İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

A JMS BASED NEWS FEEDER USING THE PUBLISH / SUBSCRIBE SCHEME

M.Sc. Thesis by

Gökhan BÜYÜKYUMUKOĞLU, B.Sc.

Department : Computer Engineering

Programme : Computer Engineering

JUNE 2007

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

A JMS BASED NEWS FEEDER USING THE PUBLISH / SUBSCRIBE SCHEME

M.Sc. Thesis by

Gökhan Büyükyumukoğlu, B.Sc.
(504031541)

Date of submission : 02 July 2007
Date of defence examination : 11 June 2007

Supervisor : Prof. Nadia Erdoğan

Members of the Examining Committee : Prof. Coşkun Sönmez

Assoc. Prof. A. Şima Etaner-Uyar

JUNE 2007

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

YAYINLAYICI / ABONE KALIBINI KULLANAN JMS TABANLI HABER SAĞLAYICISI

YÜKSEK LİSANS TEZİ

Müh. Gökhan Büyükyumukoğlu
(504031541)

Tezin enstitüye verildiği tarih : 02 Temmuz 2007
Tezin savunulduğu tarih : 11 Haziran 2007

Tez Danışmanı : Prof. Dr. Nadia Erdoğan

Diğer Jüri Üyeleri : Prof. Dr. Coşkun Sönmez

Yrd. Doç. Dr. A. Şima Etaner-Uyar

HAZİRAN 2007

ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Nadia Erdoğan for pushing me through the research and typing of my thesis and encouraging me especially while I was abroad.

Special thanks to my family for their love and patience.

May 2007 **Gökhan Büyükyumukoğlu**

CONTENTS

LIST OF TABLES

LIST OF FIGURES

YAYINLAYICI / ABONE KALIBINI KULLANAN JMS TABANLI HABER SAĞLAYICISI

ÖZET

Bu çalışmada 10 yıl öncesinin çok okuyuculu fakat az içerik sağlayıcılı internet uygulamalarından günümüzde popülerleşen çok kişinin içerik sağladığı ve yüksek miktardaki bilgi sayesinde okuyucunun azaldığı web uygulamalarına geçişte kullanılabilecek ve önem kazanan teknolojilerden mesajlaşma üzerine yoğunlaşıldı.

Günümüzde çok sık kullandığımız internet tarayıcısı gibi araçlar artık bazı konularda ihtiyaçlarımızı karşılamamaya başladı. Bilgiye ulaşmak için sürekli kullanıcının yenile tuşuna bastığı bir uygulamadan ziyade bilginin güncellendiği anda kullanıcıya kendiliğinden ulaştığı bir ortama ihtiyaç artmakta. Aynı zamanda sadece güncellenen bilginin bilgisayar ağı üzerinden iletimi, ağ kullanım trafiğinde de rahatlamaya yol açacakken günümüzde bir internet sayfasının güncellenmesi sırasında tüm internet sayfasının tekrardan kullanıcıya iletilmesi söz konusu. Bilginin sadece o bilgiye ulaşmak isteyen ve ilgisini belli etmiş kullanıcılara ulaştırılması ise ele alınması gereken ayrı bir konu. Bu çalışmada tüm bu koşulları sağlayan bir haber yayınlayıcı uygulaması geliştirilmiştir.

Geliştirilen JMS tabanlı uygulama yayınla / abone ol mesajlaşma kalıbını kullanmaktadır. Uygulama, yönetici ve istemci olmak üzere iki modülden oluşmaktadır. Yönetici modülü ile JMS sunucusu yönetilebilir, JMS sunucusu üzerinde haber konuları gibi nesneler yaratılabilir, haber mesajları veya internet linkleri yayınlanabilir. Yayınlanan mesajların içeriği hakkında bilgi veren anahtar kelimeler belirtilip istemcilerin mesajlar içinde filitreleme yapmaları sağlanabilir. İstemci modülü ile ise, istenilen haber konularına abone olunup yayınlanan mesajlar okunabilir, ilgili mesajların içerikleri üzerinden filitreleme yapılabilir. Geliştirilen uygulama Java programlama dili ile geliştirilmiş ve nesneye dayalı programlama metodolojileri kullanılmıştır. Geliştirme ortamı olarak Eclipse ve JMS sunucusu olarak Sonic JMS sunucusu seçilmiştir.

