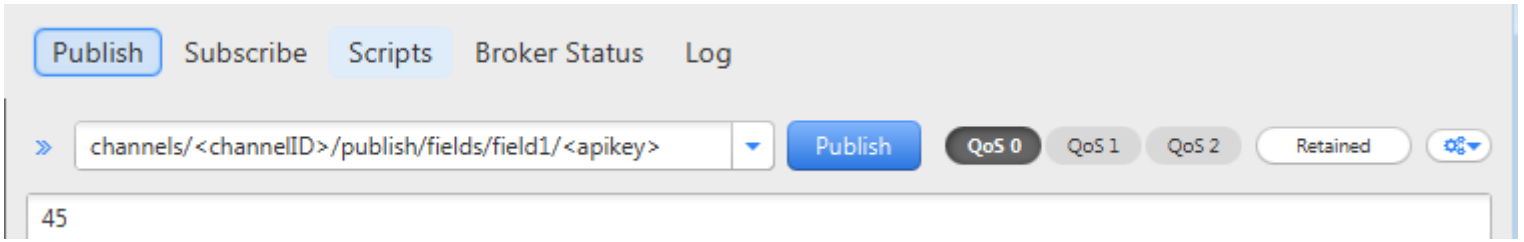
2) User Credentials :

➔ your MQTT API key from your ThingSpeak **Account > My Profile** page.

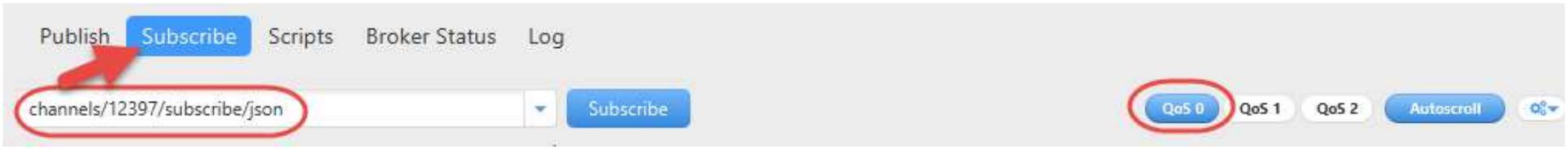
3) Publish to Channel feed :



OU BIEN



4) Suscribe to Channel feed :



Client MQTT -avec l'API Paho java

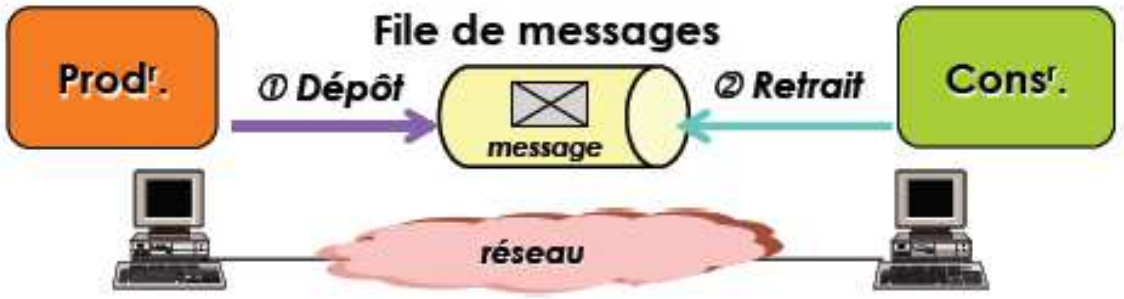
```
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;

public class MqttPublishSample {

    public static void main(String[] args) {
        String topic = "MQTT Examples";
        String content = "Message from MqttPublishSample";
        int qos = 2;
        String broker = "tcp://mqtt.eclipse.org:1883";
        String clientId = "JavaSample";
        MemoryPersistence persistence = new MemoryPersistence();
```

Client MQTT -avec l'API Paho java (2)

```
try {
        MqttClient sampleClient = new MqttClient(broker, clientId, persistence);
        MqttConnectOptions connOpts = new MqttConnectOptions();
        connOpts.setCleanSession(true);
        System.out.println("Connecting to broker: "+broker);
        sampleClient.connect(connOpts);
        System.out.println("Connected");
        System.out.println("Publishing message: "+content);
        MqttMessage message = new MqttMessage(content.getBytes());
        message.setQos(qos);
        sampleClient.publish(topic, message);
        System.out.println("Message published");
        sampleClient.disconnect();
        System.out.println("Disconnected");
        System.exit(0); }
catch(MqttException me) {
                me.printStackTrace(); } } }
```

Message Queuing (point-à-point)

- Transfert indirect de messages applicatifs
  - Producteur : dépôt dans la file
  - Consommateur : retrait de la file
- Mode **Pull** : le retrait est à l'initiative du récepteur



