11 Premisele oferite de JMS pentru un design SOA

Maniera de proiectare a unui sistem de *trading* fundamentat pe principiile unei arhitecturi orientate pe servicii (*Service Oriented Architecture* – SOA) este esenţial determinată de mecanismele concrete prin care fiecare componentă a sistemul:

- pune la dispoziţia celorlalte componente serviciile pe care le oferă;
- respectiv are acces la serviciile oferite de celelalte componente.

Aşa cum am văzut în *Fascicula 6*, JMS oferă două modele (domenii) principale de comunicaţie: punct la punct (*p2p*), respectiv publicare şi subscriere (*pub/sub*). Fiecare din aceste modele furnizează modalităţi specifice pentru susţinerea unui design SOA.

11.1 Modelul de comunicaţie *p2p*

Cele mai importante caracteristici ale modelului de comunicaţie punct la punct sunt următoarele:

- mesajele sunt interschimbate prin intermediul unui canal virtual de comunicaţie numit coadă; coada joacă rolul de destinaţie pentru producătorul de mesaje şi de sursă pentru consumatorul respectivelor mesaje;
- fiecare mesaj este livrat către un singur receptor-consumator; dacă mai mulţi receptori au ca sursă aceeaşi coadă de mesaje, atunci fiecare mesaj din coadă poate fi consumat doar de un singur receptor;
- mesajele cozii sunt ordonate; mesajele sunt livrate din coadă către consumator în ordinea în care ele au fost depuse în coadă de către producător; pe măsură ce mesajele sunt consumate, ele sunt şterse din capul structurii de coadă (cu excepţia situaţiei în care se utilizează atributul de prioritate al mesajelor, situaţie în care ordinea de citire/ştergere poate fi alterată);

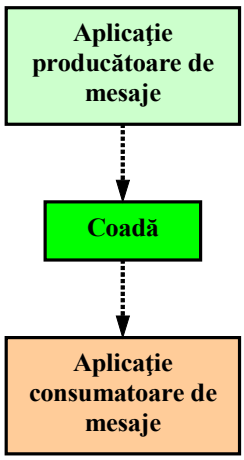

Figura 11. 1 – modelul p2p - mecanismul fire-and- forget asincron

- nu există nici un fel de cuplare directă între producător şi consumator; pot fi adăugaţi sau eliminaţi din sistem în mod dinamic, atât producători, cât şi

consumatori de mesaje, fără întreruperea funcționării sistemului ca ansamblu, permițând adaptarea complexității acestuia la cerințele concrete de exploatare.

Modelul *p2p* oferă la rândul său două tipuri de transmitere a mesajelor între clienții JMS.

- Transmitere prin intermediul mecanismului *fire-and-forget* asincron; producătorul trimite mesajul către o coadă și nu așteaptă să primească un răspuns de la aplicația (aplicațiile) consumatoare, cel puțin nu imediat; acest mecanism de transmitere poate fi folosit pentru a declanșa un anumit eveniment în sistem, sau pentru plasa o cerere pentru anumit serviciu care nu necesită un răspuns (***Figura 11.1***);

Prezentăm în continuare clasa Java **`AbstractQueueSenderNoReply`**, care implementează o conexiune JMS pentru un producător de mesaje în modelul *p2p*, utilizând mecanismul *fire-and-forget*. Clasa face parte din API-ul construit pentru proiectul ASETS (ASE Trading System) și am lăsat în mod deliberat comentariile din codul Java în limba engleză.

```
package asets.api;

import javax.jms.*;


public abstract class AbstractQueueSenderNoReply
{
    private QueueSession sendSession = null;
    private QueueSender sender = null;
    private QueueConnection connection = null;
    private String userID = null;

    // Constructor used to initialize AbstractQueueSenderNoReply
     public AbstractQueueSenderNoReply(String imqAddress, String queueName, String
        userID) throws JMSException, Exception
    {
        // Create a JMS connection factory
        String addressList = imqAddress;
        com.sun.messaging.QueueConnectionFactory queueConnectionFactory =
         new com.sun.messaging.QueueConnectionFactory();
                queueConnectionFactory.setProperty(com.sun.messaging.ConnectionConfigur
        ation.
        imqAddressList, addressList);

        // Create a JMS connection
        QueueConnection queueConnection =
        queueConnectionFactory.createQueueConnection();
        queueConnection.setClientID(userID);

        // Create JMS session object
        QueueSession queueSession = queueConnection.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);

        // Create a JMS queue
        Queue destination = queueSession.createQueue(queueName);

        // Create a JMS sender
        QueueSender queueSender = queueSession.createSender(destination);

        // Intialize the AbstractQueueSenderNoReply application variables
        this.connection = queueConnection;
        this.sendSession = queueSession;
        this.sender = queueSender;
        this.userID = userID;

        // Start the JMS connection; allows messages to be delivered
        connection.start();
```

```
    }

    // Close the JMS connection
    public void close() throws JMSException
    {
        if (connection != null)
            connection.close();
    }

    public QueueSession getSession()
    {
        return this.sendSession;
    }

    public QueueSender getSender()
    {
        return this.sender;
    }

    public String getUserID()
    {
        return this.userID;
    }
}
```

- Transmitere prin utilizarea mecanismului de cerere-răspuns realizate asincron; producătorul trimite un mesaj către o coadă şi apoi îşi opreşte execuţia (blochează) şi aşteaptă primirea unui mesaj de răspuns de la destinatar; acest mecanism de cerere-răspuns oferă însă un grad ridicat de decuplare a celor două aplicaţii implicate, deosebindu-se fundamental de maniera de comunicaţie de tip cerere-răspuns oferită de platformele bazate pe RPC; fiind o comunicaţie asincronă, mecanismul utilizează două cozi: într-una este depus mesajul de cerere de către aplicaţia care joacă rolul de client (potenţial beneficiar al unui serviciu), iar în cealaltă este plasat mesajul de răspuns de către aplicaţia care joacă rolul de server (furnizor al unui serviciu); scenariu descris este ilustrat în ***Figura 11.2***.

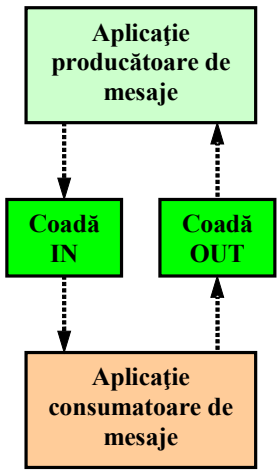

Figura 11.2 – modelul p2p – mecanismul asincron de cerere-răspuns

Prezentăm în continuare clasa Java **`AbstractQueueSenderWithReply`**, care implementează o conexiune JMS pentru un producător de mesaje în modelul *p2p*,

utilizând mecanismul cerere-răspuns realizate asincron. Trebuie remarcată mai jos, pe lângă declararea cozii pentru mesajele de cerere şi declararea cozii pe care producătorul de mesaje urmează să aştepte răspuns pentru cererile trimise.

```
public abstract class AbstractQueueSenderWithReply
{
    private QueueSession sendSession = null;
    private QueueConnection connection = null;
    private Queue requestQueue = null;
    private Queue replyQueue = null;
    private String userID = null;

    // Constructor used to initialize AbstractQueueSenderWithReply
    public AbstractQueueSenderWithReply(String imqAddress, String requestQueueName,
            String replyQueueName, String userID) throws JMSException, Exception
    {
        // Create a JMS connection factory
        String addressList = imqAddress;
        com.sun.messaging.QueueConnectionFactory queueConnectionFactory =
        new com.sun.messaging.QueueConnectionFactory();
                queueConnectionFactory.setProperty(com.sun.messaging.ConnectionConfigur
        ation.
        imqAddressList, addressList);

        // Create a JMS connection
        QueueConnection queueConnection =
        queueConnectionFactory.createQueueConnection();
        queueConnection.setClientID(userID);

        // Create JMS session object
        QueueSession queueSession = queueConnection.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);

        // Create JMS queues: request and reply, respectively
        Queue requestDestination = queueSession.createQueue(requestQueueName);
        Queue replyDestination = queueSession.createQueue(replyQueueName);

        // Intialize the AbstractQueueSenderWithReply application variables
        this.connection = queueConnection;
        this.sendSession = queueSession;
        this.requestQueue = requestDestination;
        this.replyQueue = replyDestination;
        this.userID = userID;

        // Start the JMS connection; allows messages to be delivered
        connection.start();
    }

    // Close the JMS connection
    public void close() throws JMSException
    {
        if (connection != null)
            connection.close();
    }

    public QueueSession getSession()
    {
        return this.sendSession;
    }

    public Queue getRequestQueue()
    {
        return this.requestQueue;
    }

      public Queue getReplyQueue()
    {
        return this.replyQueue;
    }
```

```
    public String getUserID()
    {
        return this.userID;
    }
}
```

