Message-Oriented-Middleware in a Distributed Environment

Sushant Goel[1], Hema Sharda[1], and David Taniar[2]

[1] School of Electrical and Computer systems Engineering,
Royal Melbourne Institute of Technology, Australia
hema.sharda@rmit.edu.au
[2] School of Business Systems, Monash University, Australia
David.Taniar@infotech.monash.edu.au

Abstract. Middleware technologies have been facilitating the communication between the distributed applications. Traditional messaging system's are synchronous and have inherent weaknesses like – limited client connections, poor performance due to lack of resource pooling, no store-and-forward mechanism or load balancing, lack of guaranteed messaging and security as well as static client and server's location dependent code. These weaknesses and increasing e-business requirements for the distributed systems motivated us to undertake this research. This paper proposes an asynchronous communication architecture – Transfer of Messages in Distributed Systems. The advantage of the proposed architecture is that the sender of the message can continue processing after sending the message and need not wait for the reply from other application.

1 Introduction

Type of applications in the computing world has evolved rapidly from stand-alone architecture to mainframe architecture to two-tier client/server or three-tier (multi-tier) client/server architecture [6]. As the applications are becoming distributed, problems of information management on a large and distributed scale have become highly apparent.

This paper aims to enhance the features of messaging-architecture in distributed environment. Middleware is important in providing communication across heterogeneous platforms. Middleware technologies also plays important role in paradigm shift from mainframe to client/server architecture. Middleware is connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network [3].

Till now the industry had different commercial product for communication between components, but Java Message Service (JMS) is an effort towards standardization of the communication protocol. *JMS is an API provided by Sun Microsystems,* which is being supported by most of the messaging vendors. The *Transfer of Messages in Distributed Systems* (TMDS) architecture proposed in this paper, implements JMS specifications and enhances the features of existing

T. Böhme, G. Heyer, H. Unger (Eds.): IICS 2003, LNCS 2877, pp. 93-103, 2003.

middleware technologies. This is a step forward towards paradigm shift from synchronous to asynchronous messaging.

The basic aim of this paper to present a standard communication protocol in distributed environment and let the distributed components communicate in a more effective way. The proposed TMDS architecture puts together the benefits of synchronous and asynchronous communication (JMS). JMS was released in late 1998, and doesn't support XML, this paper enhances the feature of JMS and enable applications to communicate via XML message format.

The rest of this paper is organized as follows. Section 2 describes the background including messaging systems, middleware, JMS, and domain of messaging. Section 3 presents our proposed architecture. Section 4 presents a case study using our proposed architecture. Finally, Section 5 gives the conclusions and explains future work.

2 Background

We discuss various concepts of Message Oriented Middleware (MOM) in this section, including various domains of messaging like publish/subscribe and point-to-point messaging. Message is the package of business data, which contains the actual load and all the necessary routing information to travel in the network for delivery. Till late 90's there were couple of companies, who used to provide asynchronous communications between distributed components, like IBM's MQ Series.

2.1 Messaging and Message-Oriented-Middleware

Messaging is a peer-to-peer communication between software applications [5]. All the messaging clients are connected via an external agent and can send messages to any other client as well as can receive messages from any other messaging client [2]. The agent provides the way to communicate between the clients; it provides the facilities for creating, sending, receiving and reading the messages. Distributed applications can communicate in two ways:

In *synchronous communication* the application sends any message to another application and waits for the reply, the communication is typically known as synchronous communication. No further action could be done by the application till it receives the reply. But in *asynchronous communication* the application sends the message and continues processing without waiting for reply from another application [8].

Middleware in general could be defined as software that is designed for building large scale distributed systems [5]. Middleware is connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network [2,5].

To open up from the tightly synchronized hardware *Messaging-Oriented-Middleware (MOM)* provides the reliable data delivery mechanisms. MOM also lets the systems to be loosely coupled - not always operating at the same speed,

sometimes disconnected, and not having the recipient synchronously locked until the communication has completed [10].

