****

**Name of the Student: Choudhari Arti Rajendra Roll no: 08**

**CLASS:- B.E.[I.T] Division: A Course:- 2015**

**Assignment No.6**

**COMPUTER LABORATORY-IX**

**Marks: /10**

**Date of Performance: Sign with Date:**

## ASSIGNMENT NO.6

**ProblemStatement:**

To develop any distributed application using Messaging System in Publisher-Subscriber paradigm.

**Objective:**

1) The course aims to provide an understanding of the principles on which the distributed systems are based; their architecture, algorithms and how they meet the demands of Distributed applications.

2) The course covers the building blocks for a study related to the design and the implementation of distributed systems and applications.

**Outcomes:**

1) Demonstrate knowledge of the core concepts and techniques in distributed systems.

2) Learn how to apply principles of state-of-the-Art Distributed systems in practical application.

3) Design, build and test application programs on distributed systems.

**PEOs:2; POs: a,b,c,d,f,g,i, l, m ; PSOs: 1,2,3 and COs satisfied: 1, 2, 3.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Assignment No.** | **Assignment Title** | **Assignment Statement** | **Scenarios** | **Software Required** |
| 6 | Publish /Subscribe | To develop any distributed application using Messaging System in Publish-Subscribe paradigm. | The program will applied in case of messages which get broadcasted in logical channels. The subscribers will get requisite messages for which they are already subscribed. | 1) eclipse java(EE) version (June 2017)(Oxygen) (optional)  2) apache-activemq-5.15.8\_bin (To run activemq server); apachemq-all-5.15.8.jar : Add this external jar in Eclipse. |

## Tools /Environment:

Java Programming Environment, JDK 8, Eclipse IDE, Apache ActiveMQ 4.1.1, JMS

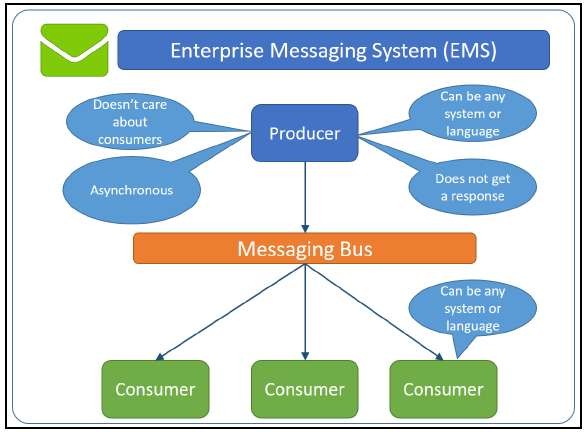
## RelatedTheory:

Large distributed systems are often overwhelmed with complications caused by heterogeneity and interoperability. Heterogeneity issues may arise due to the use of different programming languages, hardware platforms, operating systems, and data representations. Interoperability denotes the ability of heterogeneous systems to communicate meaningfully and exchange data or services. With the introduction of middleware, heterogeneity can be alleviated and interoperability can beachieved.

Middleware is a layer of software between the distributed application and the operating system and consists of a set of standard interfaces that help the application use networked resources and services.

## Enterprise Messaging System:

EMS, or the messaging system, defines system standards for organizations so they can define their enterprise application messaging process with a semantically precise messaging structure. EMS encourages you to define a loosely coupled application architecture in order to define an industry-accepted message structure; this is to ensure that published messages would be persistently consumed by subscribers. Common formats, such as XML or JSON, are used to do this. EMS recommends these messaging protocols: DDS, MSMQ, AMQP, or SOAP web services. Systems designed with EMS are termed **Message-Oriented Middleware (MOM)**. An asynchronous communication is used while messaging in EMS.



## Java Messaging Service

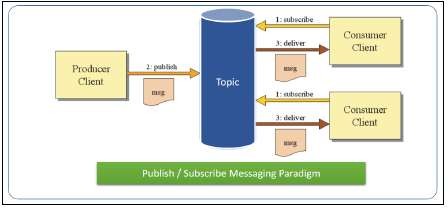
**Java's implementation of an EMS in the Application Programming Interface (API) format is known as JMS.**

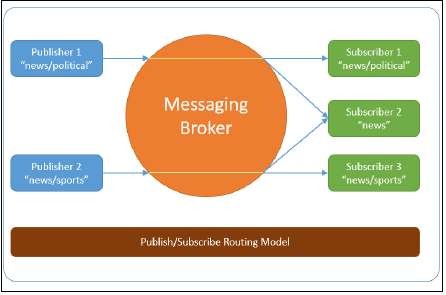
JMS allows distributed Java applications to communicate with applications developed in any other technology that understands messaging through asynchronous messages. JMS applications contain a provider, clients, messages, and administrated objects.