În general, mecanismul *fire-and-forget* este perceput ca util atunci când nu este necesar un răspuns (încă o dată, cel puțin nu unul imediat) din partea consumatorului. De exemplu, se poate folosi acest mecanism atunci când este necesară lansarea unei cereri pentru un raport cu privire la activitatea tranzacțională a unui investitor pe luna precedentă, raport ce este acceptabil de a fi generat și primit oricând până la sfârșitului zilei de lucru. Utilizarea acestui mecanism de transmitere a mesajelor nu trebuie însă limitată numai la accesul la astfel de servicii.
Trebuie să remarcăm, de la bun început și vom mai repeta acest aspect de-a lungul expunerii noastre, pentru că este fundamental și poate fi cu ușurință omis, faptul că atunci când doi clienți JMS comunică între ei mesaje care sunt marcate ca persistente de către serverul de mesagerie, transmisia acestor mesaje se realizează întotdeauna prin confirmarea primirii, la nivelul interacțiunii celor doi clienți cu platforma de mesagerie JMS. Este vorba despre un mecanism de confirmare care este transparent programatorului de aplicații și care funcționează sub nivelul interfeței JMS, care oferă cele două mecanismele *p2p* prezentate mai sunt.
Cu alte cuvinte, un mesaj transmis de către aplicația **A**, către aplicația **B**, prin mecanismul *fire-and-forget*, poate să creeze programatorului impresia că aplicația **A** nu primește nici un fel de răspuns cu privire la succesul sau insuccesul transmiterii mesajului produs de ea. În realitate însă, mesajul nu este trimis direct aplicație destinatar **B**, ci este recepționat prin TCP/IP de către serverul de mesagerie, care confirmă la acest nivel aplicației **A** recepționarea mesajului. Similar, la momentul livrării mesajului către aplicația **B**, serverul de mesagerie va stabili o conexiune TCP/IP cu aceasta, asigurându-se astfel de confirmarea primirii mesajului de către aplicația **B**. Practic, dacă un mesaj ajunge cu succes și este persistat de către platforma de mesagerie, atunci livrarea lui către destinație este garantat a fi realizată, mai devreme sau mai târziu.
De aceea, vorbim mai curând de un mecanism de tip *fire-and-forget* la nivel logic, de design al comunicării prin mesaje între aplicațiile unei arhitecturi orientată pe servicii, decât la nivelul siguranței transmiterii acestor mesaje.
Prezentăm în continuare clasa Java **`AbstractQueueReceiver`**, care implementează o conexiune JMS pentru un consumator de mesaje în modelul *p2p*. Trebuie remarcat faptul că stabilirea unei conexiuni pentru consumul de mesaje dintr-o coadă se realizează în aceeași manieră, indiferent dacă consumatorul va trimite sau nu un mesaj de răspuns producătorului mesajului inițial.

```
public abstract class AbstractQueueReceiver implements javax.jms.MessageListener
{
    private QueueSession receiveSession = null;
    private QueueConnection connection = null;
    private String userID = null;

    // Constructor used to initialize AbstractQueueReceiver
    public AbstractQueueReceiver(String imqAddress, String queueName, String userID)
            throws JMSException, Exception
    {
        // Create a JMS connection factory
        String addressList = imqAddress;
```

```
            com.sun.messaging.QueueConnectionFactory queueConnectionFactory =
            new com.sun.messaging.QueueConnectionFactory();
               queueConnectionFactory.setProperty(com.sun.messaging.ConnectionConfiguration
            .
            imqAddressList, addressList);

            // Create a JMS connection
            QueueConnection queueConnection =
            queueConnectionFactory.createQueueConnection();
            queueConnection.setClientID(userID);

            // Create JMS session object
            QueueSession queueSession = queueConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);

            // Create a JMS queue
            Queue destination = queueSession.createQueue(queueName);

            // Create a JMS receiver
            QueueReceiver queueReceiver = queueSession.createReceiver(destination);

            // Set a JMS message listener
            queueReceiver.setMessageListener(this);

            // Intialize the AbstractQueueReceiver application variables
            this.receiveSession = queueSession;
            this.connection = queueConnection;
            this.userID = userID;

            // Start the JMS connection; allows messages to be delivered
            connection.start();
        }

        // Close the JMS connection
        public void close() throws JMSException
        {
            if (connection != null)
                connection.close();
        }

        public QueueSession getSession()
        {
            return this.receiveSession;
        }

        public String getUserID()
        {
            return this.userID;
        }
    }
```

Folosirea în cadrul modelului *p2p* a mecanismului *fire-and-forget*, nu cere din partea clientului ce produce mesajul decât simpla transmitere a acestuia către serverul de mesagerie. Prezentăm în continuare metoda **`sendOrder`**, ce aparține clasei **`OrderSenderNoReply`**, componentă de API a proiectului ASETS.

```
    // Create and send an Order message
    public void sendOrder(Order order) throws JMSException
    {
        // Construct the message as an ObjectMessage (Order)
        ObjectMessage message = this.getSession().createObjectMessage();
        message.setObject(order);

        message.setStringProperty("MSG_TYPE", Enums.MessageType.ORDER.toString());
        message.setStringProperty("USER_ID", this.getUserID());

        // TO DO in the the derived class
```

```
        this.getSender().send(message);
    }
```

În cazul utilizării mecanismului de cerere-răspuns în manieră asincronă, producătorul mesajului trebuie să-l informeze pe consumatorul de la care așteaptă un răspuns asupra:

- cozii la care așteaptă mesajele de răspuns, utilizând metoda **`setJMSReplyTo`**, de la nivelul interfeței **`Message`**;
- manierei în care se va pune în corespondență mesajul de răspuns cu cel trimis de producător ca și cerere.

În exemplul pe care îl prezentăm în continuare, producătorul trimite către destinatar un mesaj conținând un ordin pentru tranzacționarea unui instrument financiar la o instituție bursieră. Producătorul construiește un filtru pentru coada pe care a așteaptă răspunsul de la consumator, filtru ce va permite acestuia să preia din coadă doar mesajul de răspuns corespunzător mesajului inițial de cerere. Acest lucru se realizează în exemplul nostru prin verificarea ca valoarea atributului de antet **`JMSCorrelationID`** a mesajului de răspuns, să fie egală cu valoarea **`JMSMessageID`** din mesajul de cerere. Implicit, consumatorul mesajului inițial, atunci când construiește mesajul de răspuns, va trebui să seteze valoarea atributului de antet **`JMSCorrelationID`** al mesajului de răspuns, cu valoarea atributului de antet **`JMSMessageID`** al mesajului de cerere. Metoda Java prezentată mai jos aparține clasei **`OrderSenderWithReply`**, componentă de API a proiectului ASETS.

```
// Create and send an Order message, waiting for reply
public Order sendOrderGetReply(Order order) throws JMSException, Exception
{
    // Construct Order request message
    ObjectMessage message = this.getSession().createObjectMessage();
    message.setObject(order);

    message.setStringProperty("MSG_TYPE", Enums.MessageType.ORDER.toString());
    message.setStringProperty("USER_ID", order.getUserID());
    message.setJMSReplyTo(this.getReplyQueue());

    // Create Order sender and send the message
    QueueSender requestSender =
    this.getSession().createSender(this.getRequestQueue());
    requestSender.send(message);

    // Wait and process Order reply
    String filter = "JMSCorrelationID = '" + message.getJMSMessageID() + "'";
    QueueReceiver replyReceiver=
    this.getSession().createReceiver(this.getReplyQueue(), filter);

    Message replyMessage = replyReceiver.receive(30000);
    if (replyMessage == null)
    {
        System.out.println("Destination not responding");
    } else {
        if (replyMessage instanceof ObjectMessage)
        {
            if(replyMessage.getStringProperty("MSG_TYPE").
             equals(Enums.MessageType.ORDER.toString()) &&
             replyMessage.getStringProperty("USER_ID").
             equals(order.getUserID()))
            {
                ObjectMessage replyObject = (ObjectMessage)replyMessage;
                Order replyOrder = (Order)replyObject.getObject();
```

```
                return replyOrder;
            }
        } else {
            throw new IllegalArgumentException("Invalid message type");
        }
    }

    return null;
}
```

Să observăm în metoda **`OrderSenderWithReply`**, prezentată mai sus, maniera asincronă de implementare a mecanismului cerere-răspuns de către platforma JMS, respectiv faptul că, atât mesajul de cerere, cât și cel de răspuns, sunt transmise pe cozi distincte, iar răspunsul este așteptat de producătorul mesajului de cerere pentru o perioadă limitată de timp (în exemplul nostru, de 30 de secunde), după care își continuă fluxul propriu de procesare. Aplicația care trimite cererea nu este blocată indefinit, dacă nu primește un răspuns sincron, așa cum se întâmplă în cazul platformelor de comunicație fundamentate pe RPC.
La nivelul consumatorului de mesaje în modelul *p2p*, diferența dintre utilizarea mecanismului *fire-and-forget* sau a celui de cerere-răspuns (ambele mecanisme asincrone) este dată de felul în care este procesat mesajul în metoda **`onMessage(Message`** `message`**)**. În exemplul următor este prezentată consumarea unui mesaj, conținând un ordin de vânzare sau cumpărare a unui instrument financiar, ordin pentru care consumatorul nu trebuie să trimită înapoi un răspuns aplicației care l-a produs (mecanismul *fire-and-forget*). Codul Java exemplificat aici face de asemenea parte din API-ul proiectului ASETS.

```
// Receive messages from Order queue
public void onMessage(Message message)
{
    try
    {
        if (message instanceof ObjectMessage)
        {
            if(message.getStringProperty("MSG_TYPE").
            equals(Enums.MessageType.ORDER.toString()))

            // TO DO in the the derived class

            {
                ObjectMessage objectMessage = (ObjectMessage)message;
                processOrder((Order)objectMessage.getObject());
            }
        }
    } catch (JMSException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

protected void processOrder(Order order) throws Exception
{

    // TO DO in the the derived class

}
```

