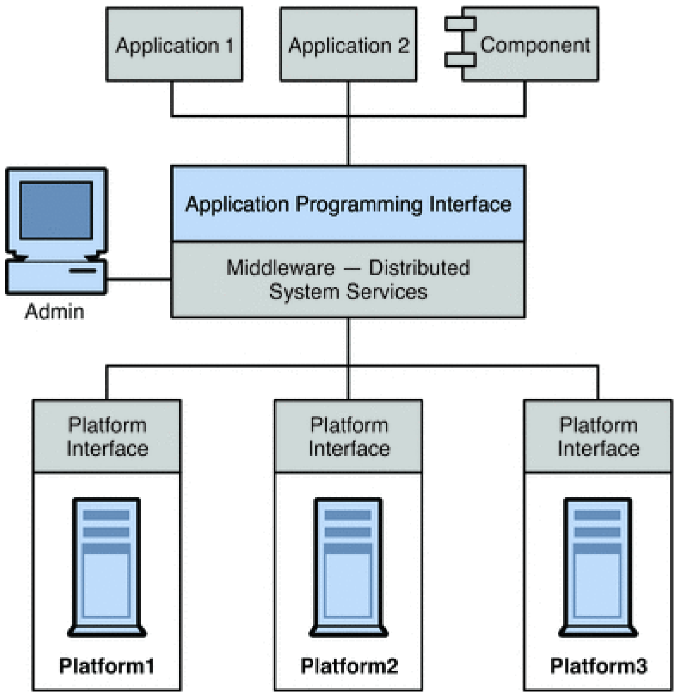
Message-Oriented Middleware

Introduction

Message-oriented middleware, M.O.M. or just messaging, can be understood as an architecture of a distributed system, where it represents a middle layer (therefore the name “message-oriented **middle**ware, and it is being used as a complete and secure solution for transfering data in a fast and smooth manner between different components of the system.

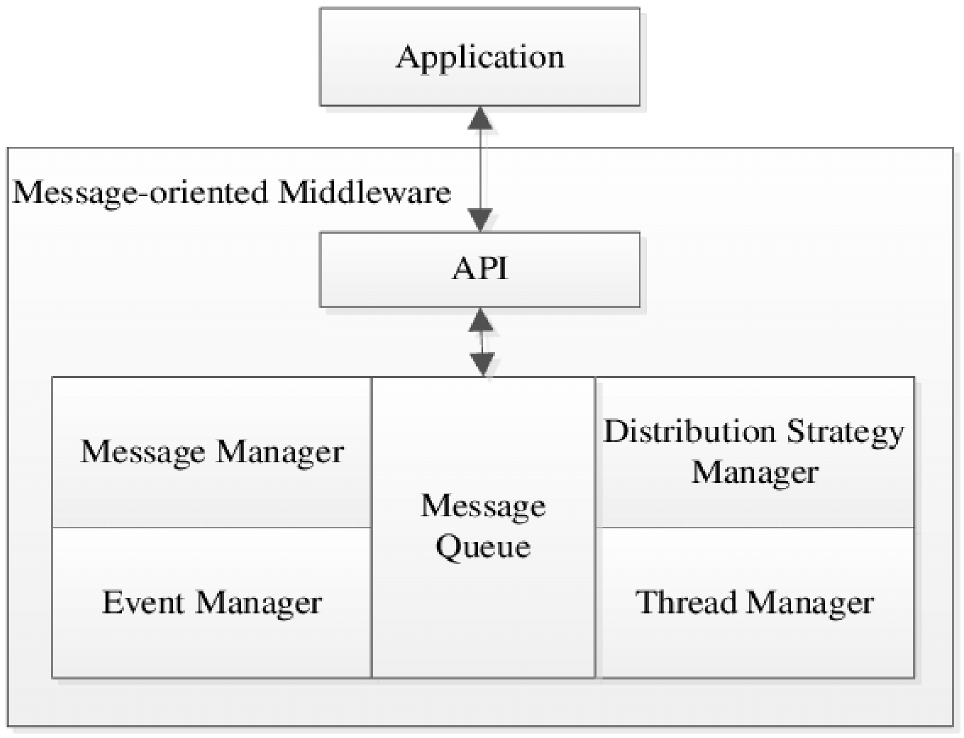
Modern systems operate in complex environments with multiple programming languages, hardware platforms, operating systems and the requirement for dynamic flexible deployments with 24/7 reliability, high throughput performance and security while maintaining a high Quality-of-Service (QoS). In these environments, the traditional direct Remote Procedure Call (RPC) mechanisms quickly fail to meet the challenges present. In order to cope with the demands of such systems, an alternative to the RPC distribution mechanism has emerged. This mechanism called Message-Oriented Middleware or MOM provides a clean method of communication between disparate software entities.

Middleware, allows software components (applications, enterprise java beans, servlets, and other components) that have been developed independently and that run on different networked platforms to interact with one another. It is when this interaction is possible that the network can become the computer.



As shown in figure, conceptually, middleware resides between the application layer and the platform layer (the operating system and underlying network services).

Message oriented middleware architecture



Middleware categories

- Remote procedure call or RPC-based middleware
- Object request broker or ORB-based middleware
- Message Oriented Middleware or MOM-based middleware

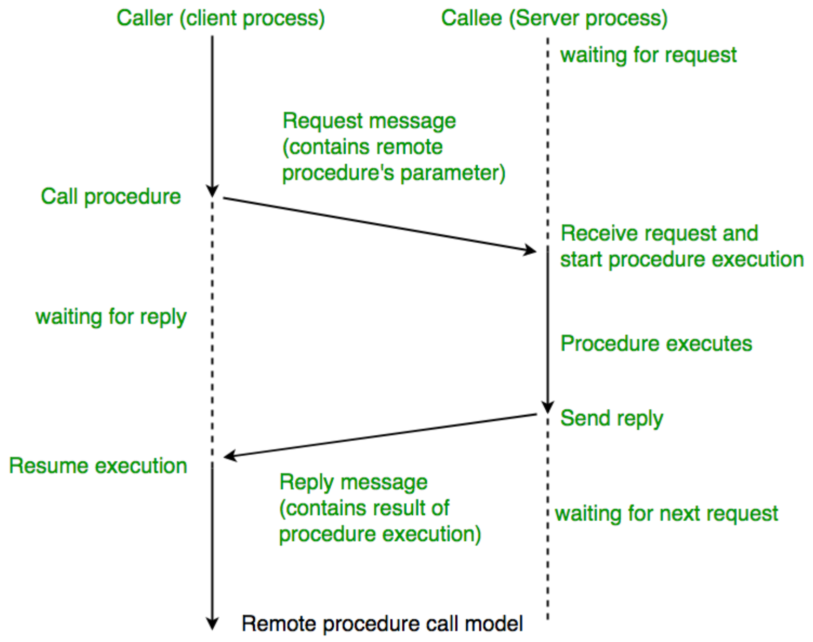
All these models make it possible for one software component to affect the behavior of another component over a network. They are different in that RPC- and ORB-based middleware create systems of tightly coupled components, whereas MOM-based systems allow for a looser coupling of components. In an RPC- or ORB-based system, when one procedure calls another, it must wait for the called procedure to return before it can do anything else. In these synchronous messaging models, the middleware functions partly as a super-linker, locating the called procedure on a network and using network services to pass function or method parameters to the procedure and then to return results.

Introduction to the Remote Procedure Call (RPC)

The traditional RPC model is a fundamental concept of distributed computing. It is utilized in middleware platforms including CORBA, Java RMI, Microsoft DCOM, and XML-RPC. The objective of RPC is to allow two processes to interact. RPC creates the fac¸ade of making both processes believe they are in the same process space (i.e., arethe one process). On the basis of the synchronous interaction model, RPC is similar to a local procedure call whereby control is passed to the called procedure in a sequential synchronous manner while the calling procedure blocks waiting for a reply to its call.

RPC can be seen as a direct conversation between two parties (similar to a person-to-person telephone conversation). An example of an RPC-based distributed system deployment is

detailed in Figure


Caller (client process)
Callee (Server process)
waiting for request
Request message (contains remote procedure's parameter)
Call procedure
Receive request and start procedure execution
waiting for reply
Procedure executes
Send reply
Resume execution
Reply message (contains result of procedure execution)
waiting for next request
Remote procedure call model

Message Queues

The message queue is a fundamental concept within MOM. Queues provide the ability to store messages on a MOM platform. MOM clients are able to send and receive messages to and from a queue. Queues are central to the implementation of the asynchronous interaction model within MOM. A queue, as shown in Figure 1.5, is a destination where messages may be sent to and received from; usually the messages contained within a queue are sorted in a particular order. The standard queue found in a messaging system is the First-In First-Out (FIFO) queue; as the name suggests, the first message sent to the queue is the first message to be retrieved from the queue. Many attributes of a queue may be configured. These include the queue's name, queue's size, the save threshold of the queue, message-sorting algorithm, and so on. Queuing is of particular benefit to mobile clients without constant network connectivity, for example, sales personnel on the road using mobile network (GSM, GRPS, etc) equipment to remotely send orders to head office or remote sites with poor communication infrastructures. These clients can use a queue as a makeshift inbox, periodically checking the queue for new messages. Potentially each application may have its own queue, or applications may share a queue, there is no restriction on the setup. Typically, MOM platforms support multiple queue types, each with a different purpose. Table 1.1 provides a brief description of the more common queues found in MOM implementations.