If any new client has to be added in the messaging infrastructure then in highly coupled system the new client should know the location of all the existing clients [1], but in the MOM architecture the new client has to connect only to the middleware. If there are N numbers of client, then for a new client to be added in a tightly coupled architecture it must add N number of new connections but in MOM architecture only one new connection is added irrespective of the number of existing clients.

2.2 Java Message Service (JMS)

The JMS API provides the way to decouple the clients. JMS is a specification, which contains interfaces and abstract classes in itself needed by the messaging clients while communicating with messaging systems. If any of the components is down it does not hinder working of the system as a whole. JMS supports two major domains of messaging [8]:

Publish and subscribe domain of messaging (Pub/sub) is used when a group of users are to be informed about a particular event. The destination of the message is not the application component but the messages are delivered to the virtual destination called '*topic'* [8]. This model allows the publisher or message-producer to broadcast the message to one or more subscribers. *Point-to-Point* domain of messaging Point-to-point messaging domain communicates between clients using the concept of '*queue*','*sender*' and '*receiver*' **.** *Sender* sends all the messages addressed to a specific `queue`. All the messages are kept in the queue until the receiver fetches them or the message expires. There can be multiple senders to the queue but only single receiver.

JMS provides standard API that java developers can use to access the common features of the enterprise message systems. The design aim of JMS is to provide consistent set of interfaces that messaging clients can use independent of the underlying message system provider. The basic component of the JMS architecture is a message [7,8].

Major components to build up the application are:

Administered objects: JMS destinations and connection factories are maintained administratively and not programmatically. The messaging client lookup these administered object using JNDI API.

***Connection Factories*:** Connection factory encapsulates set of connection configuration parameters defined by the administrator. This object is used to create the connection.

Destination*:* A destination is the object a client uses to specify the target of messages it produces and the source of message it consumes.

Connections*:* Connection object provides resource allocation and management. Connections are created by the connection-factories and encapsulate a virtual connection with a JMS provider.

Sessions*:* Sessions are objects, which provide context for producing and consuming messages. Session creates the message producers, message consumers and message itself.

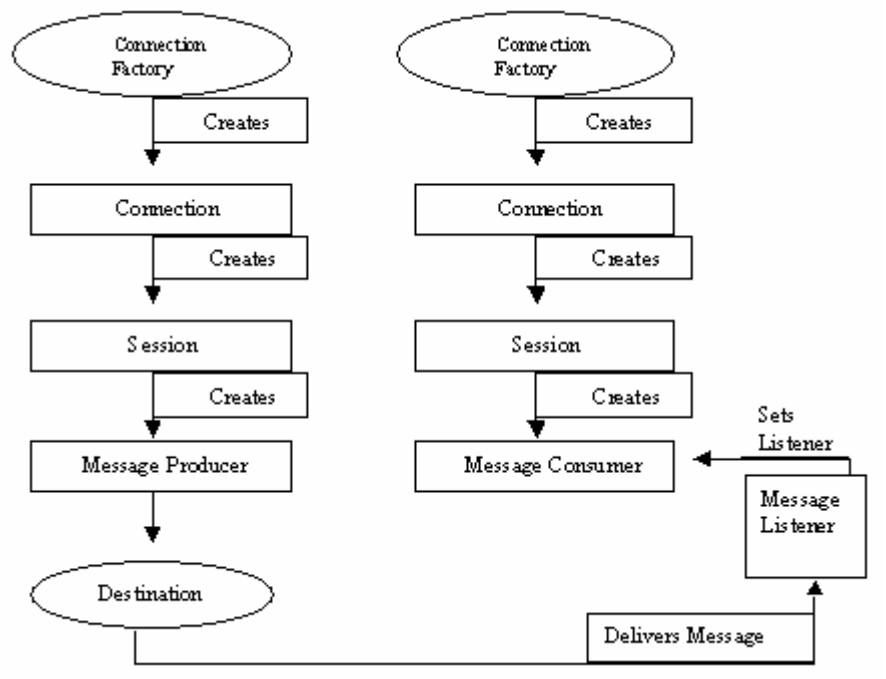

Fig. 1. Architecture of JMS application [8]

Message Producers: Session object creates message producers, to send or publish the message to the destination.