În exemplul următor este prezentată consumarea unui mesaj, conținând un ordin de vânzare sau cumpărare a unui instrument financiar, ordin pentru care consumatorul

trebuie să trimită la rândul lui un mesaj de răspuns. Acest lucru se realizează punând în corespondență noul mesaj creat ca răspuns, cu cel primit sub formă de cerere, prin preluarea valorii **`JMSMessageID`** din mesajul de cerere și setarea cu această valoare a atributului de antet **`JMSCorrelationID`** pentru mesajul de răspuns.
De asemenea, așa cum am văzut în prezentarea clasei **`AbstractQueueReceiver`**, consumatorul nu își deschide de la început o conexiune pentru coada de răspuns, ci realizează acest lucru pe baza valorii setate în **`JMSReplyTo`**, din antetul mesajului de cerere. Codul Java exemplificat aici face de asemenea parte din API-ul proiectului ASETS. Mesajul de răspuns conține tot un obiect de tip **`Order`**, ale cărui câmpuri au fost actualizate în conformitate cu logica tranzacțională a sistemului de *trading*.

```
// Receive messages from Order queue
public void onMessage(Message message)
{
    try
    {
        if (message instanceof ObjectMessage)
        {
            if(message.getStringProperty("MSG_TYPE").
            equals(Enums.MessageType.ORDER.toString()))
            {
                ObjectMessage objectMessage = (ObjectMessage)message;

                // Process Order request and suply Order reply
                 Order replyOrder =
                 processOrderRequest((Order)objectMessage.getObject());

                // Construct the reply message as an ObjectMessage (Order)
                ObjectMessage replyMessage = this.getSession().createObjectMessage();
                replyMessage.setObject(replyOrder);

                replyMessage.setStringProperty("MSG_TYPE",
                Enums.MessageType.ORDER.toString());
                replyMessage.setStringProperty("USER_ID",
                message.getStringProperty("USER_ID"));

                replyMessage.setJMSCorrelationID(message.getJMSMessageID());

                // Create the sender and send Order
                QueueSender replySender =
                this.getSession().createSender((Queue)message.getJMSReplyTo());
                replySender.send(replyMessage);
            }
        }
    } catch (JMSException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

protected Order processOrderRequest(Order order) throws Exception
{
    Order replyOrder = new Order();

    // TO DO in the the derived class

    return replyOrder;
}
```

Dacă, spre exemplu, o aplicație care oferă interfața grafică pentru un sistem de *trading* poate trimite o cerere de plasare a unui nou ordin serverului de gestiune a ordinelor (*call-forward*), această cerere poate fi efectuată fie prin utilizarea mecanismului asincron de

cerere-răspuns, fie optându-se pentru plasarea ordinului fără a mai aștepta un răspuns imediat de la server, privind confirmarea primirii cererii și obținerea identificatorului asociat ordinului primit; serverul poate ulterior să informeze clientul cu privire la starea ordinului (*call-back*) prin același mecanism , sau prin utilizarea modelului *pub/sub*;

11.2 Modelul de comunicație *pub/sub*

În modelul de comunicație publicare și subscriere, producătorul publică mesajul către un *topic* sau subiect de mesagerie. Consumatorul sau consumatorii mesajului trebuie să subscrie la acel subiect de mesagerie pentru a avea acces la mesajul publicat.
Caracteristicile fundamentale ale acestui model de comunicație sunt următoarele:

- mesajele sunt interschimbate între aplicații printr-un canal virtual de comunicație numit *topic* (***Figura 11.3***);
- fiecare mesaj este livrat către mai mai mulți consumatori, care au subscris în prealabil la subiectul respectiv de mesagerie; există mai multe modalități de a subscrie la un *topic*: în manieră durabilă, nedurabilă, în mod dinamic; vom detalia mai jos aceste tipuri de subscriere;
- aplicația care publică mesaje nu știe cine a subscris la acestea, sau câte subscrieri au fost realizate;

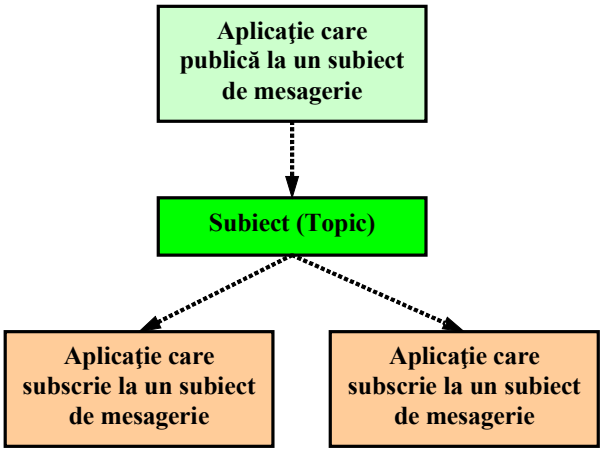

Figura 11.3 – Modelul publicare și subscriere

- mesajele sunt livrate către consumatori fără ca aceștia să le solicite în mod explicit (*pushed*); simpla aderarea a acestora (subscriere) la un subiect de mesagerie le asigură primirea mesajelor care au fost publicate la acel subiect;
- într-un astfel de model de comunicație, producătorii și consumatorii de mesaje nu sunt direct cuplați unii cu alții; este o caracteristică esențială a oricărei arhitecturi fundamentate pe un MOM, ca aplicațiile care compun sistemului să fie interconectate prin intermediul platformei de mesagerie;
- fiecare client ce subscrie la un topic, primește propria-i copie a mesajelor publicate la acel topic.

Spre exemplificare, prezentăm în continuare clasa **AbstractTopicPublisher** care implementează o conexiune destinată publicării de mesaje la un *topic* ce urmează a fi specificat în constructorul clasei la momentul instanţierii acesteia.

```
public abstract class AbstractTopicPublisher
{
    private TopicSession pubSession = null;
    private TopicPublisher publisher = null;
    private TopicConnection connection = null;
    private String userID = null;

    // Constructor used to initialize AbstractTopicPublisher
    public AbstractTopicPublisher(String imqAddress, String topicName, String userID)
            throws JMSException, Exception
    {
        // Create a JMS connection factory
        String addressList = imqAddress;
        com.sun.messaging.TopicConnectionFactory topicConnectionFactory =
         new com.sun.messaging.TopicConnectionFactory();
         topicConnectionFactory.setProperty(com.sun.messaging.ConnectionConfiguration.
         imqAddressList, addressList);

        // Create a JMS connection
         TopicConnection topicConnection =
         topicConnectionFactory.createTopicConnection();
         topicConnection.setClientID(userID);

        // Create JMS session object
         TopicSession topicSession = topicConnection.createTopicSession(false,
         Session.AUTO_ACKNOWLEDGE);

        // Create a JMS topic
        Topic pubTopic = topicSession.createTopic(topicName);

        // Create a JMS publisher
        TopicPublisher topicPublisher = topicSession.createPublisher(pubTopic);

        // Intialize the AbstractTopicPublisher application variables
        this.connection = topicConnection;
        this.pubSession = topicSession;
        this.publisher = topicPublisher;
        this.userID = userID;

        // Start the JMS connection; allows messages to be delivered
        connection.start();
    }

    // Close the JMS connection
    public void close() throws JMSException
    {
        if (connection != null)
            connection.close();
    }

    public TopicSession getSession()
    {
        return this.pubSession;
    }

    public TopicPublisher getPublisher()
    {
        return this.publisher;
    }

    public String getUserID()
    {
        return this.userID;
    }
}
```

Consumatorii care subscriu la un *topic* în modelul pub/sub pot realiza această subscrie în mod durabil sau nedurabil.
Clienții JMS care subscriu nedurabil la un subiect de mesagerie vor primi mesajele publicate către acel *topic* numai dacă ascultă în mod activ la *topic*. Altfel spus, atât timp cât aplicația client JMS este conectată la platforma de comunicație, ea primește mesajele publicate către subiectul de mesagerie la care a subscris. Atunci când nu este conectat la platforma de mesagerie, mesajele publicate către subiectul altfel de interes pentru client, vor fi irecuperabil pierdute de către acesta din urmă (***Figura 11.4***).