A JMS BASED NEWS FEEDER USING THE PUBLISH / SUBSCRIBE SCHEME

SUMMARY

Our research was focused on emerging technologies that would help to catch the shift from the old styled mass-reading with few publishing to the many publishing with narrowband reading web applications.

Web browsers which are the most popular and frequent used application in our daily lives have some short commings nowadays. Instead of a user who is continously hitting the refresh key of a browser, there is a need for an environment where the information is pushed to the user's application at the moment it is produced. At the same time another important issue is that only the newly produced information should be transmitted to the user's computer over the computer network. Today when a user refreshes his browser to update a web page, the whole document is actually sent over the network with the new updates to the page. The network traffic could be reduced if partial data could be sent. Another issue is that the information should only be sent to the interested users. In this thesis an application which has all these capabilities is developed.

The application is a JMS based news feeder software using the publish / subscribe scheme. Our software has two modules which are the administrator and the client. While the administrator module can manage the JMS server, managed objects on the server like news topics and publish news messages or web hyperlinks with associated keywords for filtering, the clients can create durable subscriptions to these topics, receive messages and filter received messages based on message properties. All software were developed using the Java programming language and object oriented methodologies. Eclipse was used as the development environment and Sonic JMS Server as the JMS implementation.

1. INTRODUCTION

Today the internet's primary use is still human to computer communication where users request information such as web pages or e-mails from specific servers with the use of tools like web browsers such as firefox or mail clients like outlook. This type of behavior can be categorized as the first web. Its main focus was the ability of publishing documents which was quite revolutionary ten years ago. Each document had a unique identifier or address called the Uniform Resource Locator (URL) which we used to reach the document and view it through the standard application web browser. This traditional view presents documents or content as a rather static good [3]. This concept was about mass reading. On the other hand publishers were few. Thousands of people were getting content from the few main web sites.

For simple applications such as catalog browsing the first web's model was sufficient. However there are new types of applications which are emerging such as news services, auction trackers, traffic information systems and online decision support systems. These application are information and event driven and rely on up to date and timely view on dynamic content that is likely to change rapidly and unpredictably. Providing this dynamic information to consumers who require up to date views of content places a big burden on the consumer's side especially for mobile clients which lack resources and have limited connectivity [3].

In the internet world there is an ongoing transformation that we can not recognize all the time, bit by bit, component by component and day by day. This change is not easily seen and for some people its hardly noticeable. We have to look at a large period of time to understand how important these changes are similarly to the importance of the first web's birth ten years ago.

The next generation of the internet will focus more on mass publishing with narrowband reading. While many people publishing their documents or content easily on the internet with little efforts, only a few who are interested will be accessing this content which makes for narrowband reading. The building blocks of this web, which are applications, and developer tools are slowly forming and falling in place. Some people call it the two way web, some call it the writable web. A more appropriate description may be the publish – subscribe web or the PubSubWeb.

The first examples of the PubsubWeb are currently on the web. We call them bloggers which are the information publishers and their readers get the content by using RSS aggregators. (News Readers). Here the important part is not the blogs itself, but the ability to easily publish and share the content with only who are interested.

From the first days of the net, information and content has increased in size dramatically bringing a huge number of transactions with us to handle each day. But despite the fact that content is getting bigger, the tools we use to handle them are still the same like the email client and the browser. This is the reason why we are overloaded with information. The PubSubWeb may be the solution.

1.1 The Information System

Blogs are just one kind of information that is published by individuals or communities. There is a lot more out there to publish which is really hard to do so but this is what the PubSubWeb makes easy.