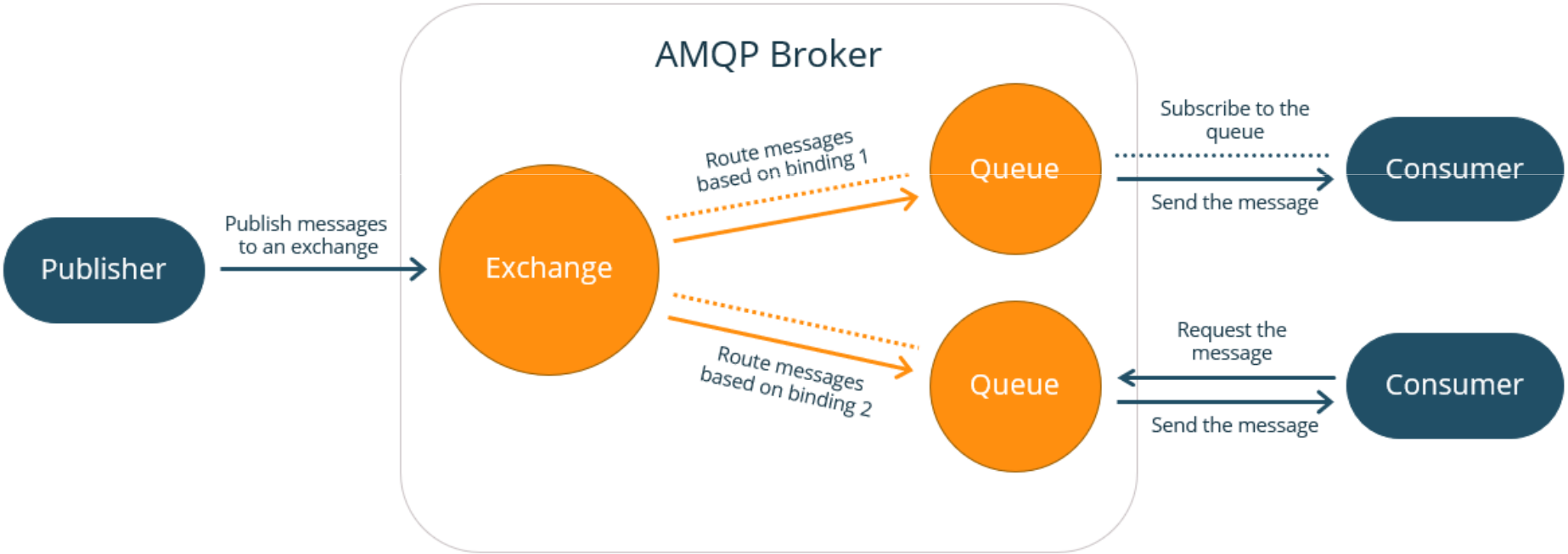
Fonctionnalités d'un MOM

- Transport des messages applicatifs :
  - Offre un bus de communication entre les applications
  - La detistination est une file d'attente/ sujet
- Communication asynchrone:
  - L'application émettrice d'un message et l'application réceptrice du message n'ont pas besoin d'être actives en même temps. La file d'attente reçoit le message de l'application émettrice et le stocke jusqu'à ce que l'application réceptrice vienne lire le message.
- Routage:
  - Les messages peuvent être routés entre MOM. Par exemple, pour router un message entre deux sites distants disposant chacun d'un MOM installé localement.
- Persistance des messages:
  - Les messages présents dans les files d'attente peuvent être sauvegardés sur un support physique pour en assurer la conservation en cas de panne.
- Fiabilité:
  - Chaque message envoyé par une application fait l'objet d'un accusé de réception par le MOM. Chaque application qui consomme un message envoie un accusé de réception au MOM

Exemples de MOM -Msg queuing

- Standards
  - CORBA Event Service – OMG
  - JMS (Java Message Service) – Sun/Oracle
  - DDS (Data Distribution Service) – OMG
  - AMQP (Advanced Message Queuing Protocol) – OASIS
- Exemples de solutions
  - OpenJMS, Joram
  - Apache ActiveMQ
  - RabbitMQ
  - IBM Webshere MQ
  - OpenDDS