JMS providing a standard, portable way for Java programs to send/receive messages through a MOM product. Any application written in JMS can be executed on any MOM that implements the JMS API standards. The JMS API is specified as a set of interfaces as part of the Java API. Hence, all the products that intend to provide JMS behavior will have to deliver the provider to implement JMS-defined interfaces. With programming patterns that allow a program to interface, you should be able to construct a Java application in line with the JMS standards by defining the messaging programs with client applications to exchange information through JMS messaging.

## The publish/subscribe messaging paradigm:

The publish/subscribe messaging paradigm is built with the concept of a topic, which behaves like an announcement board. Consumers subscribe to receiving messages that belong to a topic, and publishers report messages to a topic. The JMS provider retains the responsibility for distributing the messages that it receives from multiple publishers to many other subscribers based on the topic they subscribe to. A subscriber receives messages that it subscribes to based on the rules it defines and the messages that are published after the subscription is registered; they do not receive any messages that are already published, as shown in the followingdiagram:





## JMS interfaces

JMS defines a set of high-level interfaces that encapsulate several messaging concepts. These high-level interfaces are further extended for the Point-To-Point and publish/subscribe messagingdomains:

ConnectionFactory:This is an administered object with the ability to create a connection.

Connection:This is an active connection handle to the provider.

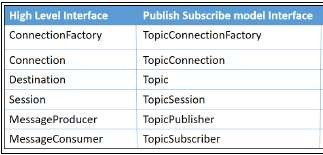
Destination: This is an administered object that encapsulates the identity of a message destination where messages are sent to/received from.

Session:This is a single-threaded context for sending/receiving messages. To ensure a simple session-based transaction, concurrent access to a message by multiple threads is restricted. We can use multiple sessions for a multithreaded application.

MessageProducer:This is used to send messages.

MessageConsumer:This is used to receive messages.

The following table shows interfaces specific to publish/subscribe paradigms enhanced from their corresponding high-levelinterface:

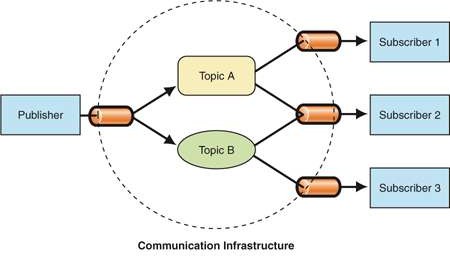


## Designing thesolution:

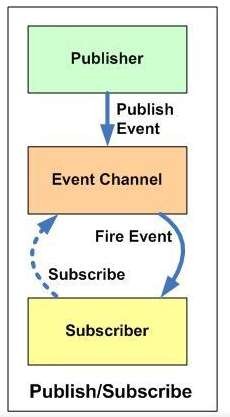
In ‘Publisher-Subscriber’ pattern, senders of messages, called **publishers**, do not program the messages to be sent directly to specific receivers, called **subscribers.**

For example, consider there is a publisher publishes news (topics) related to politics and sports; they publish to the Messaging Broker, as shown in the following diagram. While Subscriber 1 receives news related to politics and Subscriber 3 receives news related to sports, Subscriber 2 will receive both political and sports news as it subscribed to the common topics.

In designing our solution, we have created one publisher and subscriber wherein the publisher creates topic.



The **Publisher/Subscriber** pattern is mostly implemented in an ***asynchronous*** way (using message queue).

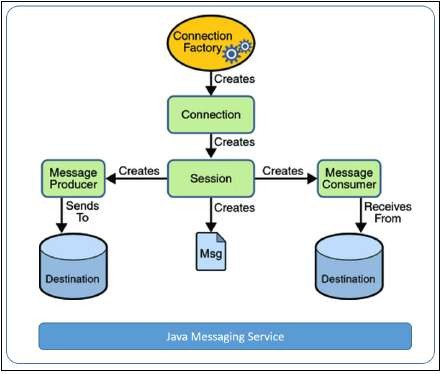


Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

JMS is a Java API that allows applications to create, send, receive, and read messages. The JMS API enables communication that is loosely coupled, asynchronous and reliable.

To use JMS, we need to have a JMS provider that can manage the sessions, queues, and topics. Some examples of known JMS providers are Apache ActiveMQ, WebSphere MQ from IBM or SonicMQ from Aurea Software. Starting from Java EE version 1.4, a JMS provider has to be contained in all Java EE application servers.