There are the information producers and information consumer on the internet ecosystem. Producers would like tools and software to make publishing and distribution easier, while the consumers would like to have tools which enable them to receive information or subscribe to them. At the moment when producers put some content on a web site, they publish it by using email notifications or search engine advertisements. At the other hand consumers usually have bookmarks of specific web sites they often visit or subscribe to mailing lists or newsletters, or they search then content on the web bu using search engines.

This method is usually not scalable which results in people visiting only a handful of sites. Even with the sites visited, the consumer usually has to visit that site

periodically to find out what is new. A better way to distribute and access information. There is a class of information that has the following four attributes:

- It is frequently updated (as opposed to being static).
- It needs to be repeatedly distributed to a continuously interested set of entities (as opposed to one-off, need based access).
- Access to it is incremental (as opposed to getting a complete web page).
- There is a need for a push, near real time delivery or notification (as opposed to demand driven pull)

Some examples for the kind of information are:

- General examples
    - Stock quotes
    - Sports scores
    - Flight arrival and departure information
    - Weather
    - News headlines
- Examples on the enterprise level
    - Inventory levels
    - Sales status
- Examples on a personal level
    - Alerts for meetings
    - Events taking place nearly (Discounts from markets, seminars ...)

Usually most of the information overload problem occurs because we keep getting lots of information that we don't really need to get or we have to poll some site at regular intervals to find whats new and we end up getting the same information if no updates are even present. We need a kind of environment where when exceptional events happen and if we are interested in that event, then we should be notified near simultaneously. At the heart of the problem we see that there is a need for an information stream between the producer and consumer for the content that meets the above four criteria.

There are tools like wikis and weblogs which make writing a lot easier and search engines like Google make finding information and websites much faster. But we still need to do a periodic reading of information and there is no trigger when the content is updated. We only need to get the incremental content (content which is new) but we also receive a lot of irrevelant content downloaded to our browser around the information we really need. The stuff we really need is called the microcontent which does not have any unnecessary information around it. It will be a great thing to do this with the tools we already have like email clients and browsers. Learning a third or fourth application just gets things more complicated.

Another way to think of the PubSubWeb is as an EventWeb. Each update of the content (publishing) is the occurrence of an event. What is needed is for us to be able to (a) subscribe to the event stream and (b) receive notification and details of the event as and when it happens. From a publishers point of view, there may also be a need to restrict access to who can subscribe to the event stream.

What we think of as microcontent or events is only limited by our imagination: a breaking news story, a commentary published by one of our favourite writers, an intimation of a speech in the town we live, the birthday of a friend, a stock quote, the latest scores of a cricket match, an alert about a flights arrival or departure, an alert about the most recent credit card transactions, or maybe an update on the new leads that have come in to the sales department.

So far, the tools that we have been using have been focused on one-way flow of information. Our ability to focus only on the incremental information has been limited to this day. In most cases, we have to go seek out the information source and pull in the information we need. In most cases, push would have worked far better.

1.2 Communication Models in Large Scale Distributed Applications

An important requirement to consider while building large scale distributed software systems is to design in such a way that the system should be able to scale across networks and adapt themselves to changing environments. Internet, a large network provides a big pool of resources from which distributed systems can be constructed [4]. These distributed systems are faced with communication and reliability problems for the following reasons :

- Heterogeneous technologies

- Mobility
- Large scale deployment
- New and unpredictable user behaviours

The scale of these object oriented software, client-server systems is limited by server capacity. The unreliable communication and server unavailability problems is the second reason to search for alternative methods on software design in terms of communication and information flow [4]. The first reason was the push / pull problem discussed on the previous topic The Information System.