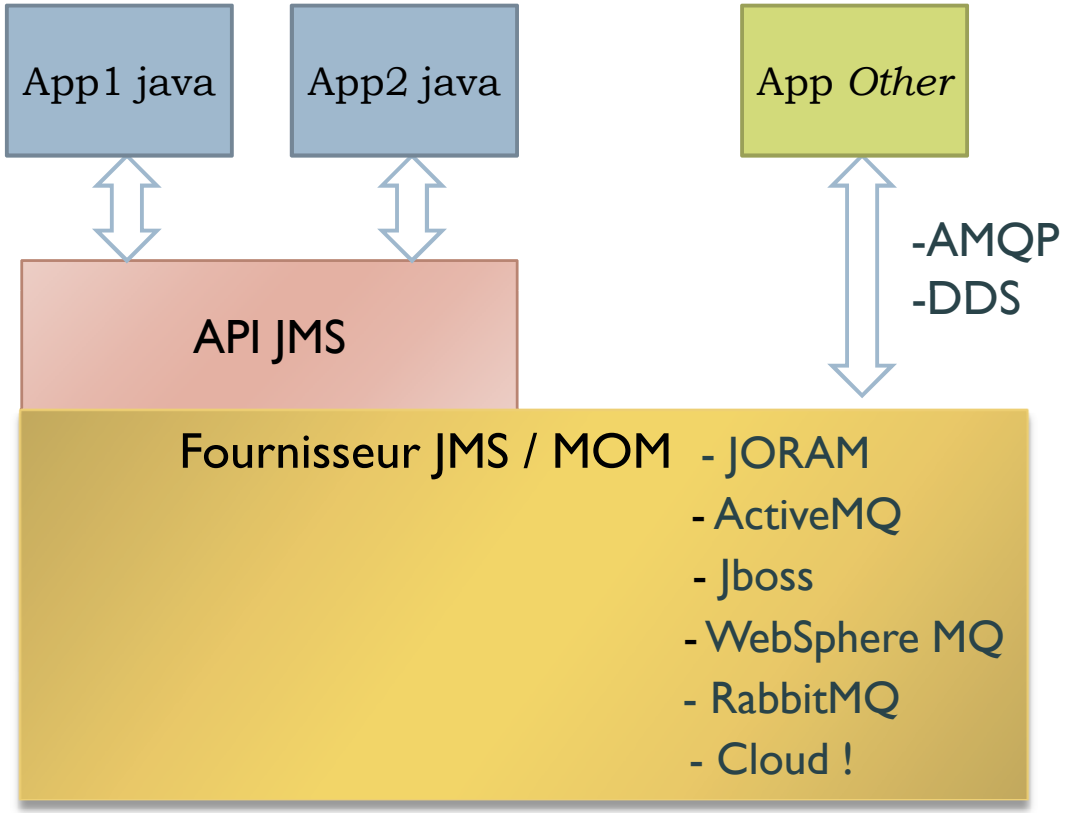
Exemple: AMQP(*Advanced Message Queuing Protocol)*



Exemple de MOM: JMS (*Java Messaging Service*)

- JMS est un standard Java qui permet d'implémenter des MOM fonctionnant en mode Message Queuing
- L'architecture JMS nécessite :
  - l'API JMS qui permet d'écrire les applications qui s'échangent les message : package javax.jms
  - Un fournisseur (provider) JMS qui gère l'administration des files de message, le stockage des messages, les sujets...
    - Il peut soit être implémenté en Java ou être un front-end JMS pour une application existante écrite dans un autre langage.
  - Un administrateur d'objets et de connexions (JNDI)
    Java Naming and Directory Interface