Refer to the following diagram:



A JMS provider is a messaging server that supports the creation of connections (multithreaded virtual links to the provider) and sessions (single-threaded contexts for producing and consuming messages). A JMS client is a Java program that either produces or consumes messages.

JMS messages are objects that communicate information between JMS clients and are composed of a header, some optional properties, and an optional body.

**Administered objects** are preconfigured JMS objects, such as a connection factory (the object a client uses to create a connection to a provider) and a destination (the object a client uses to specify a target for its messages).

JMS applications are usually developed in either the publish/subscribe or Point-To-Point paradigm.

The following are the objectives of JMS, as highlighted in its specification:

* Defining a common collection of messaging concepts andfeatures
* Minimizing the number of concepts a developer should learn todevelop
* applications asEMS's
* Improving the application messagingportability
* Reducing the effort involved in implementing aprovider
* Providing client interfaces for both Point-To-Point and pub/subdomains

## Implementing thesolution:

1. To execute the pub-sub programs, you need the message queue environment.

The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. It is a Java API that allows applications to create, send, receive, and read messages. The JMS API enables communication that is loosely coupled, asynchronous andreliable.

To use JMS, we need to have a JMS provider that can manage the sessions, queues, and topics. Some examples of known JMS providers are Apache ActiveMQ, WebSphere MQ from IBM or SonicMQ from Aurea Software. Starting from Java EE version 1.4, a JMS provider has to be contained in all Java EE application servers.

Here we are implementing the JMS concepts and illustrates them with a JMS Hello World example using ActiveMQ.

Interfaces extending core JMS interfaces for Topic help build publish-subscribe components.

1. The Publisher.javaprogram to publish messages to the Publish-Subscribe topic. The code for which is shown in the belowsection.
2. While the preceding program helps publish messages to the Publish-Subscribe Topic, the Subscribe.javaprogram is used to subscribe to the Publish-Subscribe Topic, which keeps receiving messages related to the Topic until the quit command isgiven.

**Compilation and Executing thesolution:**

**For Setting up an environment:**

1. Download the 2 Jar files javax.jms.jar for JMS and apache-activemq-4.1.1.jar for ApacheActiveMQ.
2. Download Apache MQ and Install it using the Apache MQ InstallationLink

Links for Download and installation instruction:

* 1. Jms<http://www.java2s.com/Code/Jar/j/Downloadjavaxjmsjar.htm>……… [Jarfile]
  2. Apache <http://www.java2s.com/Code/Jar/a/Downloadapacheactivemq411jar.htm>[Jarfile]
  3. Download-<http://activemq.apache.org/activemq-5158-release.html> [ApacheMQDownloadlink]
  4. Install -https://docs.wso2.com/display/BAM200/Installing+Apache+ActiveMQ+on+Linux

…………. [Apache MQ Installation Instructions]

* 1. Concept -https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

## Steps to execute:

* + 1. Create a Publisher.javafile and copy paste the Publisher code
    2. Create a Subscriber.javafile and copy paste the Subscriber code

1. Add externaljars
   1. Right Click on Project in eclipse package explorer
   2. Go to BuildPath
   3. Select Configure BuildPath
   4. Add externaljars
   5. Select both the downloaded jars from the firststep
2. Run activemq with the followingcommand:

## sudosh active start

1. Run the publisher code and pin console forpublisher
2. RunSubscriber

**Conclusion:** This assignment includes study of Publish-Subscribe model of Communication which is implemented using JMS and Apache ActiveMQ. The topic based filtering requires the messages to be broadcasted into logical channels, the subscribers only receives messages from logic channels they aresubscribed.

**Code:-**

.

//1)PUBLISHER

package pubsub;

import javax.jms.\*;

import org.apache.activemq.ActiveMQConnection;

import org.apache.activemq.ActiveMQConnectionFactory;

public class Publisher {

private static String url = ActiveMQConnection.DEFAULT\_BROKER\_URL;

public static void main(String[] args) throws JMSException {

ConnectionFactoryconnectionFactory = new ActiveMQConnectionFactory(url);

Connection connection = connectionFactory.createConnection();

connection.start();

// JMS messages are sent and received using a Session. We will

// create here a non-transactional session object. If you want

// to use transactions you should set the first parameter to 'true'

Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);

Topic topic = session.createTopic("CL9");

MessageProducer producer = session.createProducer(topic);

// We will send a small text message saying 'Hello'

TextMessage message = session.createTextMessage();

message.setText("This is a new message from publisher");

// Here we are sending the message!

producer.send(message);

System.out.println("Sent message '" + message.getText() + "'");

connection.close();