Two kinds of programming models have been used for programming distributed applications as the design of these systems became more complex. The two main communication models between different system components are :

- Communication by messages
- Communication by remote method invocation

The choice about which model to use is still and early decision that should be made at the time of the application design. The first method of communication which is asynchronous communication via messaging is the main focus of our research since we beleive it will be the solution to the two problems discussed in this chapter. Before going into the details of messaging, asynchronous communication and publish / subscribe mechanisms we would like to give an example distributed system scenario where such a need is present.

1.2.1 Scenario and Motivation

We'll think about an IT company which has many offices around the country and the world even. Many people work on different projects and cooperate on similar projects which have different roles such as programmer, analyst, designer, project manager, tester, database administrator and so on. The interaction between these people is changing constantly through out the project according to phase in the project and people's focus at that time. Also on each phase of the project different groups of people are more active such as the designers being more active on the beginning of a project where as on later phases engineers and testers are more

dedicated. Different departments may be dealing with different roles or software modules of the project, while people may be involving or leaving these groups dynamically [1]. This kind of development and group work needs tools such as :

- Configuration management repositories
- Databases
- Word processing
- CAD
- Project management tools and so on

These tools are used to produce the products as well as their meta information too such as documentation, specifications and metrics. In such an environment different people should be aware of each other and their current activities and statuses. People are interested with different kinds of information according to their roles in the organization such as programmers being interested on the event that when some software modules in the repository are ready for integration. They mostly would like to be notified of a change request coming to the bug database [1].

In such a dynamic environment people face other problems like mobility and heterogeneity. The device people use can be :

- Desktop computers
- Non-stop servers
- Workstations
- Laptops
- PDA's
- Mobile phones
- Motion sensors and so on

So the systems used in such organizations have to adjust their information delivery policies to comply with different contexts such as time, place and organizational roles and so. While managers want to be notified about the progress of their projects via such tools, users may not be that interested on all events. A user may want to be informed about meetings that he should attend but only when he is in the office and not when he's at home or on a business trip [1].

Users may not be online or connected to the network all the time. For example a manager may return from a one week vacation and would like to see the event history of the past week to see the progress of the project. So the underlying system should have persistence mechanisms to store such events for some time so they can be delivered when the user comes online again. However not all of the events should be persisted. Ordinary events are useless when they are out of date so this functionality should also be configurable. An example could be a temporary unavailability of a printer that ran out of paper.

The scenario we discussed shows the many features of such a system must support such as :

- Integration of heterogeneous event sources
- Composing events
- Filter information
- Event expiration time
- Mobile application support
- User context awareness
- Timing constraints
- Roles
- Priorities
- Notion of groups

2. MESSAGING

In this chapter we will be discussing the topic messaging. We think that asynchronous messaging is the answer to all the problems we have discussed in chapter 1 Introduction. Messaging is a method of communication between software components or applications. A messaging system is a peer to-peer facility: A messaging client can send messages to, and receive messages from, any other client.

Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages. Messaging makes the messaging system responsible for transferring data from one application to another, so the applications can focus on what data they need to share as opposed to how to share it.

Messaging also differs from electronic mail, which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components.

Messaging enables distributed communication that is loosely coupled. So lets look at what coupling is. In computer science, coupling is the degree to which each program module relies on each one of the other modules. Low coupling means that one module does not have to be concerned with the internal implementation of another module, and interacts with another module with a stable interface. With low coupling, a change in one module will not require a change in the implementation of another module. Low coupling is a sign of a well structured computer system.

In messaging world a component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to know anything about their implementations or method signatures. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only what message format and what destination to use. This provides the loose coupling among software modules.

There is also time decoupling in messaging systems. The communicating software systems do not need to be actively participating in the interaction at the same time. While the message sender application can send some messages with the receiver side being disconnected, the receiver also could be notified of these event while the sender is disconnected. We will continue to look on some messaging basics and vocabulary in this chapter.

2.1 Message Exchange Patterns

Message exchange patterns (MEP) are a type of design pattern that describe distributed communications. The patterns describe the interactions between the different participants on the network (multiple clients, multiple servers). Common message exchange patterns are :

- One-way (send a message and don't expect any answer)
- Request-Response (send a message and expect a reply)

More complex patterns can be formed by combining these building blocks.