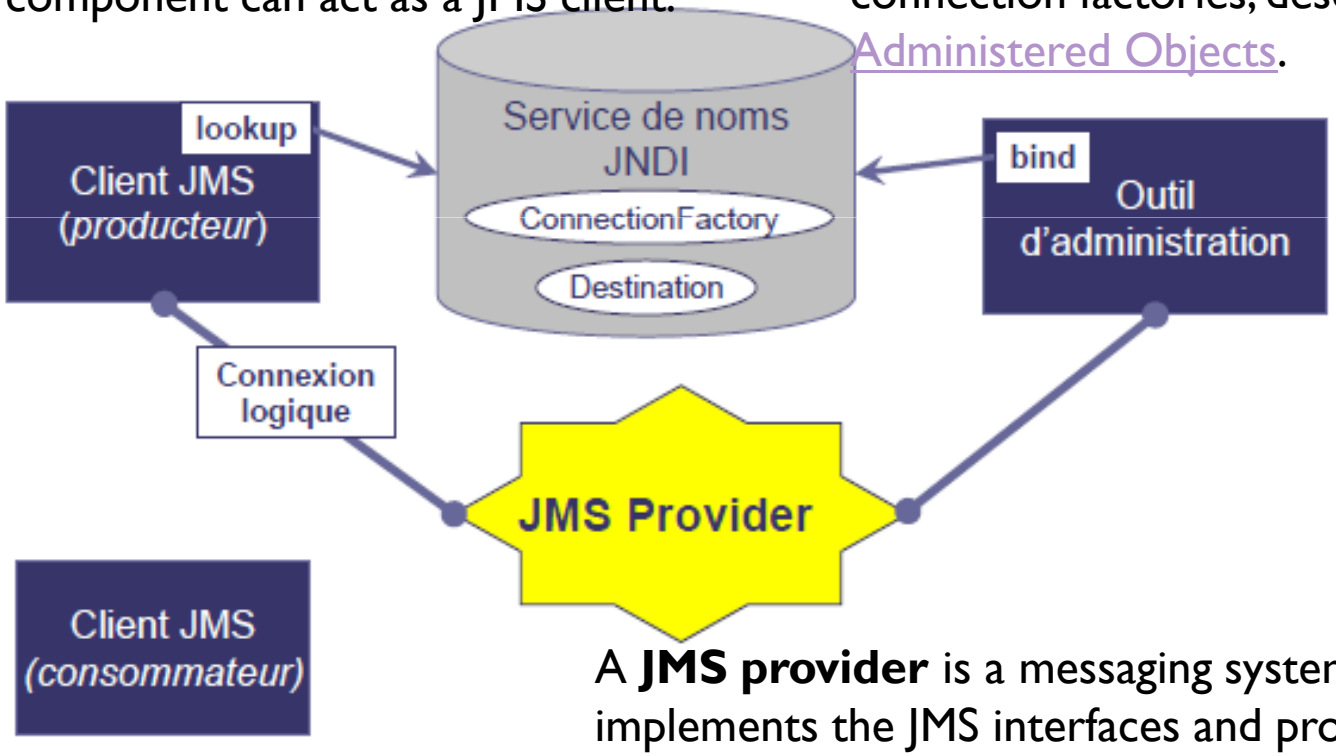
JMS


App1 java
App2 java
App *Other*
-AMQP
-DDS
API JMS
Fournisseur JMS / MOM
- JORAM
- ActiveMQ
- Jboss
- WebSphere MQ
- RabbitMQ
- Cloud !

Architecture d'une application JMS

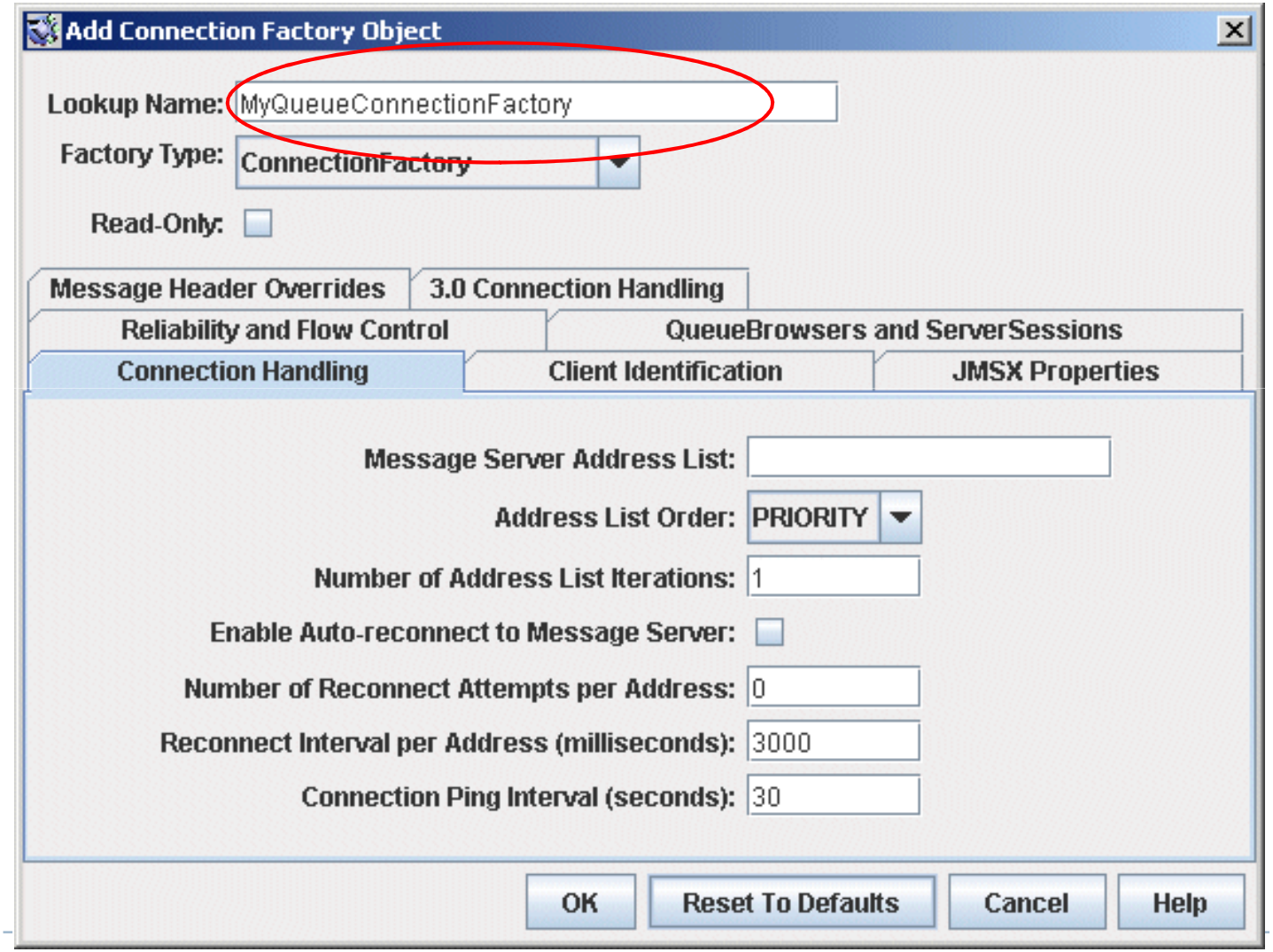
JMS clients are the programs or components, written in the Java programming language, that produce and consume messages. Any Java EE application component can act as a JMS client.