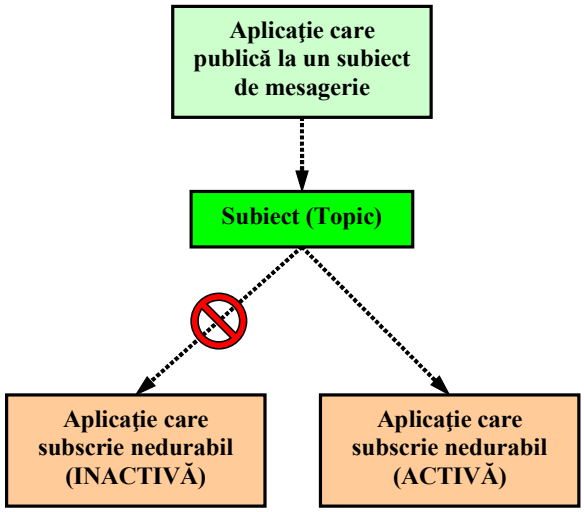

Figura 11.4 – Clienți JMS care subscriu nedurabil

Clasa **`AbstractTopicSubscriber`**, prezentată mai jos, implementează stabilirea unei conexiuni nedurabile pentru subscrierea la un *topic* de mesaje ce va fi furnizat constructorului clasei la momentul instanțierii.

```
public abstract class AbstractTopicSubscriber implements javax.jms.MessageListener
{
    private TopicConnection connection = null;
    private String userID = null;

    // Constructor used to initialize AbstractTopicSubscriber
    public AbstractTopicSubscriber(String imqAddress, String topicName, String userID)
            throws JMSException, Exception
    {
        // Create a JMS connection factory
        String addressList = imqAddress;
        com.sun.messaging.TopicConnectionFactory topicConnectionFactory =
        new com.sun.messaging.TopicConnectionFactory();
        topicConnectionFactory.setProperty(com.sun.messaging.ConnectionConfiguration.
        imqAddressList, addressList);

        // Create a JMS connection
        TopicConnection topicConnection =
        topicConnectionFactory.createTopicConnection();
        topicConnection.setClientID(userID);

        // Create JMS session object
```

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

            // Create a JMS topic
            Topic subTopic = topicSession.createTopic(topicName);

            // Create a JMS subscriber
            TopicSubscriber topicSubscriber = topicSession.createSubscriber(subTopic,
            null, true);

            // Set a JMS message listener
            topicSubscriber.setMessageListener(this);

            // Intialize the AbstractTopicSubscriber application variables
            this.connection = topicConnection;
            this.userID = userID;

            // Start the JMS connection; allows messages to be captured
            connection.start();
        }

        // Close the JMS connection
        public void close() throws JMSException
        {
            if (connection != null)
                connection.close();
        }

        public String getUserID()
        {
            return this.userID;
        }
    }
```

Pe de altă parte, clienții care subscriu durabil la un subiect de mesagerie vor primi toate mesajele publicate către acel *topic* (în funcție de selecția aplicată asupra mesajelor la care au subscris), indiferent dacă întreţin o conexiune activă sau nu. Mecanismul utilizat este numit *store-and-forward* şi prin implementarea lui serverul de mesagerie are posibilitatea de a păstra de o manieră persistentă mesajele destinate unui client care a subscris durabil la un topic, dar care nu este tot timpul conectat la server (***Figura 11.5***).

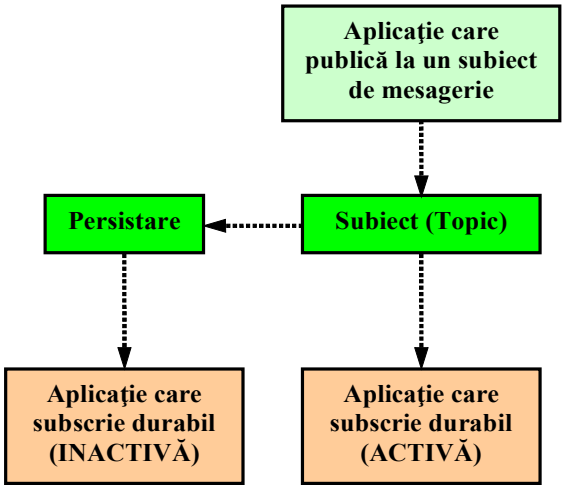

Figura 11.5 – Clienți JMS care subscriu durabil

Am văzut mai sus cum se poate realiza o subscriere nedurabilă la un *topic*, prin prezentarea clasei **`AbstractTopicSubscriber`**.

```
TopicSubscriber topicSubscriber = topicSession.createSubscriber(subTopic, null, true);
```

Dacă metoda **`createSubscriber`**, ce aparţine clasei **`Session`**, este înlocuită cu metoda **`createDurableSubscriber`**, aparţinând de asemenea clasei **`Session`**, atunci subscrierea se realizează în mod durabil. Clasa **`AbstractDurableTopicSubscriber`**, implementată în API-ul ASETS, diferă de clasa **`AbstractTopicSubscriber`**, prezentată anterior, doar prin utilizarea metodei **`createDurableSubscriber`** în locul metodei **`createSubscriber`**. Parametrul adiţional (**`userID`**) reprezintă numele asociat, la nivelul serverul de mesagerie, subscrierii durabile creată.