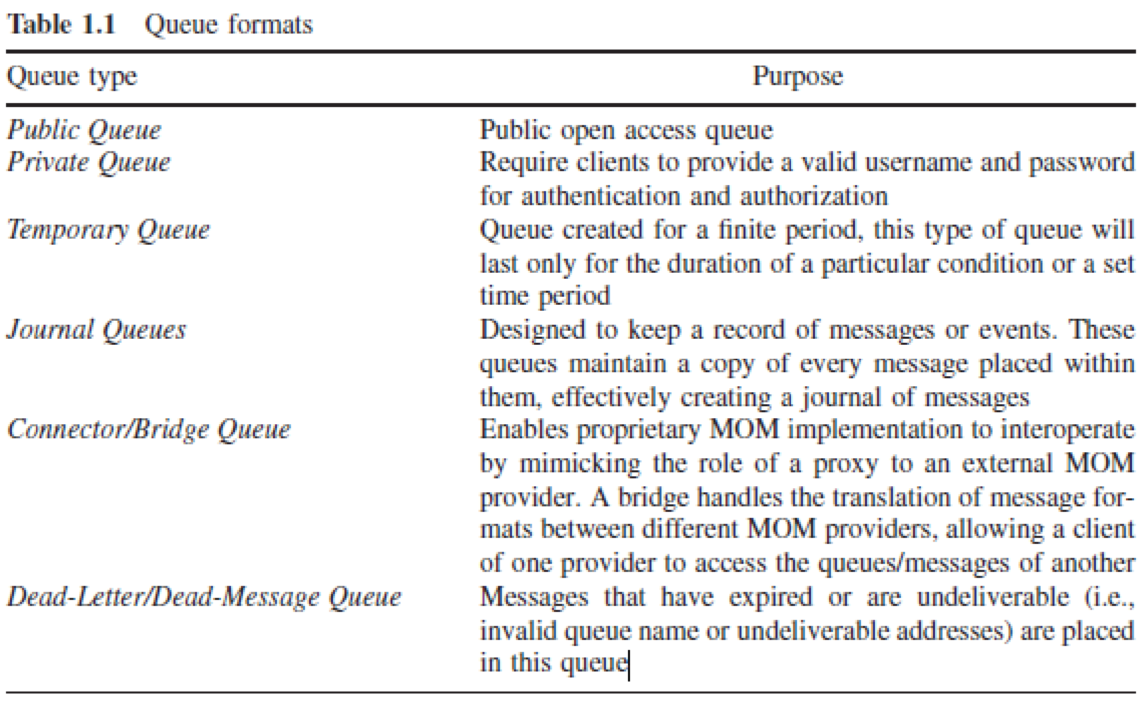


Table 1.1 Queue formats

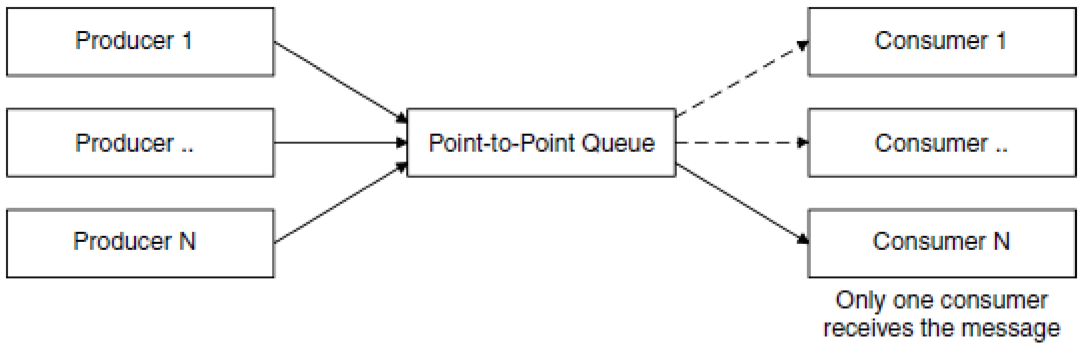
Queue type	Purpose
Public Queue	Public open access queue
Private Queue	Require clients to provide a valid username and password for authentication and authorization
Temporary Queue	Queue created for a finite period, this type of queue will last only for the duration of a particular condition or a set time period
Journal Queues	Designed to keep a record of messages or events. These queues maintain a copy of every message placed within them, effectively creating a journal of messages
Connector/Bridge Queue	Enables proprietary MOM implementation to interoperate by mimicking the role of a proxy to an external MOM provider. A bridge handles the translation of message formats between different MOM providers, allowing a client of one provider to access the queues/messages of another
Dead-Letter/Dead-Message Queue	Messages that have expired or are undeliverable (i.e., invalid queue name or undeliverable addresses) are placed in this queue

Messaging Models

A solid understanding of the available messaging models within MOM is key to appreciate the unique capabilities it provides. Two main message models are commonly available, the point-to-point and publish/subscribe models. Both of these models are based on the exchange of messages through a channel (queue). A typical system will utilize a mix of these models to achieve different messaging objectives.

Point-to-Point

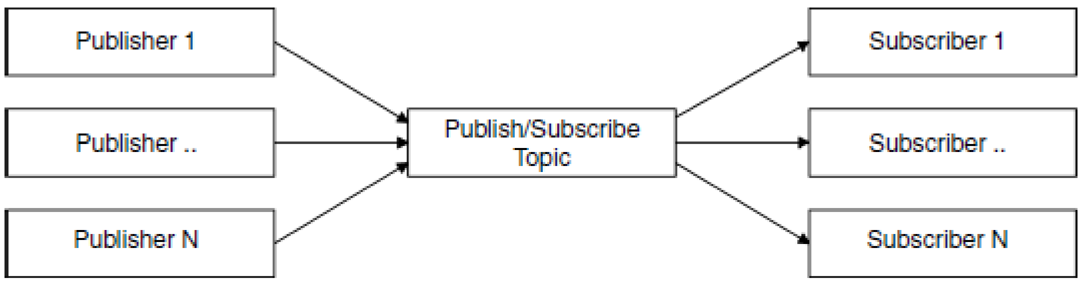
The point-to-point messaging model provides a straightforward asynchronous exchange of messages between software entities. In this model, shown in Figure 1.6, messages from producing clients are routed to consuming clients via a queue. As discussed earlier, the most common queue used is a FIFO queue, in which messages are sorted in the order



Publish/Subscribe

The publish/subscribe messaging model, is a very powerful mechanism used
to disseminate information between anonymous message consumers and producers. These
one-to-many and many-to-many distribution mechanisms allow a single producer to send
a message to one user or potentially hundreds of thousands of consumers.
In the publish/subscribe (pub/sub) model, the sending and receiving application is free
from the need to understand anything about the target application. It only needs to send
the information to a destination within the publishes/subscribes engine. The engine will then
send it to the consumer. Clients producing messages "publish" to a specific topic or channel,
these channels are then "subscribed" to by clients wishing to consume messages. The
service routes the messages to consumers on the basis of the topics to which they have subscribed
as being interested in. Within the publish/subscribe model, there is no restriction
on the role of a client; a client may be both a producer and consumer of a topic/channel.

A number of methods for publish/subscribe messaging have been developed, which support
different features, techniques, and algorithms for message filtering, publication,
subscription, and subscription management distribution.



Extending Message-Oriented Middleware

In the past, private and public sector organizations were focused on the custom development of software applications and paid little attention to the integration issues. Software applications were integrated within each organization and the integration required using the same Information Technology (IT) platform to run all of them.

Recently, many organizations are faced with the software integration problem. Sometimes, the problem is a consequence of the globalization process. For instance, when different organizations are merged, each with its own IT infrastructure, there is a need to consolidate data produced and maintained by the applications that cross organizational boundaries, often developed in different programming languages and running on different platforms. The integration problem may also arise when the applications are designed to run as web services, with high-level services built by integrating multiple individual services. Not only such services are running in different environments, the integration must be done dynamically at runtime. The integration problem is particularly acute in the public sector, where government agencies are developing their own IT solutions based on individual needs. Now, trying to deliver better public services and improve efficiency through the use of Information and Communication Technologies (ICT), they have to integrate heterogeneous software applications to deliver public services that seamlessly cross agency boundaries.