Administered objects are preconfigured JMS objects created by an administrator for the use of clients. The two kinds of JMS administered objects are destinations and connection factories, described in JMS Administered Objects.

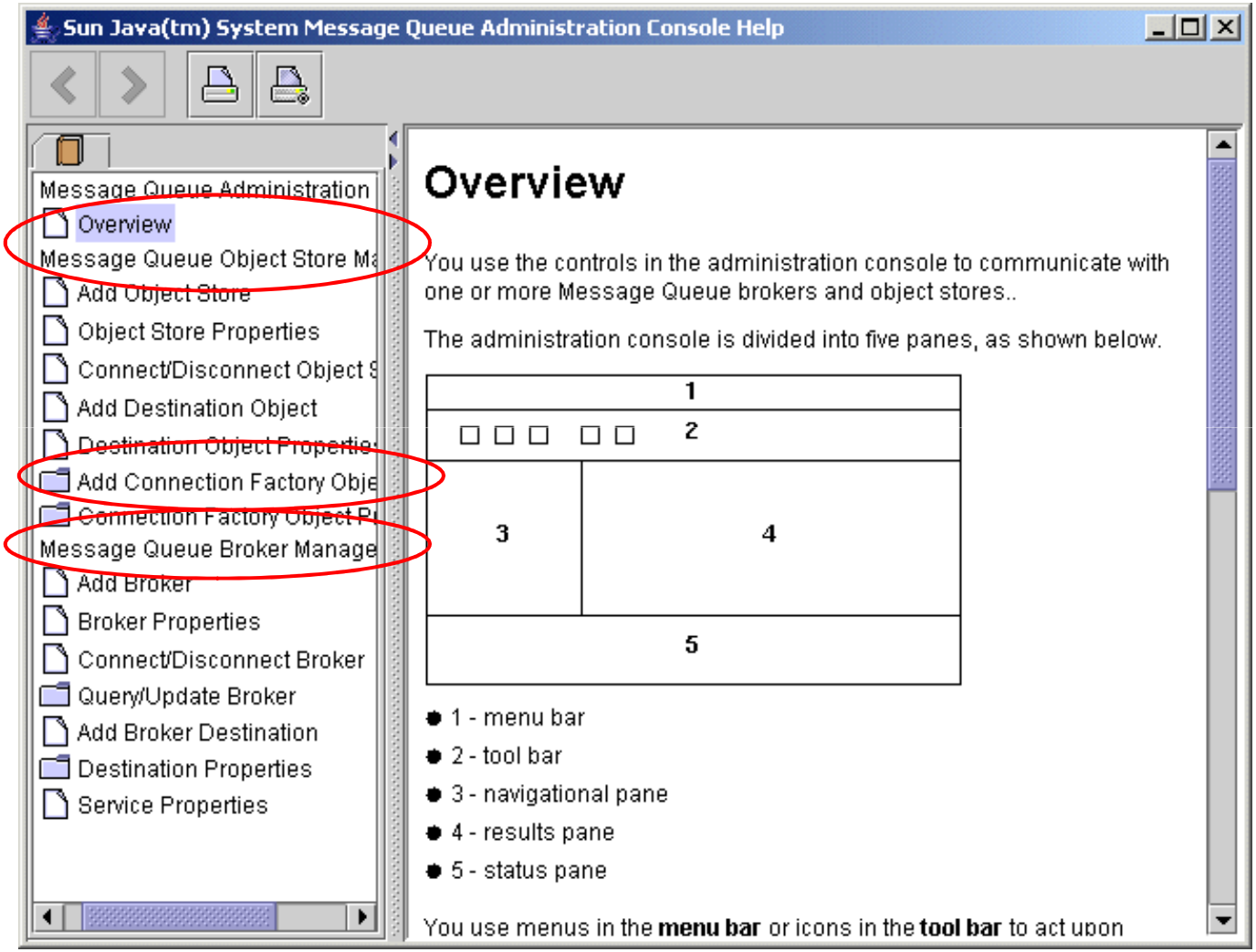


A **JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the Java EE platform includes a JMS provider.