```
TopicSubscriber topicDurableSubscriber = topicSession.createDurableSubscriber(subTopic,
userID, null, true);
```

Ceilalţi parametri reprezintă: numele subiectului la care se face subscrierea (**`subTopic`**), filtrul care se aplică asupra mesajelor furnizate de *topic* (este un parametru de tip **`String`** care în cazul nostru are valoarea **`null`**, semnificând faptul că nu se aplică nici o filtrare) şi dacă sunt sau nu sunt ignorate mesajele plasate la subiectul specificat de către consumatorul JMS care a creat conexiunea de subscriere (parametrul de tip **`boolean`**, care în cazul nostru are valoare **`true`**, semnificând faptul că mesajele generate de clientul local sunt ignorate).

În practică, sunt numeroase situaţiile în care o subscriere durabilă la un topic este utilă, dar sunt şi multe cazuri când aceasta nu se justifică. Cu toate că cerinţele concrete ale activităţii modelate de un sistem informatic dictează în general ce modalitate de subscriere să se adopte, sunt şi alte considerente care trebuie avute în vedere, cum ar fi volatilitatea datelor şi capacitate de stocare cerută pentru persistarea acestora. De exemplu, dacă un client subscrie la un topic la care se publică la fiecare câteva secunde (sau chiar mai frecvent) preţuri ale acţiunilor tranzacţionate la o instituţie bursieră, atunci probabil că o subscriere durabilă nu este cea mai potrivită alegere. În primul rând, în afară de situaţia în care datele sunt folosite pentru o analiză a tendinţei, interesul este în a cunoaşte preţul curent al unei acţiuni şi nu preţul din urmă cu 15 minute, de exemplu. În al doilea rând, dacă un client care a subscris în mod durabil nu este activ pentru o lungă perioadă timp, se pot acumula mii de mesaje, până la urmă inutile pentru clientul în cauză, mesaje care nu fac decât să consume în mod nejustificat spaţiul de stocare persistentă al serviciului de mesagerie.

Unii furnizori de sisteme de mesagerie oferă posibilitatea definirii de subscrieri durabile prin intermediul unor fişiere de configurare, sau a unei interfeţe cu rol de administrare a sistemului. În această situaţie, se spune despre aceste subscrieri că sunt subscrieri durabile administrate, desemnând faptul că ele sunt definite în mod static şi cunoscute de către serverul JMS la momentul lansării lui în execuţie.

Pe de altă parte, specificaţiile JMS permit definirea de subscrieri durabile în mod dinamic, în timpul execuţiei serverului de mesagerie.

Depinde de rolul jucat de fiecare client, care subscrie în mod durabil la un sistem orientat pe mesaje, dacă această subscriere este util a fi definită în mod static sau dinamic. Atunci când consumatorul este de aşteptat să fie interesat în mod neîntrerupt şi pe o perioadă

indefinită de mesajele publicate la anume topic (de exemplu, capturarea tuturor execuţiilor primite de la bursă de către un sistem de gestiune a tranzacţiilor), atunci subscrierea durabilă poate fi definită în mod static.
Dacă dimpotrivă, consumatorul este interesat de mesajele publicate la topic pe o perioadă de precis determinată, chiar dacă subscrie durabil, pentru că doreşte să nu piardă nici unul din mesajele publicate în acea perioadă, este de preferat să subscrie durabil în mod dinamic, având ulterior posibilitate de a renunţa la subscriere, de asemenea în mod dinamic, prin invocarea metodei **`Session.unsubscribe()`**.

11.3 Garantarea livrării mesajelor

Componenta de garantare a livrării mesajelor în cadrul unei platforme JMS conţine trei părţi esenţiale:

- autonomia mesajului JMS;
- mecanismul de *store-and-forward*;
- semantica asociată confirmării mesajelor într-o manieră care este transparentă programatorului de aplicaţii.

11.3.1 Autonomia mesajului JMS

Mesajele JMS sunt entităţi autonome şi autosuficiente. Un mesaj poate fi trimis şi retrimis de mai multe ori între multipli clienţi JMS. Fiecare client, va consuma mesajul, îl va examina şi îl va prelucra în conformitate cu rolul pe care îl joacă în ansamblul sistemului. Prelucrarea poate să implice modificarea mesajului şi transmiterea lui către un alt client, sau crearea unui nou mesaj pe baza datelor conţinute în mesajul iniţial. Într-un anume sens, fiecare client JMS are un contract încheiat cu restul sistemului privind prestarea unui anumit serviciu, sau set de servicii, asupra fiecărui mesaj care îi este destinat. Când serviciul prestat asupra mesajului primit este încheiat, el poate transmite mesajul mai departe sau poate crea un nou mesaj pe care îl depune într-o coadă, sau îl publică la un *topic*. Sistemul de mesagerie garantează livrarea în continuare a mesajului către un alt client JMS. Odată ce mesajul este preluat de serverul de mesagerie, el este considerat ca şi livrat către destinaţie.

11.3.2 Mecanismul store-and-forward

Atunci când un mesaj este marcat ca fiind persistent, devine responsabilitatea serverului de mesagerie de a asigura utilizarea unui mecanism de tip *store-and-forward* pentru a-şi îndeplini contractul pe care îl are încheiat cu clientul în ceea ce priveşte garantarea livrării mesajului. Mecanismul de stocare implică salvarea mesajului pe disc (sau pe un alt mediu care garantează persistenţa), pentru a asigura posibilitatea recuperării mesajului, în cazul în care serverul de mesagerie ar eşua în execuţie, sau consumarea mesajului nu s-ar realiza cu succes la destinaţie. Mesajul poate fi stocat de un depozitar central (cum este cazul arhitecturilor centralizate), sau la nivel local, pentru fiecare client care produce şi/sau consumă mesaje în cadrul sistemului (soluţie utilizată în cazul arhitecturilor descentralizate). Mecanismul de stocare a mesajelor poate să folosească fie fişiere pe disc, fie baze de date, fie o combinaţie optimală a ambelor variante.

11.3.3 Confirmarea mesajelor și condiții de eșec

Interfața JMS specifică un set de moduri de confirmare a mesajelor. Aceste moduri de confirmare reprezintă elementul cheie, în ceea ce privește mecanismului de garantare a livrării mesajelor, de către un sistem de mesagerie. Confirmarea unui mesaj este parte a protocolului încheiat între componenta de interfață JMS a clientului și serverul de mesagerie. Serverul confirmă primirea mesajelor de la producătorii JMS și consumatorii JMS confirmă primirea mesajelor de la serverul de mesagerie. Protocolul de confirmare îi permite furnizorului de mesagerie să monitorizeze progresul realizat în transmiterea mesajului, știind în acest fel dacă mesajul produs a fost sau nu consumat cu succes. Deținând aceste informații, furnizorul de mesagerie poate asigura gestiunea distribuirii mesajelor și garantarea livrării acestora.

11.4 Modul de transmisie cu autoconfirmare

În capitolul anterior am arătat că există trei moduri de confirmare a mesajelor de către un consumator JMS:

- **`AUTO_ACKNOWLEDGE`**
- **`DUPS_OK_ACKNOWLEDGE`**
- **`CLIENT_ACKNOWLEDGE`**