Core Message Gateway

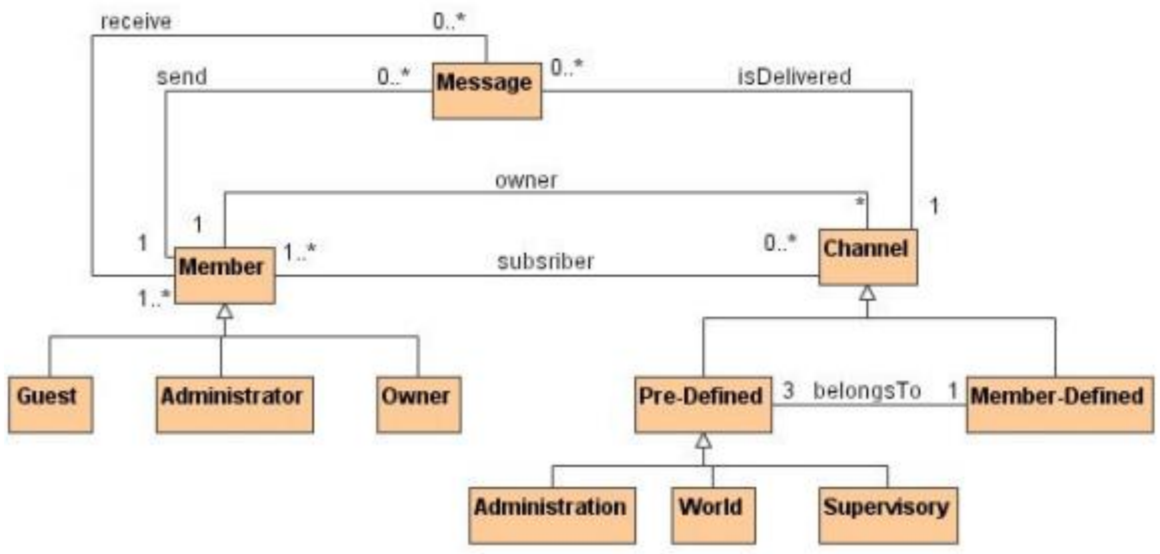
MOM systems rely upon the basic concepts of messages and channels. A message is an atomic packet of data that can be transmitted through a channel. A channel is a virtual pipe that connects a sender to a receiver [7]. Many design decisions must be taken related to messages and channels when designing a messaging system, such as: how to distribute messages, how to format them, how the channels are defined, and others. Two particular models for communicating messages through channels are publish-and subscribe and point-to-point. The publish-and-subscribe model describes a one-to many broadcasts of messages between a single producer and several consumers of messages. The point-to-point model describes one communication between a single producer and a single consumer. In the former approach, many consumers can receive the same message. In the latter, exactly one consumer receives the message.

• **Message**: A message is a package of information. It is prepared and delivered to a particular channel by one of its subscriber (producer) and is received by all other subscribers of this channel (consumers). Messages are sent as XML documents.

• **Member**: A member is a registered user of the system, typically a software application or a human user. Three distinguished member types are: guest, administrator and owner. A guest is the member who can only communicate with the administrator, usually to register a new member. An administrator is the member who can register and un-register other members. There is only one guest and one administrator in the whole system. An owner is associated with every channel, being the member who created this channel. An owner has the right to monitor the flow of messages passing through the channel and to carry out channel management functions, including the right to destroy the channel. Different members can ask the owner to subscribe and unsubscribe the channel. Once subscribed, a member can post messages to this channel as well as receive all messages posted to the channel by other subscribers. Each channel has exactly one owner and several subscribers. The owner of a channel is also its subscriber.

• **Channel:** A channel is an abstraction created for the purpose of exchanging messages between a set of members (subscribers). A subscriber of a channel is allowed to post messages to the channel, read messages posted to the channel or both. The gateway contains three kinds of pre-defined channels: administration channel, world channel and supervisory channels,

one for every member-defined channel. The administration channel connects all members with the administrator. The world channel connects all members with each other. A supervisory channel connects all subscribers of a given user-defined channel with the owner of this channel. Other channels are member-defined.



These concepts are depicted as the UML conceptual class diagram.

Message Gateway Extensions

The core messaging framework assures that the messages are delivered reliably between relevant members of the gateway. However, for realistic applications this basic property is necessary but likely insufficient. Other functions will be needed, allowing for instance to validate, transform and encrypt messages. The aim of this section is to briefly present the set of possible extensions to the core framework.

The extensions include:

1) Auditing: This extension enables the auditing of messages sent through a channel, so that the history of messages can be scanned later by the owner of this channel. Once the owner of a channel requests the administrator to audit the channel, all messages sent through the channel will be stored in a database. With this function enabled, the owner of the channel will be able to scan the history of messages sent as well as retrieve individual messages. A special audit member and audit channel will be created for that purpose, with the owner and audit member as the only subscribers. Possible criteria for selecting messages based on the history of the channel are: the time period during which a message was sent, a member who sent a message, the format of a message, and others.

2) Validation: This extension enables validating the format of messages sent through a channel. Initially, the owner of a channel asks the administrator to validate the messages posted to this channel, including the format definition document with the request. As messages are written with XML syntax, one of several languages for defining XML instance languages may be used, such as DTD, XML Schema, Relax NG or others. All messages that do not confirm to a given format would be returned to the sender with an error status or audited for later inspection

3) Transformation: This extension enables the transformation of messages sent through a channel from one format to another. Initially, the owner of the channel requests the administrator to transform messages passed through this channel. The request specifies the transformation to be carried out. As all messages use XML syntax, the transformation is typically expressed as an XSLT template, but other DOM- or SAX-compliant transformations could be used as well. Different transformations may be applied depending on the member posting a message. While passing messages through the channel, the transformation is carried out before the message is delivered to the channel subscribers.

4) Composition: This extension enables the composition of existing channels into new, more complex channels. Some operations to carry out composition include: a. Linking: The exit-point of one channel (source) is joined with the entry-point of another channel (target). As a result, all messages received from the source channel are automatically forwarded to the target channel. b. Splitting: The exit-point of the source channel is joined with the entry-points of all target channels. This is similar to the linking operation but there are several target channels. As a result, all messages received from the source channel are automatically forwarded to all target channels. c. Joining: The exit-points of all source channels are joined with the entry-point of a target channel, symmetrically to the splitting operation. All messages passing through any of the source channels are forwarded to the target channel. d. Filtering: This is a linked channel structure, with a filter inserted between a source and target channels to decide which messages received from the former channel will be forwarded to the latter channel. Other messages are discarded. e. Routing: This is a split channel structure with a router inserted between the source and target channels. For each message received from the source, the router selects the target channel to be used to forward this message on the basis of a state maintained by the router or perhaps the message itself.

5) Tracking: This extension enables to determine the position of a message in the system while in transit from the sender to receivers. Initially, the owner of a channel requests the administrator to add the tracking feature to the channel. Once enabled, any member posting a message to the channel will be able to track its own messages, while the owner will be able to track all messages posted to the channel. Tacking will take place by a member issuing requests to the special tracking member through the designated tracking channel.

6) Encryption: This extension is responsible for transforming messages in transit through a channel to a secret, not legible code. Encryption is requested by the channel owner, specifying the encryption algorithm to be used. Messages passing through an encrypted channel cannot be interpreted without knowing the decryption method. Messages are decrypted or transformed to a legible code before being delivered to the subscribers.

7) Authentication: This extension helps to assure the identity of the subscribers of a channel using digital signatures, assuring the integrity, confidentiality and nonrepudiation of the messages passing through the channel. Once an owner asks the administrator to authenticate the channel, all subscribers will be required to provide their digital signatures issued by a Certification Authority. Thereafter, every message posted to a channel will contain the digital signature of the sender.

8) Mailboxes: By default, each member is provided with a single mailbox to store all incoming messages and a single mailbox to store all outgoing messages from all subscribed channels. In order to facilitate the processing of messages from heavy traffic channels, a member can ask the owner of a channel to provide a dedicated pair of mailboxes for that particular channel. The request would be forwarded to the administrator. It successful, the member would be assigned a dedicated inbox to contain the messages received from the heavy-traffic channel and a dedicated outbox to store the messages delivered to this channel.

9) Alliances: This extension enables the creation and management of member alliances. A member alliance is a group of members that decide to act together to carry out messaging on particular channels. A member may carry out messaging individually and at the same time belong to one or more member alliances. In addition to its regular members, the alliance includes a member who acts as its supervisor. The supervisor ensures that the whole member alliance appears to other members in exactly the same way as a single regular member. The supervisor acts on behalf of the whole alliance to add or remove regular members, to create or destroy channels, to subscribe or unsubscribe channels, and to send and receive messages. The alliance exists as long as its supervisor does.

10) Directories: This extension enables members to find existing channels and registered members though a directory service. The service includes white and yellow pages. White pages allow finding the unique identifier of a member or a channel through their names, provided during respectively member registration or channel creation. Yellow pages allow finding all members or channels that specialize in a particular topic, supported through the topics extension and specified during member registration of channel creation.

11) Topics: This extension maintains a hierarchy of topics that can be used to describe the interest of the members and the content of the channels. During registration, a member may choose any number of topics to describe itself. Likewise, during channel creation a member may choose any number of topics to describe the content of the channel. Each topic is selected from the tree-like hierarchy of topics related using ancestor and descendant relationships. Members can use the topics defined in the hierarchy to describe themselves and to search for other members through yellows pages. They can also modify the hierarchy, by adding and removing topics, and adding and removing relationships between topics.