One-Way message exchange is the simplest pattern. It states that the sender will give a best-effort to move a message from a source location to the destination. No response to the operation is expected.

- One-way messages may or may not be guaranteed.
- One-way messages may or may not be reliable.

The request-response message exchange pattern is where a client asks a service provider a question and then receives the answer to the question. The answer may come in the form of a fault/exception. Both the request and the response are independent messages. The request/response pattern is often implemented using

synchronous operations for simple operations. For longer running operations, asynchronous (with message correlation) is often chosen.

2.2 Synchronous and Asynchronous Operations

A synchronous operation is one that waits for a response before continuing on. This forces operations to occur in a serial order. It is often said that an operation, "blocks" or waits for a response. Synchronous operations will open a communication channel between the parties, make the request and leave the channel open until the response occurs. This method is effective unless there are large numbers of channels being left open for long periods of time. In this case, asynchronous operations may be more appropriate.

An asynchronous operation is one that does not wait for a response before continuing on. This allows operations to occur in a parallel. Thus, the operation does not, "block" or wait for the response.

Asynchronous operations will open a communication channel between the parties, make the request and close the channel before the response occurs. Message correlation is used to relate the inbound message to the outbound message.

This method is effective when there are large numbers of transactions that could take long periods of time to process. In the case where the operations are short or need to run in serial, synchronous operations may be more appropriate.

This asynchronous operation is also called as synchronization decoupling where senders are not blocked while producing events and and receivers can get asynchronously notified while performing some concurrent activity. The production and consumption of events do not happen in the main flow of control of the senders and receivers, and do not therefore happen in synchronous manner.

Decoupling the production and consumption of information increases scalability by removing all explicit dependencies between the interacting participants. In fact, removing these dependencies strongly reduces coordination and thus synchronization between the different entities, and makes the resulting communication infrastructure well adapted to distributed environments that are asynchronous by nature, such as mobile environments [6].

2.3 Guaranteed / Non-Guaranteed Reliable Delivery

Guaranteed delivery means that a message transmitter is guaranteed to know whether the intended recipient received the message or not. Guaranteed delivery should not be confused with reliable delivery. Guaranteed delivery doesn't promise that the message will get there, it only guarantees that the transmitter will know whether the message got there.

Non-guaranteed delivery simply means that an attempt will be made to send a message, but there's no telling if it ever made it to its recipient. One subtle point to be understood about a protocol that is supposed to have guaranteed delivery is whether the protocol has an end to end or only a single hop guarantee.

For instance, if a message is going to be transmitted through several nodes over HTTP, the sender will know whether the first hop received the message based on a 200 or a 400 status code. However, if the message can't be sent on to a second hop, HTTP doesn't have a built in way to inform the original transmitter of that problem. A higher-level protocol may be needed to give end to-end guarantees.

Reliable delivery means the message transport mechanism promises that a message received by the transport will be transmitted to its intended recipient, will not be duplicated, and will have deterministic ordering.

Reliable delivery should not be confused with guaranteed delivery. Reliable delivery can make designing a client or service much easier, but because reliable delivery requires a substantial overhead in computational and network resources, it should not be used when unreliable delivery will do.

2.4 Deterministic / Non-Deterministic Ordering

Deterministic ordering means that a service is guaranteed to receive messages from a single requestor in the same order that the requestor sent them. Obviously,

this can make designing the service much easier than non-deterministic ordering, especially if the requestor and the service must engage in a multi-step conversation. However, deterministic ordering can require more computational and network resources, so it shouldn't be used if it's not needed.

Non-deterministic ordering means that a service may or may not receive messages from a single requestor in the same order that the requestor sent them. This requires less network and computational resources than deterministic ordering, but can be much harder to design for when a requestor and the service must engage in a multi-step conversation.