Message Consumers: Session object also creates message consumers, to receive the messages sent to a destination. The message consumer registers the interest in a destination with a JMS provider.

Messages: Message is the most important part of the messaging system, which is the point of communication between applications. The purpose of JMS application is to produce and consume messages.

JMS defines five types of message body [7]: *(i) Text Message (ii) Object Message (iii) Bytes Message (iv) Map Message (v) Stream Message*

The `Message-Listener` interface has a method called `onMessage()` The JMS provider invokes this method automatically when the message arrives. This is called as asynchronous message delivery.

3 Transfer of Messages in Distributed Systems

This section proposes the Transfer of Messages in Distributed Systems (TMDS) architecture. The proposed TMDS architecture is based on the extension of specifications provided by JMS. The application should be able to handle the messages not only when the consumer of the message is disconnected, but the

application should also be able to provide the acknowledgment of the received message, TMDS architecture takes care of these issues by implementing durable subscribers and different acknowledgment modes. Few applications cannot afford to have messages re-delivered, TMDS architecture takes care of this situation, by implementing the once-and-only once delivery mechanism.

TMDS architecture supports XML message formats. The benefit of using the XML message format is that, the industry has a unanimously agreed standard of communication and the messages can be shared between different vendors without any conflict. One more advantage of using XML data format is that, self-defined data formats can be used. The motivation to develop the TMDS architecture is to support both the models of messaging; Publish/Subscribe and Point-To-Point in the same architecture. This can be explained as:

- To develop architecture for electronic exchange.
- To provide the ability to transport XML data as a document.
- To ensure the delivery of important messages to the recipient and acknowledge the delivery.

A message is sent from one participant (the Sender) to a second participant (the Recipient). Additionally, it might be sent on behalf of a third participant (the Originator). Essentially, an interaction (message) between two participants might require the recipient to forward a similar message to some other participant. In this case, it is often necessary for the latter to know for whom the message is being sent. This can be modelled as shown in Figure 2.

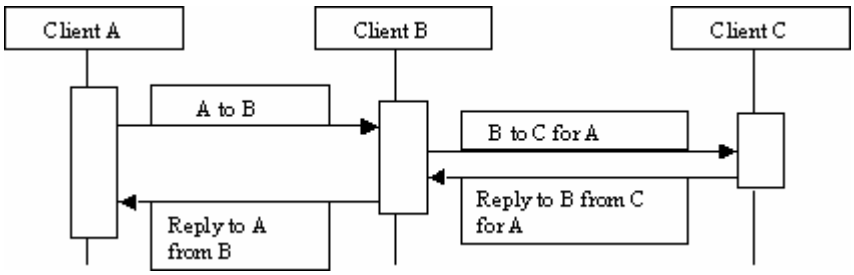

Fig. 2. XML data transfer between applications

3.1 Message Acknowledgment

Message acknowledgment protocol is most important in the guaranteed messaging domain. The JMS API provides message acknowledgment infrastructure. The successful message consumption takes place in three stages: (*i*) Message is received, (*ii*) Message is processed, and (*iii*) Message is acknowledged. Acknowledgment is set when the session is created:

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

Different types of message acknowledgment that are supported: `AUTO_ACKNOWLEDGE, DUPS_OK_ ACKNOWLEDGE, and CLIENT_ ACKNOWLEDGE.`

`AUTO_ ACKNOWLEDGE`: The session automatically acknowledges a client's receipt of a message when the client has successfully executed the `receive()` method for the queue or when the `messageListener()` is successfully executed for the topic. `AUTO_ACKNOWLEDGE` mode can be viewed in three different perspectives: Message producer, Message server and Message consumer.

Publish() and send() methods of TopicPublisher and QueuerSender respectively are synchronous methods. These methods are responsible for sending the message and wait for an acknowledgment from the server. If the server is down or the message expires and the acknowledgment could not be sent the message is considered to be undelivered and the message is sent again.