Am insistat acolo asupra modului de confirmare explicită a mesajelor din partea clientului și asupra oportunității lucrului cu mesaje duplicate. Vom analiza aici modul de confirmare **`AUTO_ACKNOWLEDGE`**, din perspectiva producătorului mesajului, din cea a serverului de mesagerie, precum și din perspectiva consumatorului mesajului.

11.4.1 Perspectiva producătorului de mesaje

Așa cum am arătat mai înainte, metode ca **`TopicPublisher.publish()`** sau **`QueueSender.send()`**, în mod transparent pentru programatorul de aplicații, funcționează de fapt în manieră sincronă. Aceste metode sunt folosite pentru trimiterea mesajelor către serverul de mesagerie și această trimitere se realizează cu blocarea execuției producătorului până la primirea confirmării de primire din parte serverului JMS. Imediat ce confirmarea este primită, firul de execuție din care a fost lansată trimiterea preia controlul asupra fluxului de procesare al producătorului mesajului. Această confirmare nu este însă vizibilă la nivelul modelului de programare a clientului JMS.
Cu toate acestea, dacă se înregistrează un eșec pentru această transmisie către server, interfața JMS generează o excepție și mesajul este considerat ca nelivrat. La nivelul interfeței de programare a clientului JMS, pasul prin care mesajul este întâi livrat către serverul de mesagerie (pentru ca apoi să fie transmis de acesta către consumator) este transparent.

11.4.2 Perspectiva serverului de mesagerie

Confirmarea trimisă în mod transparent de server către producătorul mesajului înseamnă că serverul a primit mesajul şi din acel moment a acceptat să îşi asume responsabilitatea cu privire la livrarea lui către destinaţie. Din perspectiva serverului JMS, confirmarea trimisă producătorului nu este condiţionată în mod direct de livrarea mesajului. Pentru el sunt doi paşi, la nivel logic. În realitate însă, cele două operaţii se pot realiza în paralel, dar aceasta depinde de implementarea aleasă de furnizorul de mesagerie.
Trebuie să precizăm, în acest context, că diagramele pe care le vom prezenta în continuare descriu interacţiunea la nivel logic dintre componentele unui sistem de mesagerie şi nu reflectă în mod necesar arhitectura de procesare specifică vreunui furnizor de mesagerie în particular.
În consecinţă, pentru mesajele persistente, serverul scrie mai întâi mesajul pe disc (partea de stocare din mecanismul *store-and-forward*) şi apoi trimite confirmarea către producător, cu privire la primirea mesajului (***Figura 11.6***).

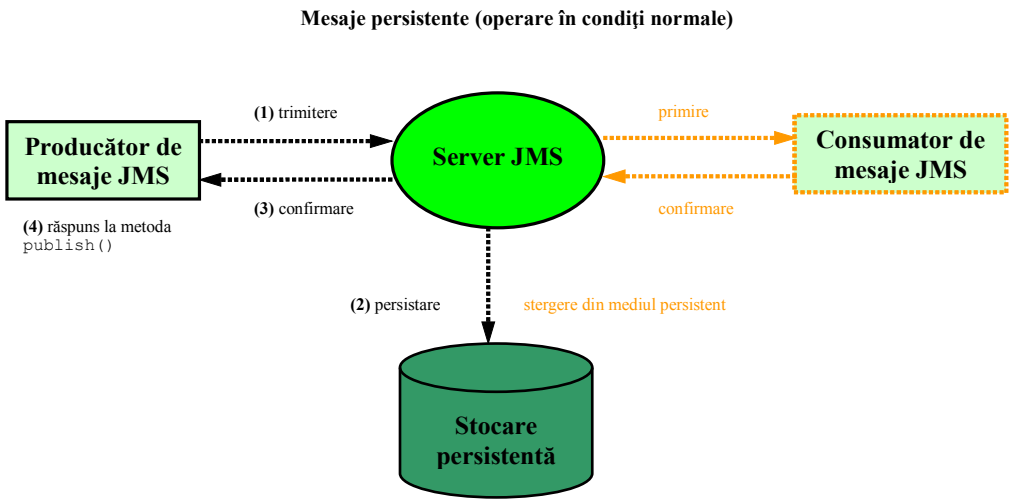

Figura 11.6 – Primirea şi trimiterea mesajelor sunt operaţiuni separate

În cazul mesajelor nepersistente, aceasta înseamnă că serverul poate trimite confirmarea de primire către producător imediat ce a primit mesajul şi îl are în memorie. Dacă nu există vreun consumator care să fi subscris la subiectul pentru care a fost publicat mesajul (dacă este vorba de modelul de comunicaţie *pub/sub*), atunci serverul poate să se dispenseze de mesaj, depinzând de implementarea aleasă de furnizorul de mesagerie. În modelul *pub/sub* serverul livrează o copie a mesajului către fiecare client care a subscris la subiectul respectiv. În cazul unor subscrieri durabile, serverul de mesagerie nu consideră că mesajul este distribuit în totalitate până când nu primeşte confirmarea de la toţi destinatarii avuţi în vedere. Serverul de mesagerie gestionează confirmările pentru fiecare consumator în parte şi poate şti în orice moment care client a primit mesajul şi care nu. Odată ce serverul a livrat mesajul către toţi consumatorii de care are cunoştinţă că au subscris la subiectul în cauză şi a recepţionat confirmarea de primire de la fiecare în parte, atunci (şi numai atunci) el poate să şteargă mesajul din mediul de stocare persistentă (***Figura 11.7***).

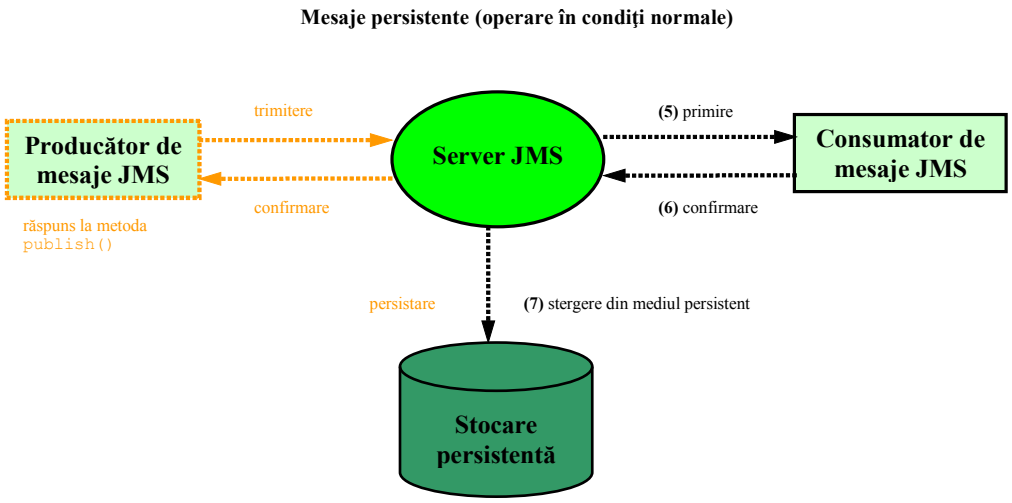

Figura 11.7 – Mesajul este șters din mediul de stocare persistentă atunci când serverul a primit confirmarea de la ultima aplicație cunoscută drept consumator al acelui mesaj

Dacă subscrierile sunt durabile și clienții care au subscris nu sunt momentan conectați la sistem, atunci mesajul va fi păstrat de server până când fie consumatorii devin activi, fie mesajul expiră (în cazul în care acesta are specificat un timp de expirare). Acest scenariu este valabil chiar și pentru mesajele marcate ca nepersistente. Dacă un mesaj nepersistent trebuie să fie livrat către un client care a subscris durabil și care este momentan deconectat, atunci serverul de mesagerie salvează mesajul pe disc, ca și cum ar fi fost un mesaj persistent. În această situație, diferența dintre un mesaj persistent și unul nepersistent este una subtilă, însă foarte importantă. Pentru mesajele nepersistente, poate exista o fereastră de timp, de la momentul la care serverul confirmă primirea mesajului către producător, până la momentul la care scrie mesajul pe disc pentru stocarea acestuia, cerută de subscrierea durabilă a unui client momentan inactiv (***Figura 11.8***).

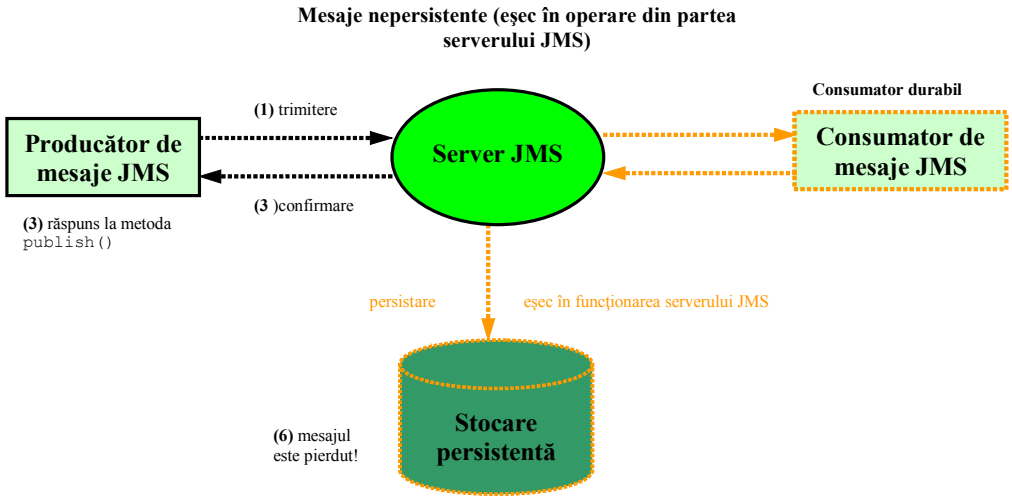

Figura 11.8 – Mesajele nepersistente pentru un consumator durabil pot fi pierdute

Dacă serverul JMS eșuează în funcționare în timpul acestei ferestre de timp, atunci mesajul poate fi pierdut. Dacă mesajul este persistent, el este salvat pe disc înainte ca serverul să confirme primirea acestuia către producător. În aceste condiții, chiar dacă serverul ar eșua, la repornire el va putea să recupereze mesajele persistente care au fost confirmate către producători (***Figura 11.9***).

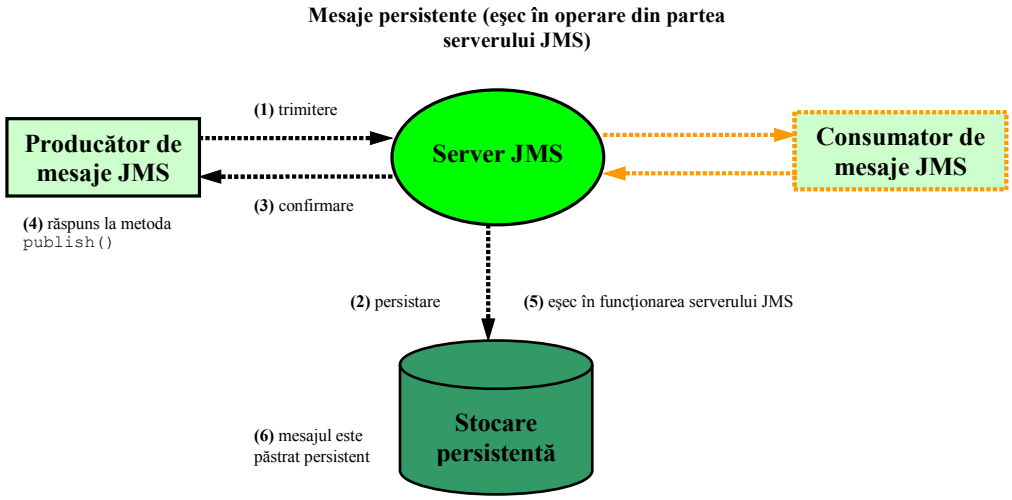

Figura 11.9 – Mesajele persistente nu sunt pierdute în cazul unui eșec din partea serverului JMS

De asemenea, din moment ce mesajele sunt salvate pe un mediu persistent, ele nu sunt pierdute și vor putea fi livrate către consumatori, atunci când serverul de mesagerie este repornit (***Figura 11.10***).

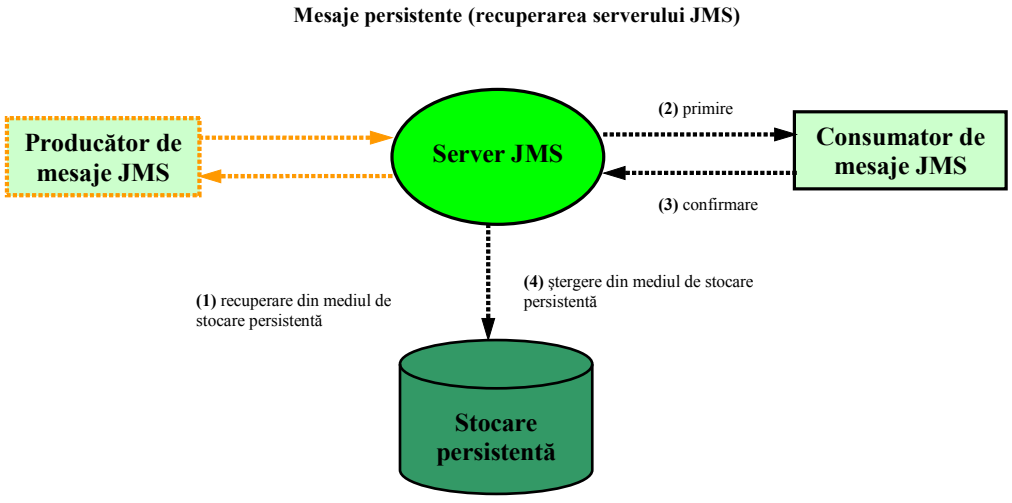

Figura 11.10 – Mesajele persistente sunt trimise către consumator când serverul JMS își reface starea

În cazul modelului de comunicație *p2p*, livrarea mesajele persistente este garantată de platforma de mesagerie. Dacă mesajele persistente sunt transmise prin mecanismele modelului *pub/sub* atunci garantarea livrării lor este asigurată numai către consumatorii

care au subscris durabil. Pentru consumatorii care au subscris nedurabil, comportamentul serverului JMS cu privire la livrarea mesajelor persistente poate varia de la un furnizor de mesagerie la altul.

11.4.3 Perspectiva consumatorului de mesaje

Există de asemenea reguli care guvernează confirmarea primirii şi condiţiile în care preluarea mesajelor poate eşua din perspectiva consumatorului. Dacă sesiunea de transmisie a mesajelor este realizată în modul **`AUTO_ACKNOWLEDGE`**, atunci componenta JMS a clientului va trimite o confirmare către serverul de mesagerie, pe măsură ce fiecare mesaj este consumat de client. Dacă serverul nu primeşte această confirmare, atunci el consideră că mesajul nu a fost livrat către consumator şi va încerca să-l retransmită.