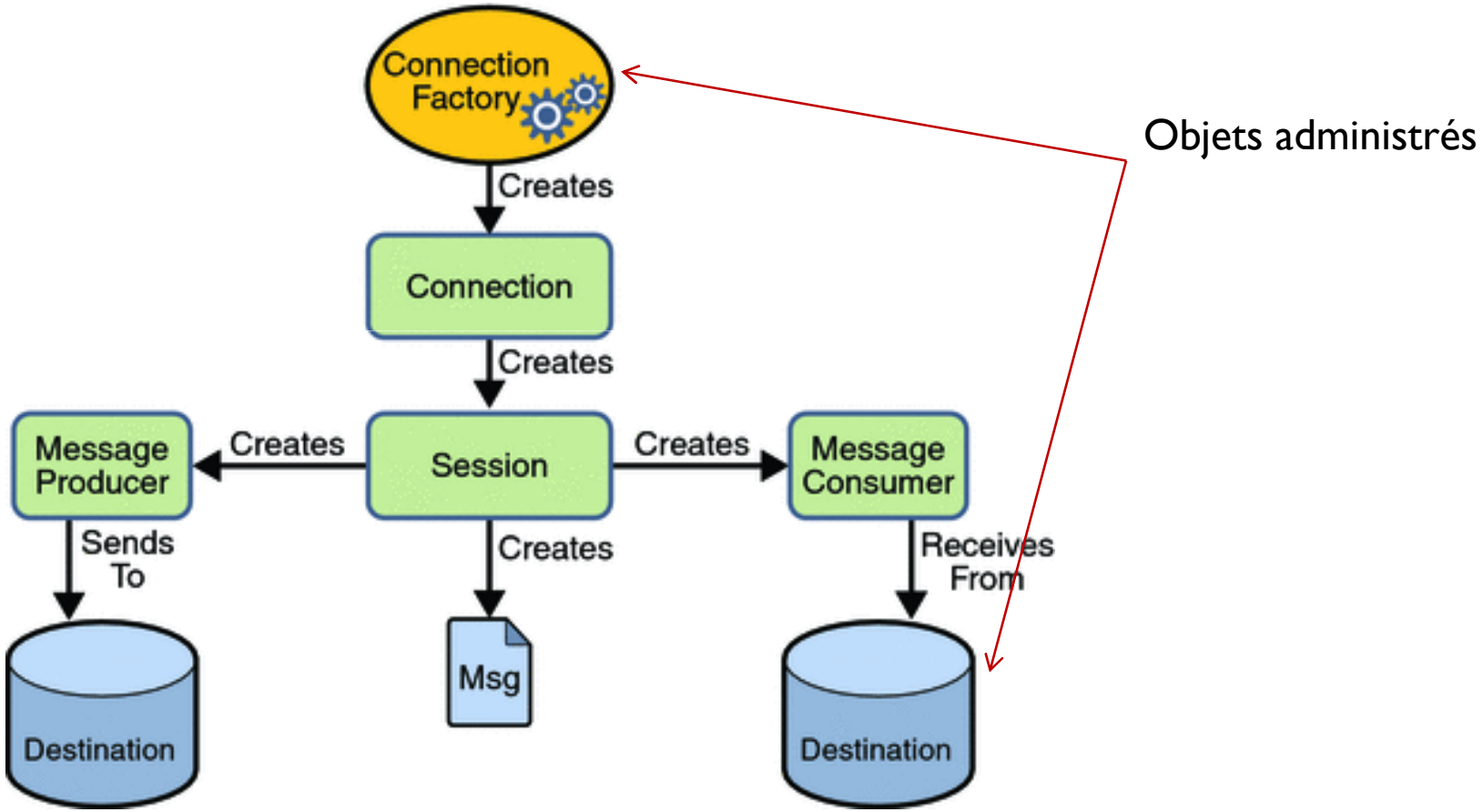
La console d'administration Oracle GlassFish