12) Persistence: The core framework realizes the push model for message delivery - all messages posted to a channel are automatically pushed into the inboxes of all channel subscribers. This extension enables an alternative pop model - messages posted to a channel are stored in memory until explicitly requested by each subscriber. In this way, a member can retrieve both new and historical messages on demand. While push channels are created by default, this extension allows choosing the delivery mode (push or pop) during channel creation. Not only different channels may realize different delivery modes, the owner of a channel may request the administrator to change the mode at run-time. All extensions should be implemented using the features and components implemented in the core framework. Moreover, it is crucial that all extensions conservatively enrich the whole messaging framework, so that the core behaviour and properties are maintained regardless of the presence of the extra functionality.

Java Message Service

For this research framework, the Quid client JMS interface is used to connect to the Quid Broker. The Java Message Service (JMS) is de facto standard Java API provided for messaging services [62]. This API is a common set of interfaces and associated semantics that allows Java applications to create, send, receive, and read messages. The JMS API is a set of Java interfaces that defines a set of unimplemented Java methods, and the implementation is vendor-specific. The JMS API enables loosely coupled, asynchronous and reliable messaging.

The JMS API defines how publishers can generate and send messages to the JMS server. Similarly, for subscribers, the JMS API defines the reception of these messages from the JMS server. The API also provides abstract methods to control the message flow by various message-filtering options. A JMS application is composed of the following parts:

- **A JMS provider:** A messaging system that implements the JMS interfaces and provides administrative and control features. This is a messaging system that implements JMS in addition to the other administrative and control functionality required of a full- featured messaging product.
- **JMS clients:** Programs or components that produce and consume messages.
- **Messages:** Objects that communicate information between JMS clients. Each application defines a set of messages that are used to communicate information between its clients. The JMS API does not address the following functionality:
- Load Balancing/Fault Tolerance:

The JMS API does not specify how multiple, cooperating clients implementing a critical service can cooperate and appear to be a single, unified service.

- Error/Advisory Notification:

JMS does not define system messages that provide asynchronous notification of problems or system events to clients. By following the guidelines defined by JMS, clients can avoid using these messages and thus prevent the portability problems their use introduces.

- Administration:

JMS does not provide an API for administering components in the messaging system.

- Security: JMS does not provide an API for controlling the privacy and integrity of messages.
- Wire Protocol: JMS does not define how messages are sent over the network between the components of the messaging system.
- Message Type Repository: JMS does not define a repository for storing message type definitions and it does not define a language for creating message type definitions. In this research, the JMS Qpid client is used to provide a JMS interface to the publishers and subscribers. The Qpid Java-broker is used as the message-broker, and communicates with the Qpid clients using Advanced Message Queuing Protocol (AMQP).

The AMQ Protocol

Messaging APIs (usually) provide a programming language specific interface to build MOM applications. The API inherently limits the interoperability between publish subscribe clients. between the MOM system components. The use of a de facto API and wire-level protocol can be combined to maximise the benefits of both, i.e. AMQP system components built using JMS and at the same time, multiple APIs in different programming languages can be supported too. A brief description of the AMQ Protocol (AMQP) is given here to help the reader understand the concepts behind the structures of a broker implementing AMQP, such as the Qpid broker used in this research framework. AMQP is an open, royalty-free and unpatented networking protocol for messaging middleware.

The AMQP Working Group aims to create a de facto standard protocol for MOM that allows the business applications to interact encouraging the ideas of partnership and collaboration. Initially, John O’Hara, from J. P. Morgan, created AMQP circa 2003. imartis joined them soon after and the working group has been growing since then. Implementations of the protocol were first deployed in 2006 and AMQP now aspires to become an IETF standard the specifications of AMQP define both a binary network wire-level protocol (AMQP) and the semantics of broker services (AMQP Model) in order to enable technology-neutral interoperability. AMQP is a networking protocol that enables full functional interoperability between conforming clients, i.e. the publishers and subscribers, and conforming messaging middleware servers, i.e. the brokers. AMQP defines both the network protocol and the broker services through:

- A defined set of messaging capabilities defined by the AMQP Model. The AMQP Model consists of a set of components that route and store messages within the broker, plus a set of rules for connecting these components together.
- A network wire-level protocol, AMQP, that lets client applications talk to the broker and interact with the AMQP Model it implements. The Internet Assigned Numbers Authority (IANA) has assigned a port for AMQP, which is the port number 5672.

The AMQP Model explains the server semantics, and specifies a modular set of components and standard rules for connecting these:

- A middleware server is a data server that accepts messages, routes them to different consumers depending on arbitrary criteria, and buffers them in memory or on disk when consumers are not able to accept them fast enough. This is commonly known as the broker.
- A virtual host is a collection of exchanges, message queues and associated objects. Virtual hosts are independent server domains that share a common authentication and encryption environment. Each connection must be associated with a single virtual host. There can be one or more middleware servers in the same machine, called virtual hosts.
- The exchange is used to receive messages from publisher applications and to route these two message queues using arbitrary criteria called bindings. Bindings are usually message properties, its header fields or its body content. Exchanges do not store messages.
- The message queue, which stores messages either in memory, on disk or some combination of these until they can be safely processed by a consuming client application. Message queues have a name agreed beforehand by applications that share them, i.e. publishers and subscribers have to agree on a queue name before sending messages. Message queues track message acquisition. Messages must be acquired before being dequeued. This prevents multiple clients from acquiring and then consuming the same message simultaneously. Hence each subscriber will have its own queue for each subscription.

The binding, which defines the relationship between a message queue and an exchange and provides the message routing criteria. Since AMQP specifies a standard model for message brokers, this would mean that the bottleneck models develop in this thesis could be applied to message brokers that follow the AMQP Model. As a specification, AMQP is also platform-agnostic and language-agnostic. Different implementations running on different platforms should be able to interoperate seamlessly, depending on how closely they follow the AMQP specifications.

We expect the output of this research would primarily be used in enhancing reliability and building resilient MOMs, hence we favour AMQP over JMS.

Since AMQP specifies a standard model for message brokers, this would mean that the bottleneck models develop in this thesis could be applied to message brokers that follow the AMQP Model. As a specification, AMQP is also platform-agnostic and language-agnostic. Different implementations running on different platforms should be able to interoperate seamlessly, depending on how closely they follow the AMQP specifications. We expect the output of this research would primarily be used in enhancing reliability and building resilient MOMs, hence we favour AMQP over JMS.

Apache Qpid

In the broker bottleneck experiments explained later in this report (Section 3.5), Apache Qpid is used as a PS-MOM. Qpid is an open-source MOM broker made by the Apache Foundation. Qpid implements the AMQP specifications 0.9 and 0.10 [66]. The Qpid software package differentiates between the Qpid brokers and Qpid clients. The Qpid Broker is the MOM broker that uses AMQP as its wire level protocol, i.e. the messages are transmitted between the Qpid Broker and the Qpid clients using the message formats defined by AMQP. Any client that communicates using AMQP can connect to the Qpid Broker directly. The Qpid client refers to an API that could be used by application developers. The Qpid client offers bindings between AMQP and other languages and APIs such as C++, JMS, and Python. This negates the need for application developers to recreate code that reads and writes AMQP.

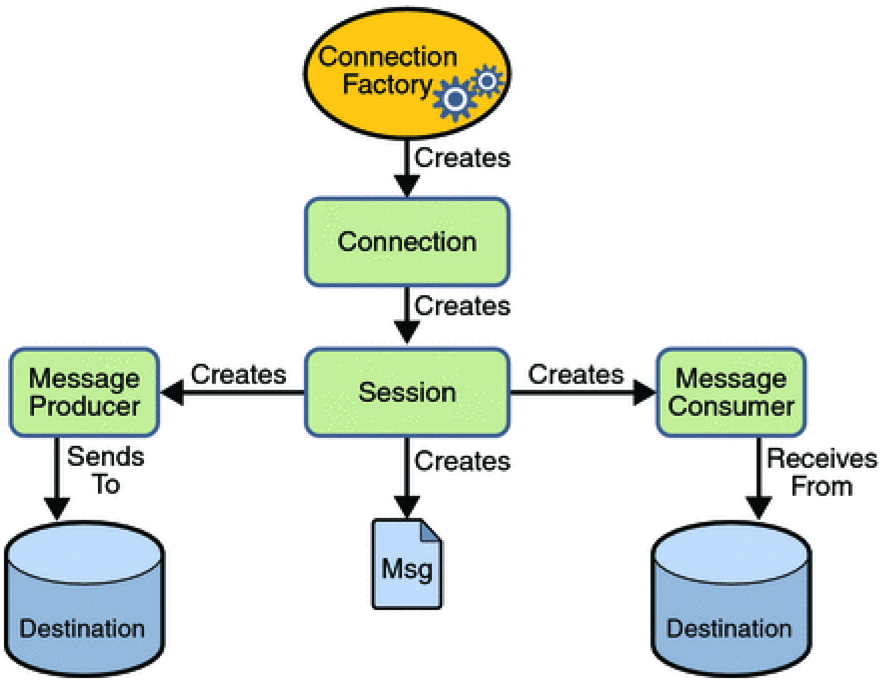
Other major AMQP MOM brokers include the following:

- **OpenAMQ** is a message broker and a set of client libraries for C/C++ and JMS that implement AMQP. It is the first AMQP implementation and it was developed by iMatix, a member of the AMQP Working Group. However, this product has been discontinued by iMatix after their switch to ZeroMQ (stylised as ØMQ). ZeroMQ is an open messaging implementation that runs without a message broker. It extends the standard socket interfaces enabling them to send and receive messages asynchronously. It supports different messaging patterns (point-to-point, publish-subscribe, request-reply, push-pull and exclusive pair).