În cazul unui consumator cu subscriere nedurabilă la un *topic*, dacă serverul eşuează în timpul livrării mesajului, atunci acesta poate fi pierdut. Dacă însă consumatorul subscrie durabil şi eşuează confirmarea primirii mesajului, trimisă către server (***Figura 11.11***), atunci serverul JMS consideră mesajul nelivrat şi va încerca să-l retransmită consumatorului (***Figura 11.12***).

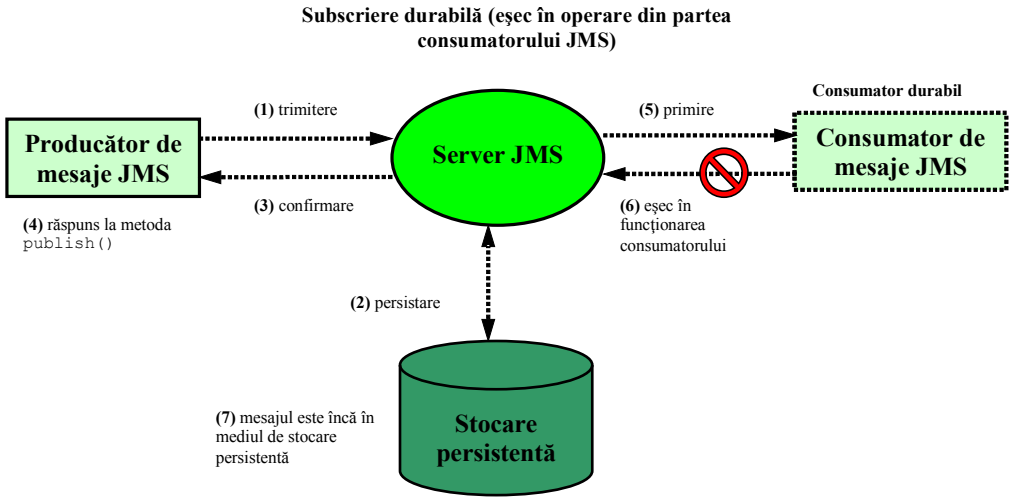

Figura 11.11 – Eşuarea livrării unui mesaj pentru un consumator JMS cu subscriere durabilă

În acest caz, cerinţa JMS de a trimite un mesaj către un consumator „o dată şi numai o dată" este pusă sub semnul întrebării. Consumatorul poate primi mesajul din nou pentru că, atunci când livrarea acestuia este garantată, este mai bine să se rişte livrarea mesajului de două ori, decât să se rişte pierderea mesajului cu totul. Mesajul retransmis va fi avea setată eticheta **`JMSRedelivered`**. Aplicaţia client consumatoare poate verificare această setare prin apelul metodei **`getJMSRedelivered`**, ce face parte din interfaţa **`Message`**. Numai mesajul cel mai recent primit este susceptibil de această ambiguitate. Pentru a se proteja de mesaje duplicate, în timp ce funcţionează în modul **`AUTO_ACKNOWLEDGE`**, o aplicaţie care consumă mesaje trebuie să verifice dacă mesajul retransmis a fost deja procesat sau nu. Una din tehnicile cele mai folosite pentru a verifica acest lucru este de a utiliza o baza de date în care se păstrează valoare **`JMSMessageID`** din antetul mesajul ca

și cheie de acces. **`JMSMessageID`** este unic pentru toate mesajele vehiculate în sistem și poate fi folosit pentru a monitoriza întreaga istorie a mesajelor.

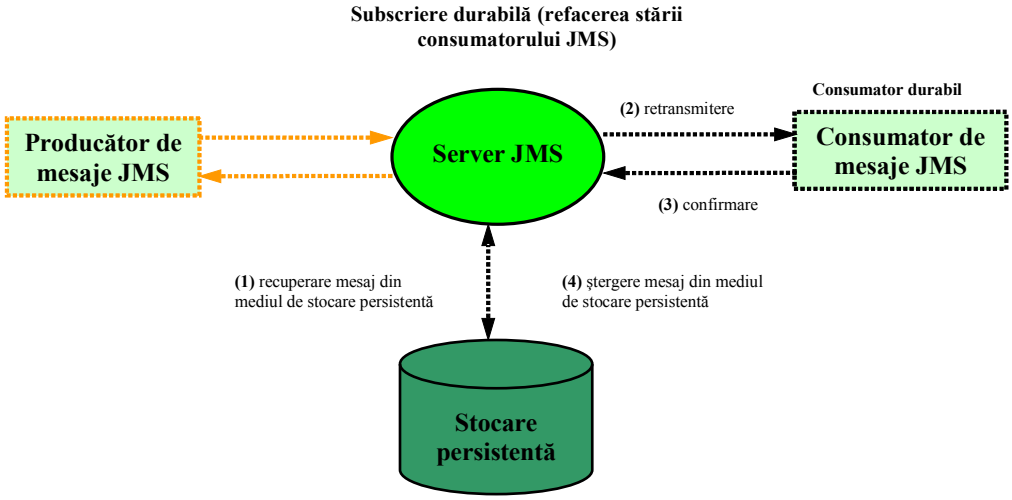

Figura 11.12 – Refacerea stării unui consumator JMS cu subscriere durabilă

În cazul utilizării cozilor, în modelul modelul *p2p*, mesajele sunt marcate de către producător ca fiind persistente sau nepersistente. Dacă sunt persistente, atunci ele sunt scrise pe disc și sunt manipulate în aceeași manieră de către serverul de mesagerie, ca în cazul mesajelor persistente în cadrul modelul *pub/sub* (reguli de confirmare, condiții de eșec, modalitate de recuperare).
Din perspectiva consumatorului, regulile sunt cumva mai simple, din moment ce numai un singur consumator poate primi o instanță dată a unui mesaj. Mesajul este reținut în coadă până când este livrat către consumator sau până expiră. Este o situație analogă celei descrise pentru un consumator care subscrie durabil la un topic. Consumatorul poate fi deconectat de la platformă în timp ce sunt produse mesajele, fără însă a le pierde. Odată reconectat, el va primi toate mesajele produse în timpul perioadei de inactivitate. Dacă mesajele sunt nepersistente, atunci nu există nici o garanție că ele vor supraviețui în cazul eșuării serverului de mesagerie.

12 Gestiunea ordinelor și a portofoliilor într-o abordare SOA

Am văzut în capitolul precedent care sunt premisele care stau la baza comunicării într-un sistem informatic fundamentat pe un *middleware* orientat pe mesaje. Vom încerca în cele ce urmează să identificăm caracteristicile funcționale ale unui *Order Management Server* (OMS) și ale unui *Portfolio Management Server* (PMS) în contextul unei arhitecturi orientată pe servicii, care va fi susținută de un *middleware* orientat pe mesaje.
În condițiile în care, în proiectarea unei astfel de arhitecturi de *trading*, accentul nu mai cade pe manieră în care aplicațiile sistemului pot comunica unele cu altele, plecându-se de la premisele prezentate anterior pentru un suport de comunicație, abordarea se ridică la un nivel superior de abstractizare, punându-ne mai curând problema ce servicii își pot

oferi una alteia aplicaţiile componente ale sistemului. Vom analiza în continuare cerinţele funcţionale alte OMS şi PMS, privite în special din perspectiva unui sistem de *trading* într-un mediu de simulare.

12.1 Cerinţele funcţionale ale unui OMS

Un server pentru gestiunea ordinelor investitorilor presupune, din punct de vedere funcţional, realizarea următoarelor activităţi:

- întreţinerea conexiunilor cu clienţii, în ceea ce priveşte controlul accesului la activitatea de *trading*, în funcţie de modalităţile oferite pentru conectare şi deconectare;
- preluarea cererilor pentru plasarea de ordine noi, pentru modificarea şi anularea ordinelor plasate anterior;
- generarea identificatorului unic asociat fiecărui ordin intrat în sistem şi informarea clientului cu privire la identificatorul ordinului plasat de acesta; prin acest identificator, atât OMS cât şi clientul vor putea monitoriza situaţia ordinului pe toată durata unei sesiuni de *trading*;
- salvarea pe disc a ordinelor clienţilor şi actualizarea stării acestor ordine atunci când sunt înregistrate evenimente care le privesc;
- trimiterea ordinelor clienţilor către instituţia bursieră avută în vedere, respectiv către simulatorul de bursă, în cadrul mediului de simulare;
- capturarea informaţiilor privind starea ordinelor trimise la bursă;
- înştiinţarea clienţilor cu privire la modificările de stare ale ordinelor plasate;
- capturarea execuţiilor generate de instituţia bursieră, aferente ordinelor care au îndeplinit condiţiile impuse de motorul de împerechere pentru a fi puse în corespondenţă;
- salvarea pe disc a execuţiilor primite de la bursă, respectiv de la simulator;
- înştiinţarea clienţilor cu privire la rezultatele activităţii de *trading* primite de la bursă (diseminarea execuţiilor către clienţi);
- satisfacerea cererilor explicite ale clienţilor în ceea ce priveşte ordinele plasate de aceştia în timpul sesiunii de tranzacţionare;
- satisfacerea cererilor explicite ale clienţilor în ceea ce priveşte rezultatele activităţii de *trading*, respectiv furnizarea către aceştia a execuţiilor asociate ordinelor proprii, care au fost împerecheate cu succes (executate) la bursă;
- satisfacerea cererilor clienţilor cu privire la instrumentele financiare tranzacţionabile.

Putem observa posibilitatea modelării cu relativă uşurinţă a tuturor acestor activităţi ca servicii oferite de OMS către clienţii conectaţi la sistemul de tranzacţionare bursieră.
Serverul de gestiune a ordinelor este principalul furnizor de servicii al unui sistem de *trading*, cel puţin din perspectiva clienţilor, care îl percep ca prima poartă de intrare în sistem. Toate activităţile menţionate anterior sunt necesar a fi realizate de o manieră eficientă şi agilă. Această însemnând că OMS trebuie să fie întotdeauna disponibil de prelua imediat diferitele cereri venite din parte clienţilor săi. Menirea lui este aceea de a răspunde cât mai rapid acestor cereri (confirma primirea lor) şi de a delega mai departe procesul de satisfacere concretă a lor.