From the server perspective, an acknowledgment is sent to the producer of the message means that the server has received the message and it takes the responsibility to deliver the message to the concerned recipient, but it has not yet reached the final destination. Messages are further classified as `PERSISTENT` and `NON-PERSISTENT`. For persistent message the server first writes the message to the disk (store-and-forward mechanism) and then sends the acknowledgment to the producer.

In case of non-persistent message the server may send the acknowledgment as soon as it receives the message and the message is kept in the memory of the server, and if the server dies before delivering the message the message is lost and can not be recovered.

The subscriber can also be divided into two categories namely: durable subscriber and non-durable subscriber. If any of the clients is of durable nature then the JMS server keeps the message in the persistent storage till it receives acknowledgment from all the clients. Certain clients cannot afford redelivered message. To prevent this the JMS server sets the flag for the `getJMSRedelivered()` method, and thus guards against re-delivery of message and thus ensures once-and-only-once delivery of messages.

`DUPS_OK_ACKNOWLEDGE`: This type of acknowledgment is used if any application can afford to receive duplicate messages. `AUTO_ACKNOWLEDGMENT` incurs in extra over-head and affects the performance.

`CLIENT_ACKNOWLEDGE`: A client acknowledges a message by calling the `acknowledge()` method of message. Acknowledging a consumed message automatically acknowledges the receipt of all messages that have been delivered in that session prior to the consumption of the acknowledged message.

3.2 Allowing Messages to Expire and Setting Message Priority

TMDS architecture provides the facility to expire the message after a certain amount of time to increase the performance of the application. TMDS also allows setting the priority level of the message for urgent messages. Both of these values can be set in the `publish()` method of `TopicPublisher` class.

```
TopicPublisher.publish(message, DeliveryMode.NON_PERSISTENT,
                       7, 5000);
```

The above line of code sets the priority level of 7 (0 - lowest, 10 - highest) for the message and the time to live the message is 5 seconds. By default the message never expires and its priority level is 4. If the time-to-live is set to 0 the message never expires.

3.3 Creating Durable Subscription

TMDS architecture uses the `PERSISTENT` messages and durable subscription for the subscriber to ensure that the messages are delivered to the client. A durable subscriber has a higher over-head. The durable subscriber registers a durable subscription with a unique identity that is retained by the messaging server. Non-durable subscribers receive the messages only when they are active but durable subscribers receive all the messages for their subscription period whether or not they are active. The messaging server keeps the message in the persistent storage till the message is delivered to all the durable subscribers or the message expires. A durable subscription can have only one active subscriber at a time. The client ID is set administratively for a client specific connection factory using the `j2eeadmin` command.

```
J2eeadmin –addJmsFactory DURABLE_TCF topic –props clientID=MYID
```

For the normal subscriber, the subscription is only when the subscriber is active, but for the durable subscriber the subscription is still active even if the subscriber is off-line and the subscription lasts till the `unsubscribe()` method is called.

3.4 Features of TMDS Architecture

The messages are not directly delivered to the recipient but the messages are delivered to the recipient via the virtual destinations called *'topic'* or '*queue*'. Destinations are the delivery labels in messaging rather than the place where the message is ultimately delivered. A destination is the commonly understood staging area for the message. The overview of TMDS Architecture distinguishes the two JMS messaging domains.

Point-to-Point (PTP): Produces messages to a named '*queue***',** which is the virtual destination of the message, placing new messages at the back of the queue. Prospective consumers of messages addressed to a queue can either receive the front-most message (thereby removing it from the queue) or browse through all the messages in the queue, causing no changes. Several clients can send messages to a '*queue*', but only one client can receive the message (one-to-one communication).

Publish and Subscribe (Pub/Sub): Produces messages to a '*topic'***,** which is also a virtual destination for the message like *queue*. Prospective consumers of messages addressed to a topic simply subscribe to the topic. While a message can have many subscribers (one-to-many), the producer does not know how many subscribers, if any, exist for a topic.