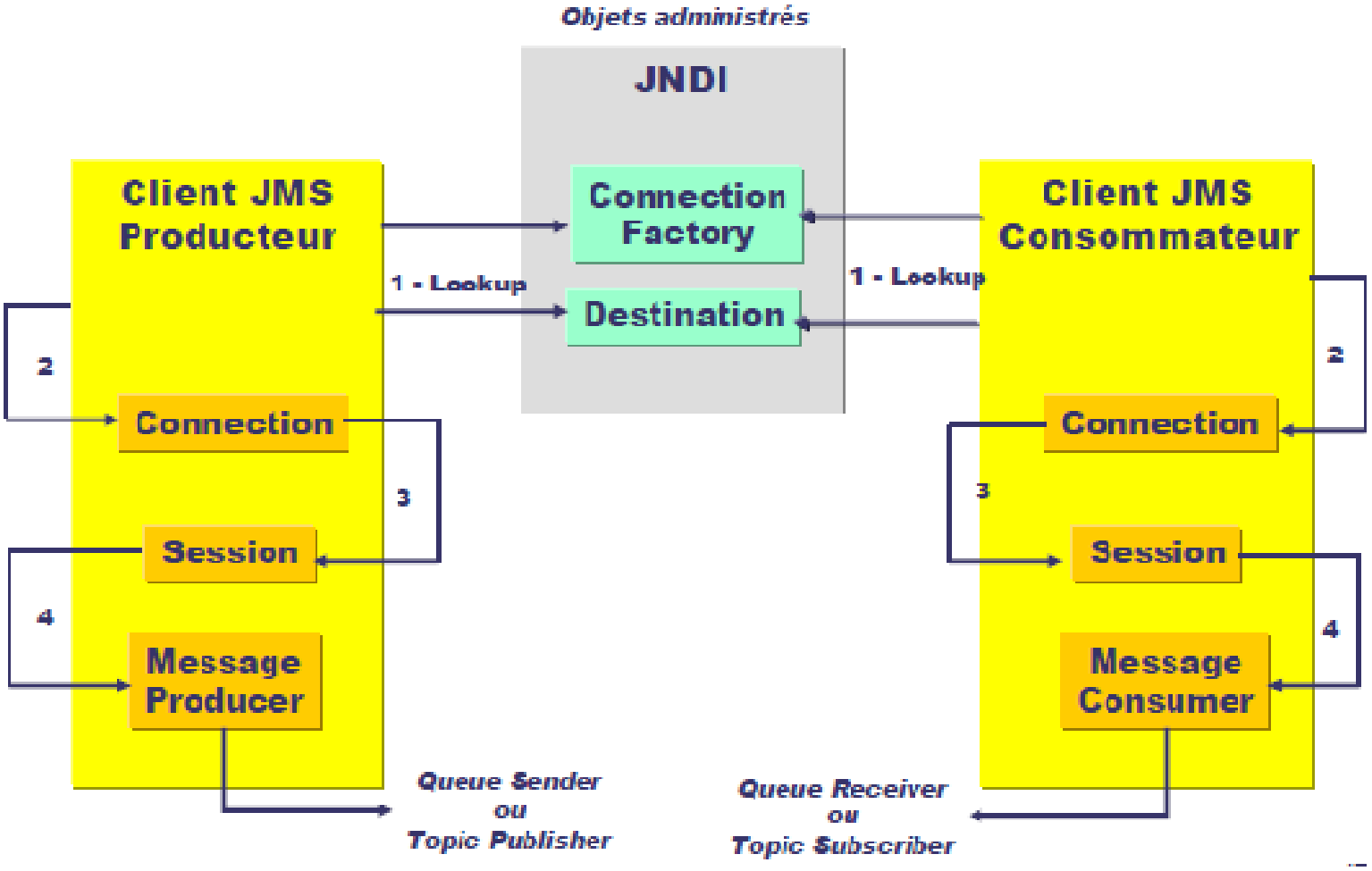
La console d'administration Oracle GlassFish



Modèle de programmation de l'API JMS



Modèle de programmation de l'API JMS (2)



Les classes de l'API JMS

Interface « parent »	*Point-à-point*	*Publish/Subscribe*
Destination	Queue	Topic
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

JMS ConnectionFactory

- A **connection factory** is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator.
- Each connection factory is an instance of the ConnectionFactory, QueueConnectionFactory, or TopicConnectionFactory interface.
- At the beginning of a JMS client program, you usually inject a connection factory resource into a ConnectionFactory object.
- For example, the following code fragment specifies a resource whose JNDI name is jms/ConnectionFactory and assigns it to a ConnectionFactory object:

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

JMS Destinations

- A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes (queues/ topics.)
- A JMS application can use multiple queues or topics (or both)
- The following code specifies two resources, a queue and a topic. The resource names are mapped to destinations created in the JNDI namespace:

```
@Resource(mappedName="jms/Queue")
private static Queue queue;
@Resource(mappedName="jms/Topic")
private static Topic topic;
```

- With the common interfaces, you can mix or match connection factories and destinations. That is, in addition to using the ConnectionFactory interface, you can inject a QueueConnectionFactory resource and use it with a Topic, and you can inject a TopicConnectionFactory resource and use it with a Queue. The behavior of the application will depend on the kind of destination you use and not on the kind of connection factory you use.