- **RabbitMQ,** formerly by Rabbit Technologies, now SpringSource, a division of VMware, is an open source enterprise messaging system that implements AMQP. It is based on the reliable, available and scalable Open Telecom Platform (OTP), written in Erlang and is used to manage flexible switching exchanges that are designed to never go down and handle high loads. RabbitMQ is an AMQP layer on top of OTP.

- **Red Hat Enterprise MRG** is a distributed IT infrastructure that includes three technologies, MRG Messaging, MRG Realtime and MRG Grid. MRG Messaging is based on Apache Qpid but includes persistence, additional components, Linux kernel optimisations and more operating system services. It supports a variety of messaging paradigms such as store-and-forward, distributed transactions, publish/subscribe, content-based routing, queued file transfer, point-to-point connections among peers and market data distribution.
- **Active MQ** is an open source product maintained by the Apache software foundation. It has multiple advanced features that support an extensive range of different requirements, protocols, and network topologies. At the time of writing, ActiveMQ version 4.x does not support AMQP. ActiveMQ supports AMQP from version 5.8 onwards

JMS Programming Model



JMS Queue Example

To develop JMS queue example, you need to install any application server. Here, we are using **glassfish3** server where we are creating two JNDI.

1. Create connection factory named **myQueueConnectionFactory**
2. Create destination resource named **myQueue**

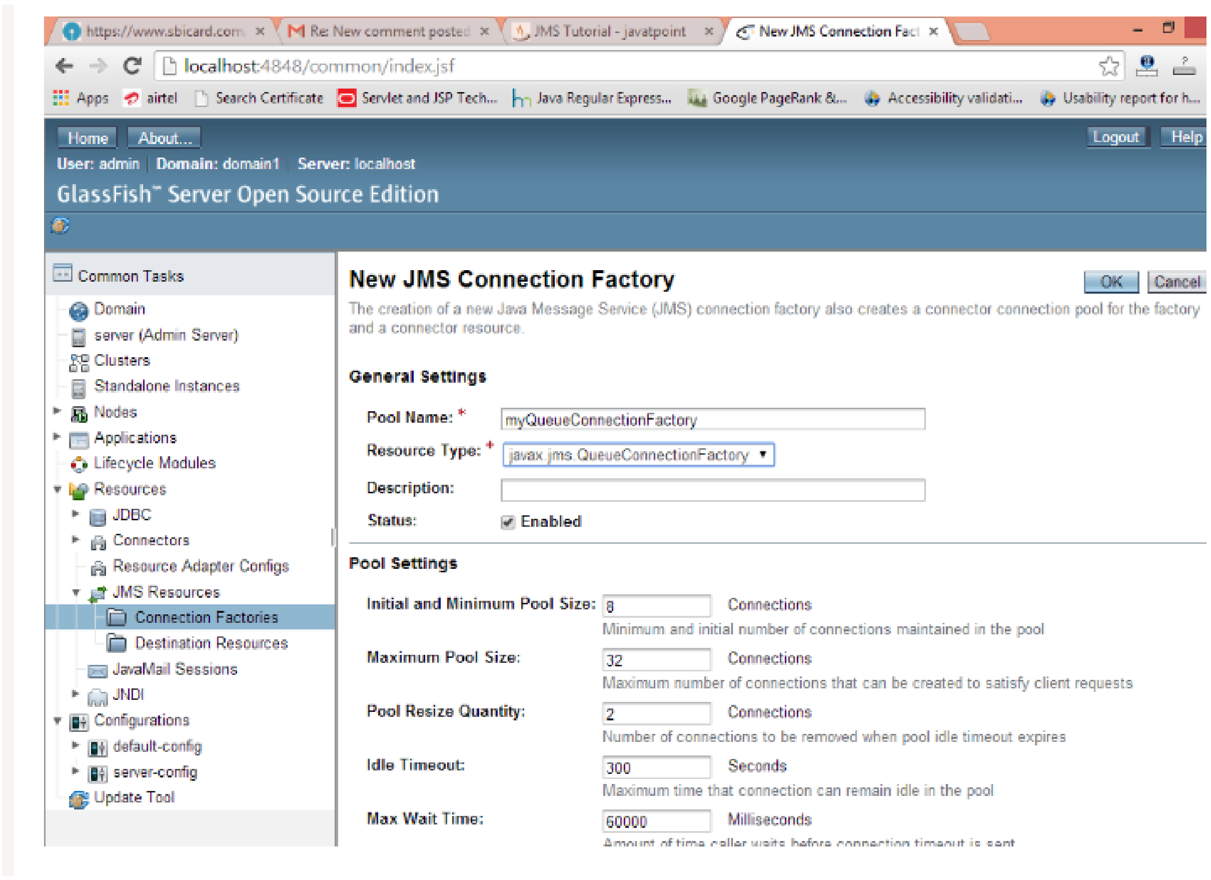
After creating JNDI, create server and receiver application. You need to run server and receiver in different console. Here, we are using eclipse IDE, it is opened in different console by default.

1) Create connection factory and destination resource

Open server admin console by the URL **http://localhost:4848**

Login with the username and password.

Click on the **JMS Resource -> Connection Factories -> New**, now write the pool name and select the Resource Type as QueueConnectionFactory then click on ok button.



New JMS Connection Factory

The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the factory and a connector resource.

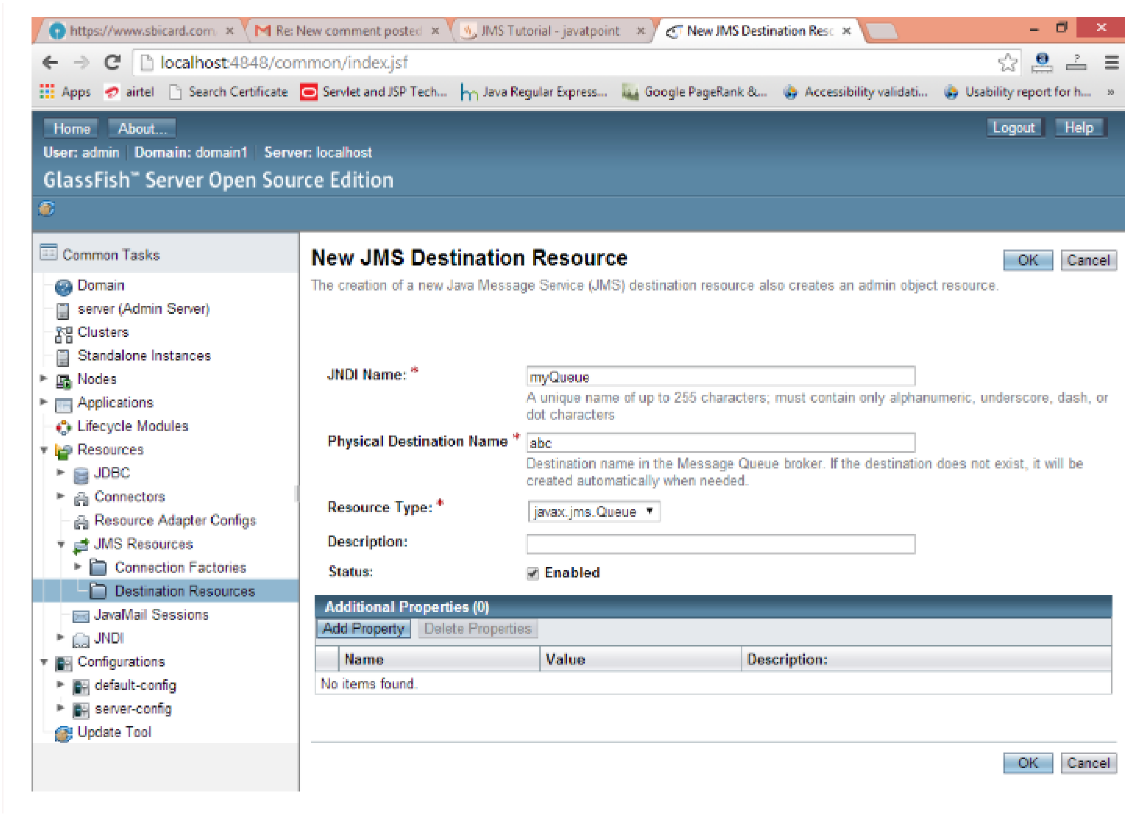
General Settings

Pool Name: *	myQueueConnectionFactory
Resource Type: *	javax.jms.QueueConnectionFactory
Description:	
Status:	✓ Enabled

Pool Settings

Initial and Minimum Pool Size:	8	Connections Minimum and initial number of connections maintained in the pool
Maximum Pool Size:	32	Connections Maximum number of connections that can be created to satisfy client requests
Pool Resize Quantity:	2	Connections Number of connections to be removed when pool idle timeout expires
Idle Timeout:	300	Seconds Maximum time that connection can remain idle in the pool
Max Wait Time:	60000	Milliseconds Amount of time caller waits before connection timeout is sent

Click on the JMS Resource -> Destination Resources -> New**, now write the JNDI name and physical destination name then click on ok button.**



New JMS Destination Resource

The creation of a new Java Message Service (JMS) destination resource also creates an admin object resource.