Scalability: With TMDS Architecture, a B2B exchange can readily scale to more number of trading partners without requiring changes to the routing architecture or the trading applications. New e-business partners can subscribe to the existing *Topic*

and get the information, they also have the option to get a dedicated channel for communication from the e-broker.

Reliability: Messages can be guaranteed to persist when a message is sent to a queue. If the pub/sub domain of messaging is used, then the message property must be set to PERSISTENT to ensure guaranteed delivery of message. Mobile users, although connected to the network frequently, need not be concerned that they missed out on messages published when they were unable to receive them.

High Performance: TMDS architecture enables flexible programming models, both PTP and pub/sub domains of messaging have been implemented. If any message is to be directed only to the concerned e-business partner, it is delivered via the PTP model and saves the over-head of publishing the message. The second model (Pub/Sub) is used when the message is to be sent to a group of interested recipient.

Enterprise Application Integration: The basic purpose of TMDS architecture is to enable communication between enterprise applications. And most of the enterprise works on their legacy systems and won't be interested in changing their systems. If standard data format is used for communication, this could solve the purpose to some extend. XML was designed to describe the data and to focus on actual data. Users can define their own XML tags to describe the data.

4 TMDS Architecture with XML: A Case Study

The traditional way of exchanging data was through EDI, which uses proprietary data formats, which is defined by a specific company and cannot be used without their permission. XML offers a method to represent the data that is not proprietary. However XML does not have a reliable way of transporting critical business data over the intra-company communication environment. Server or network failure can occur during communication. Applications participating in the distributed environment can crash or have scheduled down time, thus a reliable transport mechanism is required to overcome these issues, e.g. if any client wants to communicate price changes to all the other parties, it must be ensured that message is delivered to all the disconnected clients as well. The RPC mechanism doesn't provide features like: persistence, verification and transactional support, so these features have to be embedded in the application logic.

4.1 Case Study

The TMDS Architecture enables to communicate between different trading partners. It has been simulated for a transport exchange but can be extended for any application where clients have to communicate with each other in a distributed environment.

There are four major components in this application: (*i*) Client, (*ii*) RMI Server, (*iii*) Message Server, and (*iv*) Transport Companies (could be any e-partner in the business).

The client has to be a registered user of the site. If the client is accessing the site for the first time he has to register himself with the e-broker and the details will be stored in the database. As the client enters the site, he has to mention the origin and

destination of the goods to be delivered along with the date of delivery. The client has two options for selecting the Transport Company. The client could either select a specific transport company to deliver the goods or they could select “no choice”, if the client is not sure of the company to be used. If the client opts for a specific transport company the request is sent to the transport company through a queue and if the client has no specific option the order is published to all the transport companies.

If the order is published to all the clients then the transport companies reply back their interest in the specific order to the Message Server via the queue.

```
public interface Interface extends Remote
{
     public void publishOrder (String cFrom, String cTo,
               String cDay, String cMonth, String cYr)
               throws java.rmi.RemoteException;
     public void queueOrder (String cName)
               throws java.rmi.RemoteException;
}
```

The sample code for the publisher to the “Topic” of message is as follows:

```
public void publishOrder(String cFrom, String cTo, String cDay,
   String cMonth, String  cYr)
{
     String   topicName = null;
     Context  jndiContext = null;
     TopicConnectionFactory
     topicConnectionFactory = null;
     TopicConnection  topicConnection = null;
     TopicSession  topicSession = null;
     Topic  topic = null;
     TopicPublisher  topicPublisher = null;
     TextMessage  message = null;
     final int  NUM_MSGS =1;
}
```

/* Creates a string with XML tags and publishes this message to the topic, which the subscribers receive and store it in the file and parses the file to get the data. */

```
String xmlString = null;
Date date = new Date ();
xmlString="<?xml version="+"\"1.0\""+" ?>
     <order><origin>"+cFrom+"</origin>
     <destination>"+ cTo +"</destination>
     <delivery_day>"+cDay +"</delivery_day>
     <delivery_month>"+cMonth+"</delivery_month>
     <delivery_yr>"+cYr+"</delivery_yr>
     </order>";
```