}

}

**/**/2)SUBSCRIBER

package pubsub;

import java.io.IOException;

import javax.jms.\*;

import org.apache.activemq.ActiveMQConnection;

import org.apache.activemq.ActiveMQConnectionFactory;

public class Subscriber {

// URL of the JMS server

private static String url = ActiveMQConnection.DEFAULT\_BROKER\_URL;

// Name of the topic from which we will receive messages from = " CL9"

public static void main(String[] args) throws JMSException {

// Getting JMS connection from the server

ConnectionFactoryconnectionFactory = new ActiveMQConnectionFactory(url);

Connection connection = connectionFactory.createConnection();

connection.start();

Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);

Topic topic = session.createTopic("CL9");

MessageConsumer consumer = session.createConsumer(topic);

MessageListenerlistner = new MessageListener() {

public void onMessage(Message message) {

try {

if (message instanceofTextMessage) {

TextMessagetextMessage = (TextMessage) message;

System.out.println("Received message" + textMessage.getText() + "'");

}

} catch (JMSException e) {

System.out.println("Caught:" + e);

e.printStackTrace();

}

}

};

consumer.setMessageListener(listner);

try {

System.in.read();

} catch (IOException e) {

e.printStackTrace();

}

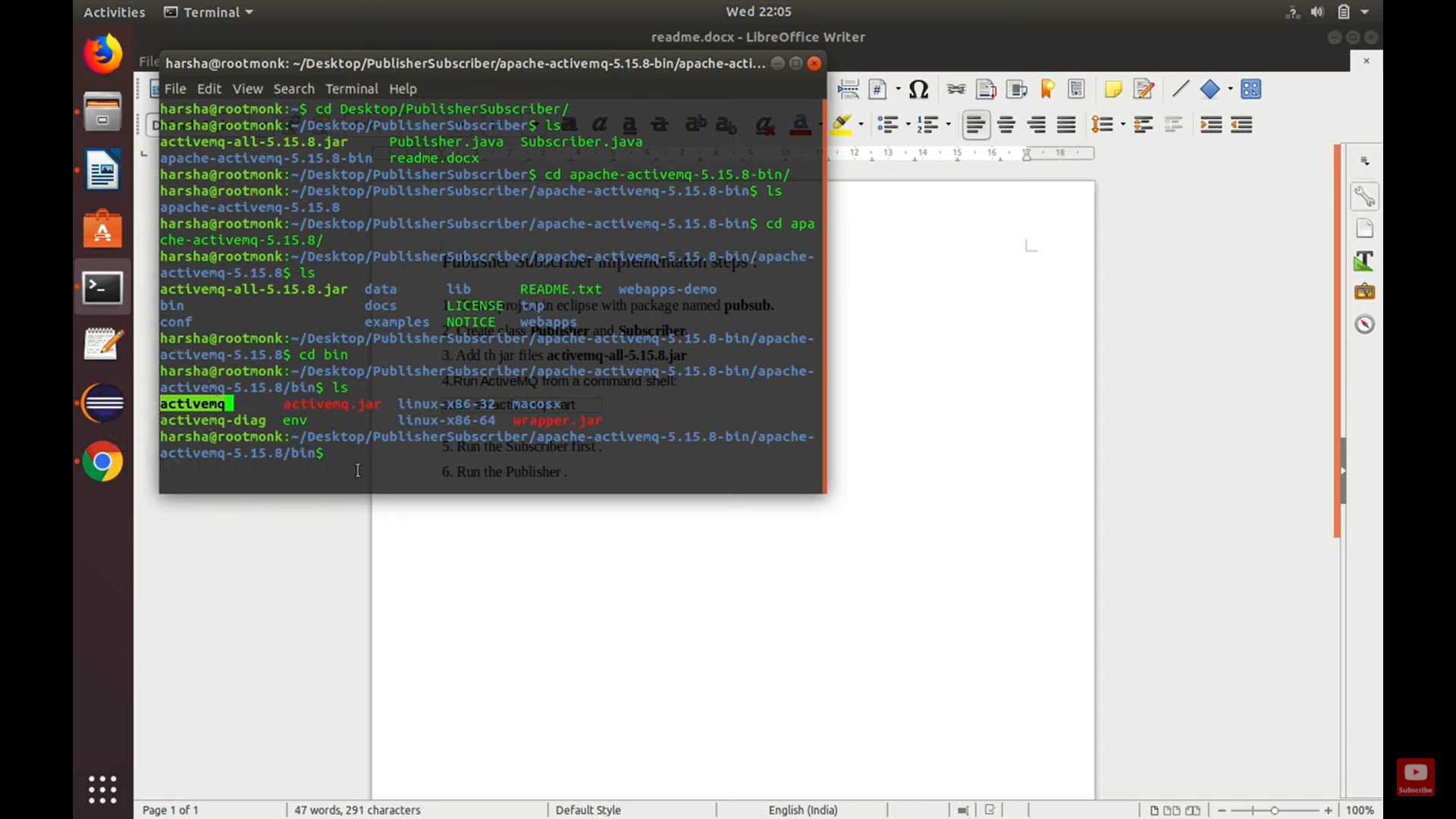
connection.close();