JNDI Name: *	myQueue A unique name of up to 255 characters; must contain only alphanumeric, underscore, dash, or dot characters
Physical Destination Name *	abc Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.
Resource Type: *	javax.jms.Queue
Description:	
Status:	✓ Enabled

Additional Properties (0)

Name	Value	Description:
No items found.		

2) Create sender and receiver application

Let's see the Sender and Receiver code. Note that Receiver is attached with listener which will be invoked when user sends message.

File: MySender.java

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import javax.naming.*;
import javax.jms.*;

public class MySender {
    public static void main(String[] args) {
        try
        {   //Create and start connection
            InitialContext ctx=new InitialContext();
            QueueConnectionFactory f=(QueueConnectionFactory)ctx.lookup("myQueueConnectionFactory");
            QueueConnection con=f.createQueueConnection();
            con.start();
            //2) create queue session
            QueueSession ses=con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            //3) get the Queue object
            Queue t=(Queue)ctx.lookup("myQueue");
            //4)create QueueSender object
            QueueSender sender=ses.createSender(t);
            //5) create TextMessage object
            TextMessage msg=ses.createTextMessage();

            //6) write message
            BufferedReader b=new BufferedReader(new InputStreamReader(System.in));
            while(true)
            {
                System.out.println("Enter Msg, end to terminate:");
                String s=b.readLine();
                if (s.equals("end"))
                    break;
                msg.setText(s);
                //7) send message
                sender.send(msg);
                System.out.println("Message successfully sent.");
            }
            //8) connection close
            con.close();
        }catch(Exception e){System.out.println(e);}
    }
```

```java
}
```

File: MyReceiver.java

```java
import javax.jms.*;
import javax.naming.InitialContext;

public class MyReceiver {
    public static void main(String[] args) {
        try{
            //1) Create and start connection
            InitialContext ctx=new InitialContext();
            QueueConnectionFactory f=(QueueConnectionFactory)ctx.lookup("myQueueConnectionFactory");
            QueueConnection con=f.createQueueConnection();
            con.start();
            //2) create Queue session
            QueueSession ses=con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            //3) get the Queue object
            Queue t=(Queue)ctx.lookup("myQueue");
            //4)create QueueReceiver
            QueueReceiver receiver=ses.createReceiver(t);

            //5) create listener object
            MyListener listener=new MyListener();

            //6) register the listener object with receiver
            receiver.setMessageListener(listener);

            System.out.println("Receiver1 is ready, waiting for messages...");
            System.out.println("press Ctrl+c to shutdown...");
            while(true){
                Thread.sleep(1000);
            }
        }catch(Exception e){System.out.println(e);}
    }

}
```

File: MyListener.java

```java
import javax.jms.*;
public class MyListener implements MessageListener {

    public void onMessage(Message m) {
        try{
        TextMessage msg=(TextMessage)m;

        System.out.println("following message is received:"+msg.getText());
        }catch(JMSException e){System.out.println(e);}
    }
}
```

JMS Topic Example

It is same as JMS Queue, but you need to change Queue to Topic, Sender to Publisher and Receiver to Subscriber.

You need to create 2 JNDI named **myTopicConnectionFactory** and **myTopic**.

File: MySender.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import javax.naming.*;
import javax.jms.*;

public class MySender {
    public static void main(String[] args) {
        try
        {   //Create and start connection
            InitialContext ctx=new InitialContext();
            TopicConnectionFactory f=(TopicConnectionFactory)ctx.lookup("myTopicConnectionFactory");

            TopicConnection con=f.createTopicConnection();
            con.start();
            //2) create queue session
            TopicSession ses=con.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
            //3) get the Topic object
            Topic t=(Topic)ctx.lookup("myTopic");
            //4)create TopicPublisher object
            TopicPublisher publisher=ses.createPublisher(t);
            //5) create TextMessage object
            TextMessage msg=ses.createTextMessage();

            //6) write message
            BufferedReader b=new BufferedReader(new InputStreamReader(System.in));
            while(true)
            {
                System.out.println("Enter Msg, end to terminate:");
                String s=b.readLine();
                if (s.equals("end"))
                    break;
                msg.setText(s);
                //7) send message
```

```java
            publisher.publish(msg);
            System.out.println("Message successfully sent.");
        }
        //8) connection close
        con.close();
    }catch(Exception e){System.out.println(e);}
  }
}
```

File: MyReceiver.java

```java
import javax.jms.*;
import javax.naming.InitialContext;

public class MyReceiver {
    public static void main(String[] args) {
        try {
            //1) Create and start connection
            InitialContext ctx=new InitialContext();
            TopicConnectionFactory f=(TopicConnectionFactory)ctx.lookup("myTopicConnectionFactory");

            TopicConnection con=f.createTopicConnection();
            con.start();
            //2) create topic session
            TopicSession ses=con.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
            //3) get the Topic object
            Topic t=(Topic)ctx.lookup("myTopic");
            //4)create TopicSubscriber
            TopicSubscriber receiver=ses.createSubscriber(t);

            //5) create listener object
            MyListener listener=new MyListener();

            //6) register the listener object with subscriber
            receiver.setMessageListener(listener);

            System.out.println("Subscriber1 is ready, waiting for messages...");
            System.out.println("press Ctrl+c to shutdown...");
            while(true){
                Thread.sleep(1000);
            }
        }catch(Exception e){System.out.println(e);}
    }

}
```

File: MyListener.java

```
import javax.jms.*;
public class MyListener implements MessageListener {

   public void onMessage(Message m) {
      try{
      TextMessage msg=(TextMessage)m;

      System.out.println("following message is received:"+msg.getText());
      }catch(JMSException e){System.out.println(e);}
   }
}
```

Common MOM Services

When constructing large-scale systems, it is vital to utilize a state-of-the-art enterpriselevel MOM implementation. Enterprise-level messaging platforms will usually come with a number of built-in services for transactional messaging, reliable message delivery, load balancing, and clustering; this section will now give an overview of these services.

Message Filtering

Message filtering allows a message consumer/receiver to be selective about the messages it receives from a channel. Filtering can operate on a number of different levels. Filters use Boolean logic expressions to declare messages of interest to the client, the exact format of the expression depends on the implementation but the WHERE clauses of SQL-92 (or a subset of) is commonly used as the syntax. Filtering models commonly operate on the properties (name/value pairs) of a message; however, a number of projects have extended filtering to message payloads. Message filtering is covered further in the Java Message Service section of this chapter.
Since there are a number of filter capabilities found in messaging systems, it is useful to note that as the filtering techniques get more advanced, they are able replicate the techniques that proceed them. For example, subject-based filtering is able to replicate channel-based filtering, just like content-based filtering is able to replicate both subject and channel-based filtering.

Transactions

Transactions provide the ability to group tasks together into a single unit of work. The most basic straightforward definition of a transaction is as follows:

In order for transactions to be effective, they must conform to the following properties commonly referred to as the **ACID** transaction properties.
In the context of transactions, any asset that will be updated by a task within the transaction is referred to as a resource. A resource is a persistent store of data that is

Filter type	Description
Channel-based	Channel-based systems categorize events into predefined groups. Consumers subscribe to the groups of interest and receive all messages sent to the groups
Subject-based	Messages are enhanced with a tag describing their subject. Subscribers can declare their interests in these subjects flexibly by using a string pattern match on the subject, for example, all messages with a subject starting of "Car for Sale"
Content-based	As an attempt to overcome the limitations on subscription declarations, content-based filtering allows subscribers to use flexible querying languages in order to declare their interests with respect to the contents of the messages. For example, such a query could be giving the price of stock 'SUN' when the volume is over 10,000. Such a query in SQL-92 would be "stock_symbol = 'SUN' AND stock_volume >10,000"
Content-based with Patterns (Composite Events)	Content-based filtering with patterns, also known as *composite events* [10], enhances content-based filtering with additional functionality for expressing user interests across multiple messages. Such a query could be giving the price of stock 'SUN' when the price of stock 'Microsoft' is less than $50

The properties of a transaction

Atomic	All tasks must complete, or no tasks must complete
Consistent	Given an initial consistent state, a final consistent state will be reached regardless of the result of the transaction (success/fail)
Isolated	Transactions must be executed in isolation and cannot interfere with other concurrent transactions
Durable	The effect of a committed transaction will not be lost subsequent to a provider/broker failure

participating in a transaction that will be updated; a message broker's persistent message store is a resource. A resource manager controls the resource(s); they are responsible for managing the resource state.