JMS Connections

- A **connection** encapsulates a virtual connection with a JMS provider.
- A connection could represent an open TCP/IP socket between a client and a provider service daemon.
- You use a connection to create one or more sessions.
- Connections implement the Connection interface.
- When you have a ConnectionFactory object, you can use it to create a Connection:

```
Connection connection = connectionFactory.createConnection();
```

- Before an application completes, you must close any connections that you have created: .

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

- Before your application can consume messages, you must call the connection's start method
- If you want to stop message delivery temporarily without closing the connection, you call the stop method.

JMS Sessions

- A **session** is a single-threaded context for producing and consuming messages. You use sessions to create the following:
  - Message producers
  - Message consumers
  - Messages
  - Queue browsers

- Sessions serialize the execution of message listeners.
- After you create a Connection object, you use it to create a Session:

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

- The first argument means that the session is not transacted; the second means that the session automatically acknowledges messages when they have been received successfully.

JMS Message Producers

- A **message producer** is an object that is created by a session and used for sending messages to a destination.
- The following examples show that you can create a producer for a Destination object, aQueue object, or a Topic object:

```
MessageProducer producer = session.createProducer(dest);
MessageProducer producer = session.createProducer(queue);
MessageProducer producer = session.createProducer(topic);
```

- You can create an unidentified producer by specifying null as the argument to create Producer. With an unidentified producer, you do not specify a destination until you send a message.
- After you have created a message producer, you can use it to send messages by using the send method:

```
producer.send(message);
```

- You must first create the messages;
- If you created an unidentified producer :

```
MessageProducer anon_prod = session.createProducer(null);
anon_prod.send(dest, message);
```

JMS Message Consumers

- A **message consumer** is an object that is created by a session and used for receiving messages sent to a destination.
  - A message consumer allows a JMS client to register interest in a destination with a JMS provider.
- The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.
- For example, you could use a Session to create a MessageConsumer for a Destination object, a Queue object, or a Topic object:

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

- You use the receive method to consume a message synchronously. You can use this method at any time after you call the start method:

```
connection.start();
Message m = consumer.receive();
connection.start(); Message m = consumer.receive(1000); // time out after a second
```

- To consume a message asynchronously, you use a message listener.

Pour aller plus loin…

- JMS Message Listeners
- JMS Message Selectors
- JMS Messages
- Message Headers
- JMS Queue Browsers
- JMS Exception Handling
- …

https://docs.oracle.com/javaee/5/tutorial/doc/l

Exemple de code émetteur

```
//The sending program, producer/src/java/Producer.java, performs the following steps:
//Injects resources for a connection factory, queue, and topic:
@Resource(mappedName="jms/ConnectionFactory") private static
ConnectionFactory connectionFactory;
@Resource(mappedName="jms/Queue")private static Queue queue;
@Resource(mappedName="jms/Topic")private static Topic topic;
//Retrieves and verifies command-line arguments that specify the destination type and
//the number of arguments:
final int NUM_MSGS;
String destType = args[0];
System.out.println("Destination type is " + destType);
if ( ! ( destType.equals("queue") || destType.equals("topic") ) ) {
            System.err.println("Argument must be \"queue\" or " + "\"topic\"");
            System.exit(1); }
 if (args.length == 2){
            NUM_MSGS = (new Integer(args[1])).intValue(); }
else { NUM_MSGS = 1; }
```

Exemple de code émetteur (2)

```
//Assigns either the queue or topic to a destination object, based on the specified
//destination type:
Destination dest = null;
try {
    if (destType.equals("queue")) {
            dest = (Destination) queue; }
    else {
            dest = (Destination) topic; }
} catch (Exception e) {
            System.err.println("Error setting destination: " + e.toString());
            e.printStackTrace(); System.exit(1); }
//Creates a Connection and a Session:
Connection connection = connectionFactory.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Exemple de code émetteur (3)

```
//Creates a MessageProducer and a TextMessage:
MessageProducer producer = session.createProducer(dest);
TextMessage message = session.createTextMessage();
//Sends one or more messages to the destination:
for (int i = 0; i < NUM_MSGS; i++) {
        message.setText("This is message " + (i + 1));
        System.out.println("Sending message: " + message.getText());
        producer.send(message); }
//Sends an empty control message to indicate the end of the message stream:
producer.send(session.createMessage());
//Sending an empty message of no specified type is a convenient way to indicate to the
//consumer that the final message has arrived.
//Closes the connection in a finally block, automatically closing the session
//and MessageProducer:
} finally {
        if (connection != null) {
        try { connection.close(); }
        catch (JMSException e) { } } }
```