/* Create a JNDI InitialContext object if none exists yet.*/

```
topicName = "NewOrder";
try
{
     jndiContext = new InitialContext ();
}
catch(NamingException e){
     System.out.print("JNDI Error "+e.toString ());
```

```
            System.exit (1);
        }
        /* Look up connection factory and topic. If either does not exist, exit. */
        try
        {
            topicConnectionFactory=(TopicConnectionFactory)
                jndiContext.lookup"TopicConnectionFactory"
            topic = (Topic)jndiContext.lookup (topicName);
        }
        catch(NamingException e){
            System.out.println("Lookup Fail"+e.toString());
            System.exit (1);
        }
        /* Create connection, Create session from connection; false means
        session is not transacted. */
        try
        {
            topicConnection =
            topicConnectionFactory.createTopicConnection();
            topicSession=topicConnection.createTopicSession
                (false, session.AUTO_ACKNOWLEDE);
            topicPublishr=topicSesion.createPublisher(topic)
            message = topicSession.createTextMessage ();
            /* sets the message stream to xml message format.*/
            message.setText(xmlString)"Order Recived:"+date;
            System.out.print("New Order"+message.getText());
            topicPublisher.publish (message);
        }
        catch(JMSException e) {
            System.out.println("Exception:"+ e.toString ());
        }
        finally
        {
            if(topicConnection != null){
                try{
                    topicConnection.close();
                }
                catch(JMSException e){}
        }
    }
}
```

The message, which is published at the message server and then is send to the subscribers, is transferred in XML format. Similarly, there is another class namely `TopicSubscriber` which subscribes to any specific `topic` and receives all the published messages.

5 Conclusion and Future Work

TMDS architecture supports both the domain of messaging, pub/sub and Point-to-Point, thus enhancing the features of JMS specification. The basic aim of the architecture is to enhance the features of existing Middle-ware technologies (like RMI, DCOM etc.). TMDS architecture uses the advantages of existing Middle-ware technology and enhances the feature of distributed communication by adding asynchronous communication facility, prioritizing the message delivery as per the importance of message and a standard format of data transfer (XML format).

Aim of a new architecture shouldn't be to replace the existing one, but it should have the capability to be integrated with the existing system. TMDS architecture has the capability of integration with the existing messaging systems (supports EAI). There are a lot of commercial products for communication in distributed environment, but industry is still waiting for a standard architecture. TMDS architecture is a step, which implements and extends the features of communication standards proposed by Sun Micro-system's JMS. TMDS architecture uses the data-centric XML. Using the document-centric XML can still enhance the features of the architecture.

References

[1] Adler, R. M. "Distributed Coordination Models for Client/Sever Computing." *Computer 28,* 4, 14-22 April 1995.

[2] Bernstein, Philip A. "Middleware: A Model for Distributed Services." *Communications of the ACM 39*, 2, 86-97, February 1996.

[3] Eckerson, Wayne W., "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications", *Open Information Systems 10*, 20, 1, January 1995.

[4] Newell D., Jones, O., and Machura, M. "Interoperable Object Models for Large Scale Distributed Systems," 30-31. *Proceedings. International Seminar on Client/Server Computing*. La Hulpe, Belgium, October 30-31, 1995. London, England: IEE, 1995.

[5] Rao, B.R. "Making the Most of Middleware." *Data Communications International 24*, 12, 89-96 September 1995.

[6] Schill, Alexander. "DCE-The OSF Distributed Computing Environment Client/Server Model and Beyond," 283. *Intl. DCE Workshop,* Germany, Springer-Verlag, Oct. 1993.

[7] Gopalan S, Grant S., Giotta P "*Professional JMS*", 1st Edition, Wrox publicaton.

[8] Mark Hapner, Java Message Service Specification version 1.0.2b, Sun Microsystems, August, 2001, http://java.sun.com/jms/

[9] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging", *Proceedings of the Middleware2000 Conference*, ACM/IFIP, Apr. 2000.

[10] Birmen K., *"Building Secure and Reliable Network applications"* Manning Publishing and Prentice Hall, December 1996.