MOM has the ability to include a message being sent or received within a transaction. This section examines the main types of transaction commonly found in MOM. When examining the transactional aspects of messaging systems, it is important to remember that MOM messages are autonomous self-contained entities.

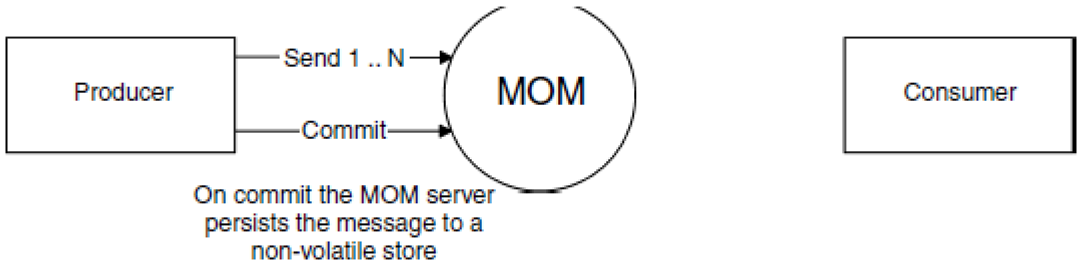
Within the messaging
domain, there are two common types of transactions, Local Transactions and Global Transactions. Local transactions take place within a single resource manager such as a single messaging broker. Global transactions involve multiple, potentially distributed heterogeneous resource managers with an external transaction manager coordinating the transaction.

Transactional Messaging

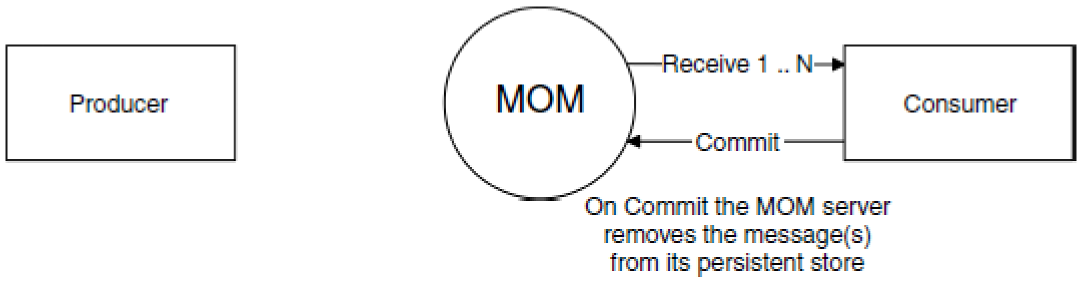
When a client wants to send or retrieve messages within a transaction, this is referred to as Transactional Messaging. Transactional messaging is used when you want to perform several messaging tasks (send 1-N messages) in a way that all tasks will succeed or all will fail. When transactional messaging is used, the sending or receiving application has the opportunity to commit the transaction (all the operations have succeeded), or to abort the transaction (one of the operations failed) so all changes are rolled back. If a transaction is aborted, all operations are rolled back to the state when the transaction was invoked.

Messages delivered to the server in a transaction are not forwarded on to the receiving client until the sending client commits the transaction. Transactions may contain multiple messages. Message transactions may also take place in transactional queues. This type of queue is created for the specific purpose of receiving and processing messages that are sent as part of a transaction. Nontransactional queues are unable to process messages that have been included in a transaction.

Transaction Roles


Producer
Send 1 .. N
Commit
MOM
Consumer
On commit the MOM server persists the message to a non-volatile store

Role of a producer in a transaction


Producer
MOM
Receive 1 .. N
Commit
Consumer
On Commit the MOM server removes the message(s) from its persistent store

Role of a consumer in a transaction

Reliable Message Delivery

A MOM service will typically allow the configuration of the Quality-of-Service (QoS) delivery semantics for a message. Typically, it is possible to configure a message delivery to be of *at-most once*, *at-least-once*, or *once-and-once-only*. Message acknowledgment can be configured, in addition to the number of retry attempted on a delivery failure. With persistent asynchronous communication, the message is sent to the messaging service that stores it for as long as it takes to deliver the message, unless the *Time-to-Live* (TTL) of the message expires.

Guaranteed Message Delivery

In order for MOM platforms to guarantee message delivery, the platform must save all messages in a nonvolatile store such as a hard disk. The platform then sends the message to the consumer and waits for the consumer to confirm the delivery. If the consumer does not acknowledge the message within a reasonable amount of time, the server will resend the message. This allows for the message sender to "fire and forget" messages, trusting the MOM to handle the delivery. Certified message delivery is an extension of the guaranteed message delivery method.

Once a consumer has received the message, a
consumption report (receipt) is generated and sent to the message sender to confirm the consumption of the message.

Message Formats

Depending on the MOM implementation, a number of message formats may be available to the user. Some of the more common message types include Text (including XML), Object, Stream, HashMaps, Streaming Multimedia [13], and so on. MOM providers can provide mechanisms for transforming one message format into another and for transform ing/altering the format of the message payload; some MOM implementation allows XSL transformation to be carried out by an XML message payload. Such MOM providers are often referred to as Message brokers and are used to "broker" the difference between diverse systems.

Load Balancing

Load balancing is the process of spreading the workload of the system over a number of servers (in this scenario, a server can be defined as a physical hardware machine or software server instance or both). A correctly load balanced system should distribute work between servers, dynamically allocating work to the server with the lightest load. Two main approaches of load balancing exist, “push” and “pull”. In the push model, an algorithm is used to balance the load over multiple servers. Numerous algorithms exist, which attempt to guess the least-burdened and push the request to that server. The algorithm, in conjunction with load forecasting, may base its decision on the performance record of each of the participating servers or may guesstimate the leastburdened server. The push approach is an imperfect, but acceptable, solution to load balancing a system.

In the pull model, the load is balanced by placing incoming messages into a point-topoint queue, and the consuming servers can then pull messages from this queue at their own pace. This allows for true load balancing, as a server will only pull a message from the queue once they are capable of processing it. This provides the ideal mechanism as it more smoothly distributes the loads over the systems.

Clustering

In order to recover from a runtime server failure, the server’s state needs to be replicated across multiple servers. This allows a client to be transparently migrated to an alternative server, if the server it is interacting with fails. Clustering is the distribution of an application over multiple servers to scale beyond the limits, both performance and reliability, of a single server. When the limits of the server software or the physical limits of the hardware have been reached, the load must be spread over multiple servers or machines to scale the system further. Clustering allows us to seamlessly distribute over multiple servers/machines and still maintain a single logical entity and a single virtual interface to respond to the client requests. The grouping of clusters creates highly scalable and reliable deployments while minimizing the number of servers needed to cope with large workloads.

Service-Oriented Architectures

The problems and obstacles encountered during system integration pose major challenges for an organizations IT department:
“70% of the average IT department budget is devoted to data integration projects”–IDC
“PowerPoint engineers make integration look easy with lovely cones and colorful boxes”–Sean McGrath, CTO, Propylon
“A typical enterprise will devote 35%–40% of its programming budget to programs

whose purpose is solely to transfer information between different databases and legacy systems."–Gartner Group
Increasing pressure to cut the cost of software development is driving the emergence of open nonproprietary architectures to utilize the benefits of reusable software components. MOM has been used to create highly open and flexible systems that allow the seamless integration of subsystems. MOM solves many of the transport issues with integration. However, major problems still exist with the representation of data, its format, and structure. To develop a truly open system, MOM requires the assistance of additional technologies such as XML and Web Services. Both of these technologies provide a vital component in building an open cohesive system. Each of these technologies will be examined to highlight the capabilities they provide in the construction of open system architectures.

XML

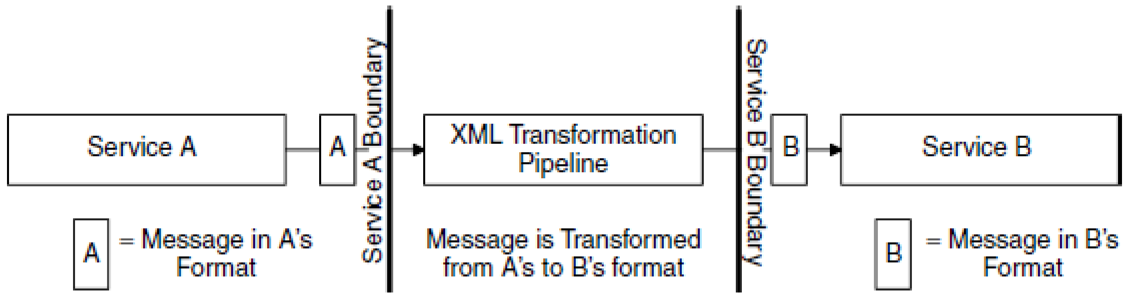
The eXtensible Mark-up Language (XML) provides a programming language and platform-independent format for representing data. When used to express the payload of a message, this format eliminates any networking, operating system, or platform binding that a binary proprietary protocol would use. XML provides a natural independent way of representing data. Once data is expressed in XML, it is trivial to change the format of the XML using techniques such as the eXtensible Stylesheet Language: Transformations (XSLT). In order for XML to be used as a message exchange medium, standard formats need to be defined to structure the XML messages. There are a number of bodies working on creating these standards such as ebXML and the OASIS Universal Business Language (UBL). These standards define a document layout format to provide a standard communicative medium for applications. With UBL, you convert your internal message formats to the standard UBL format and export to the external environment. To import messages, you mirror this process. An extensive examination of this process and relevant standards is outside the scope of this chapter.