}

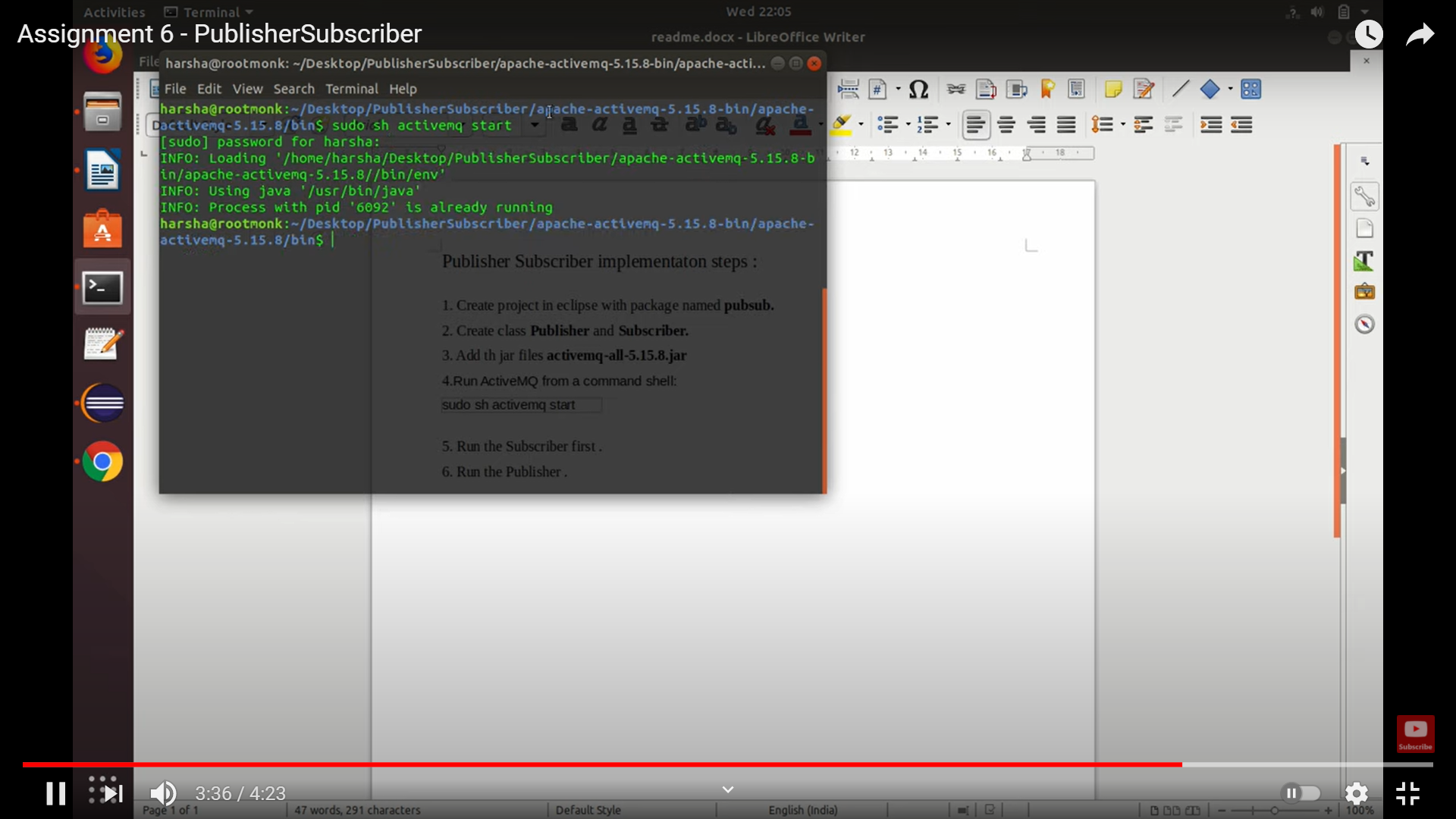
}

**OUPUT:**

**FIG1) ACTIVE MQ**



**FIG 2) GENERATE OUTPUT**



**FIG3) MESSAGE SENT FROM PUBLISHER TO SUBSCRIBER**