Menționăm faptul că multe din cererile de servicii lansate de clienți ajung la OMS și sunt procesate de acesta în paralel. Aceasta fiind o facilitate oferită în mod natural de platforma de comunicație orientată pe mesaje. Cu toate acestea, OMS trebuie până la urmă să realizeze satisfacerea fiecărei cereri conform principiul priorității temporale. Ordinele clienților în particular, necesită prelucrarea lor în ordinea strictă în care au fost primite de către OMS. Această cerință impune organizarea prelucrării datelor în interiorul serverului după principiul „primul venit, primul servit" și utilizarea unor mecanisme de tip coadă, acolo unde este necesară serializarea evenimentelor.
Din perspectivă procedurală, activitățile descrise mai sus pot fi transpuse într-un algoritm generic pe care îl prezentăm în continuare.

Procedura de inițializare OMS
- **preluare lista instrumente financiare tranzacționabile ← BD**
- **preluare ordine plasate în sesiunea curentă ← BD (dacă există)**
- **preluare `lastOrderID` generat ← BD (în cazul primei rulări va fi zero)**
- **setare `newOrderID` = `lastOrderID` + 1**
- **preluare execuții capturate în sesiunea curentă (dacă există)**

while (`sesiunea de trading este deschisă`)

Procesează cerere `Log In` client
- **verificare `userID` în BD**
- **actualizare stare client în BD (ultima conectare)**
- **trimitere răspuns către client**

Procesează cerere `Log Out` client
- **verificare `userID`**
- **trimitere răspuns către client**

Procesează `ADD`, `CANCEL`, `UPDATE` ordin client
- **generare `orderID` pentru un ordin nou**
- **inserare ordin nou, sau modificare ordin existent în BD**
- **confirmare primire către client**
- **trimitere a ordinului către bursă**

Disemnează stare ordine clienți capturate de la bursă
- **preluare actualizare ordin de la bursă**
- **modificare stare ordin în BD**
- **propagare actualizare ordin către client**

Disemnează execuții clienți capturate de la bursă
- **preluare execuție de la bursă**
- **inserare execuție în BD**
- **propagare execuție către client**

Procesează cerere client privind lista instrumentelor financiare tranzacționabile

Procesează cerere client privind ordinele plasate în cursul sesiunii de trading curente

Procesează cerere client privind execuțiile primite în cursul sesiunii de trading curente

În procedura de inițializare a OMS este necesară luarea în considerare atât a unei lansări în execuție realizată la începutul zilei de *trading* (atunci când nu sunt nici un fel de ordine și execuții înregistrate pentru sesiunea curentă), cât și situația unei reporniri în timpul sesiunii de tranzacționare.

12.2 Cerințele funcționale ale unui PMS

Serverul dedicat gestiunii portofoliilor clienților, cel puțin în cadrul mediul de simulare, trebuie să ofere, din perspectivă funcțională, realizarea următoarelor activități:

- capturarea execuțiilor asociate ordinelor clienților și diseminate în sistem de către OMS;
- generarea identificatorului unic pentru tranzacția asociată execuției primite și procesate;
- crearea unui obiect de tip **`Trade`** și popularea câmpurilor acestuia cu datele primite la nivel de execuție, la care se adaugă identificatorul generat în prealabil (**`tradeID`**);
- salvarea tranzacției în baza de date;
- actualizarea portofoliului clientului pe baza noii tranzacții create;
- furnizarea tuturor tranzacțiilor realizate de un client, la cererea explicită a acestuia;
- furnizarea portofoliului unui client, la cererea explicită a acestuia.

Întrucât PMS are nevoie de toate execuțiile asociate ordinelor plasate de clienții sistemului de *trading*, el nu își permite pierderea nici unei execuții diseminate în timpul sesiunii de tranzacționare, fie că el este activ sau nu. De aceea, el va subscrie în mod durabil la subiectul de mesagerie la care sunt publicate aceste execuții de către OMS.
Atunci când o nouă tranzacție este generată de PMS, aceasta este în primul rând salvată în baza de date, iar apoi se trece la actualizarea portofoliului clientului pe baza datelor din noua tranzacție. Dacă instrumentul financiar conținut în noua tranzacție este deja deținut de către client în portofoliul său, atunci PMS va actualiza cantitatea deținută. Dacă tranzacția este una de cumpărare, atunci noua cantitate tranzacționată va fi adăugată la cea existentă în portofoliu. Dacă noua tranzacție este una de vânzare, atunci cantitatea vândută va fi scăzută din cantitatea deținută de client în portofoliul său. Dacă prin noua tranzacție de vânzare, clientul a vândut toată cantitatea deținută în portofoliu din instrumentului financiar în cauză, atunci se poate șterge definitiv înregistrarea din portofoliu corespunzătoare respectivului instrument financiar.
În practică însă, este de preferat să se marcheze instrumentul financiar respectiv ca fiind vândut din portofoliu, dar să se păstreze înregistrarea în baza de date pentru eventuale referințe ulterioare. În general, este de preferat această modalitate de lucru cu baza de date. Cel puțin imediat, este indicat ca înregistrările să nu fie șterse definitiv. Se poate reveni mai târziu, utilizându-se eventual un proces care rulează periodic, pentru ștergerea sau mutarea într-o arhivă a înregistrărilor marcate ca ne mai fiind de interes.
În cazul în care a fost mărită cantitatea deținută deja de client dintr-un anumit instrument financiar (tranzacție de cumpărare), sau dacă nu s-a vândut toată cantitatea deținută (tranzacție de vânzare), atunci este necesar să calculeze un nou preț mediu, asociat cantității rămase din instrumentul respectiv în portofoliu. Acesta va fi un preț mediu

ponderat, calculat pe baza prețului și a cantității existente în portofoliu, respectiv a prețului și a cantității oferite de noua tranzacție.
Din perspectivă procedurală, activitățile realizate de serverul de gestiune a portofoliilor pot fi transpuse în următorul algoritm generic.

Procedura de inițializare PMS
- **preluare listă instrumente financiare tranzacționabile ← BD**
- **preluare `lastTradeID` generat ← BD (în cazul primei rulări va fi zero)**
- **setare `newTradeID` = `lastTradeID` + 1**

while (`sesiunea de trading este deschisă`)

Procesare execuție client, diseminată de OMS
- **generare tradeID pentru o nouă tranzacție**
- **crearea unui nou obiect de tip `Trade`**
- **salvare tranzacție nouă în BD**

Actualizează portofoliu client în BD

if (`simbolul există deja în portofoliu`)

if (`tranzacție de cumpărare`)

$$Q_p = Q_p + Q_t$$

$$P_p = \frac{P_p Q_p + P_t Q_t}{Q_p + Q_t}$$

else

$$Q_p = Q_p - Q_t$$

$$P_p = \frac{P_p Q_p - P_t Q_t}{Q_p - Q_t}$$

actualizare înregistrare portofoliu în BD

else

inserare înregistrare portofoliu în BD

Procesare cerere client cu privire la furnizarea tranzacțiilor realizate

Procesare cerere client privitoare la situația curentă a portofoliului

Q_p și P_p reprezintă cantitatea, respectiv prețul instrumentului financiar deținut de client în portofoliu până la procesarea ultimei tranzacții. Q_t și P_t reprezintă cantitatea și prețul conținute în tranzacția care urmează să se actualizeze portofoliul clientului. În cazul unei tranzacții de vânzare, se va verifica dacă nu cumva se vinde mai mult decât se deține. Evident, o astfel de verificare este de presupus că a fi fost în prealabil făcută la nivelul interfeței grafice de *trading*.