Web Services

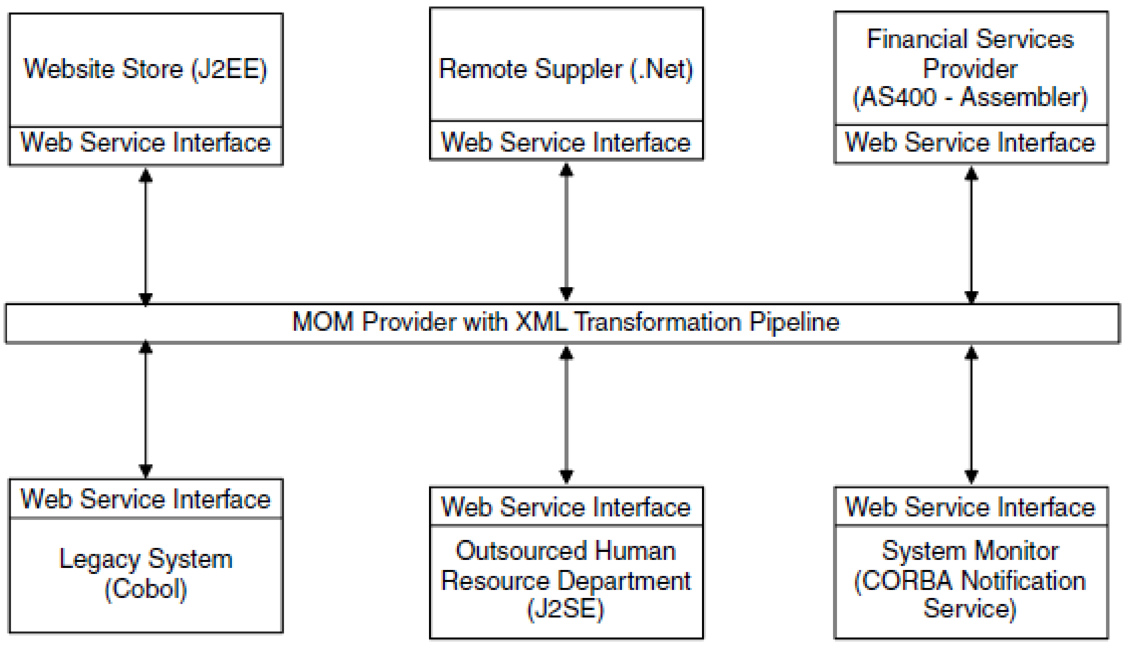
Web Services are platform- and language-independent standards defining protocols for heterogeneous system integration. In their most basic format, web services are interfaces that allow programs to run over public or private networks using standard protocols such as Simple Object Access Protocol (SOAP). They allow links to be created between systems without the need for massive reengineering. They can interact with and/or invoke one another, fulfilling tasks and requests that in turn carry out specific parts of complex transactions or workflows. Web services can be seen in a number of ways, such as a business-to-business/enterprise application integration tool or as a natural evolution of basic RPC mechanism. The key benefit of a web services deployment is that they act as a fac¸ade to the underlying language or platform, a web service written in C and running on Microsoft's Internet Information Server can access a web service written in Java running on BEA's Weblogic server.

Developing Service-Oriented Architectures

Through the combination of these technologies, we are able to create Service-Oriented Architectures (SOA). The fundamental design concept behind these architectures is to reduce application processing to logic black boxes. Interaction with these black boxes is achieved with the use of a standard XML message format; by defining the required XML input and output formats, we are able to create black box services. The service can now be accessed by transmitting the message over an asynchronous messaging channel.
With traditional API-based integration, the need to create adaptors and data format converters for each proprietary API is not a scalable solution. As the number of APIs increases, the adaptors and data converters required will scale geometrically. The important aspect of SOAs is their message-centric structure. Where message formats differ, XML-based integration can convert the message to and from the format required using an XML transformation pipeline, as is shown in Figure
In this approach, data transformation can be seen as just another assembly line problem, allowing services to be true black box components with the use of an XML-in and XMLout contract. Users of the services simply need to transform their data to the services contract XML format. Integration via SOA is significantly cheaper than integration via APIs; with transformations taking place outside of the applications, it is a very noninvasive method of integration. Service-Oriented Architecture with transforming XML is a new and fresh way of looking at Enterprise Application Integration (EAI) and distributed systems and a new way of looking at web services that are often touted as a replacement for traditional RPC. Viewed in this light, they are an evolution of the RPC mechanism but still suffer from many of its shortcomings.
With Service-Oriented Architectures, creating connections to trading partners and legacy systems should be as easy as connecting to an interdepartmental system. Once the initial infrastructure has been created for the architecture, the amount of effort to connect to further systems is minimal. This allows systems created with this framework to be highly dynamic, allowing new participants to easily join and leave the system.
Figure 1.13 illustrates an example SOA using the techniques advocated. In this deployment, the system has been created by interconnecting six subsystems and integrating.


Service A
A
Service A Boundary
XML Transformation Pipeline
Service B Boundary
B
Service B
A = Message in A's Format
Message is Transformed from A's to B's format
B = Message in B's Format

XML transformation pipeline



System deployed using web service, XML messages and MOM to create a SOA

them, each of the subsystems is built using a different technology for their primary implementation.

The challenges faced in this deployment are common challenges faced daily by system developers. When developing a new system it is rare for a development team not to have some form of legacy system to interact with. Legacy systems may contain irreplaceable business records, and losing this data could be catastrophic for an organization. It is also very common for legacy systems to contain invaluable business logic that is vital for an organization's day-to-day operation. It would be preferable to reuse this production code, potentially millions of lines, in our new system. Transforming a legacy system into an XML service provides an easy and flexible solution for legacy interaction. The same principle applies to all the other subsystems in the deployment, from the newly created J2EE Web Store, or the *Financial Services* provider running on an AS400 to the *Remote Suppliers* running the latest Microsoft .NET systems. Each of these proprietary solutions can be transformed into a service and join the SOA. Once a subsystem has been changed into a service, it can easily be added and removed from the architecture. SOAs facilitate the construction of highly dynamic systems, allowing functionality (services) such as payroll, accounting, sales, system monitors, and so on, to be easily added and removed at run time without interruptions to the overall system. The key to developing a first-rate SOA is to interconnect services with a MOM-based communication; MOM utilization will promote loose coupling, flexibility, reliability, scalability, and high-performance characteristics in the overall system architecture.

$$\frac{\text{XML} + \text{Web Services} + \text{MOM}}{\text{Service Oriented Architecture}} = \text{Open Systems}$$

REFERENCES

[1] **Qusay I. Sarhan**University of Duhok and **Idrees S. Gawdan**Duhok Polytechnic University

https://sjuoz.uoz.edu.krd/index.php/sjuoz/article/view/438

[2] Guruduth Banavar Tushar Chandra Robert Strom and Daniel Sturman

DISC 1999: Distributed Computing pp 1-17|

https://link.springer.com/chapter/10.1007/3-540-48169-9_1

[3] **Thomas Rausch** 5.81 TU Wien, **Schahram Dustdar** 40.22 TU Wien

https://www.researchgate.net/profile/R_Ranjan

[4] M. Saranya Research Scholar, Department of Computer Science Department of Computer Science Muthayammal College of Arts & Science, Namakkal, Tamilnadu.

A. Anusha Priya Associate Professor, Department of Computer Science Department of Computer Science Muthayammal College of Arts & Science, Namakkal, Tamilnadu.

http://www.ijirst.org/articles/IJIRSTV4I3015.pdf

[5] https://docs.oracle.com/cd/E19340-01/820-6424/aeraq/index.html

[6] Edward Curry National University of Ireland, Galway, Ireland

https://pdfs.semanticscholar.org/98e1/90fd2ed9e15514f7f5d5ea3dbd8aeb382a9c.pdf

[7] **Published in:** Proceedings. IEEE International Conference on Web Services, 2004.

Print ISBN: 0-7695-2167-3 **INSPEC Accession Number:** 8205453

https://ieeexplore.ieee.org/abstract/document/1314778

[8] Piyush Maheshwari, Trung Nguyen Kien and Abdelkarim Erradi

WISE 2004: Web Information Systems – WISE 2004 Workshops pp 241-251|

https://link.springer.com/chapter/10.1007/978-3-540-30481-4_24

[9] https://patentimages.storage.googleapis.com/fa/11/d0/c827616397bc7f/US7512668.p df

[10] Andreas Wolber and Bernd Follmeg https://patents.google.com/patent/US8626878B2/en