2.5 Message Correlation

Message Correlation is the concept of tying messages together. Usually, it occurs when one service calls another service asynchronously and passes in a unique correlation token. When the invoked service finishes, it calls the initiating service and passes back the original token. This allows the initiator to correlate the two calls.

2.6 Durable / Non-Durable Messages

A durable message is one that will still get sent to its intended recipient even if the recipient is disconnected or otherwise unable to receive the message when it is transmitted. A non-durable message is one that will not get sent to its intended recipient if the recipient is disconnected or otherwise unable to receive the message when it is transmitted.

2.7 Messaging Systems

Up to now we have discussed very basic concepts about messaging when u look at the user perspective. We will now discuss some concepts from the messaging systems perspective and how these Message Oriented Middlewares implement the features we have seen so far.

2.7.1 Message Channel

So lets say that an enterprise has two applications that needs to communicate via messaging and a messaging system has been set up. So how do they communicate with each other. Apparently the messaging system does not provide a magical way as in the following figure :

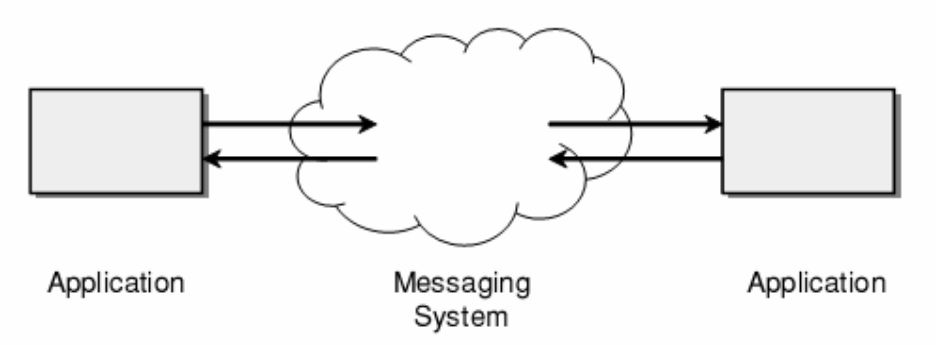

Figure 2.1 : Connecting Applications via Messaging System [7]

Applications do not just throw the information in to the messaging system while other applications receive whatever they can. This is not the way how things work in the messaging world. Rather the applications that are sending the information know what kind of information that is. Also the applications which are receiving messages actually know what kind of information they are interested. Messaging systems are not big black boxes that you can throw information into or pull messages out of. Rather it's a set of connections that applications use to communicate by sending and receiving messages in a predetermined, predictable ways [7].

When an application has a message to send, the application just puts the message on a particular message channel that is present on the messaging middleware.

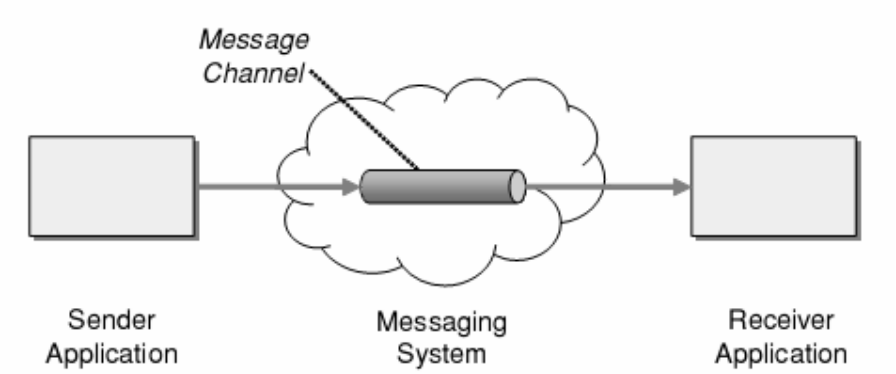

Figure 2.2 : Message Channel [7]

Also the receiving applications don't pick up messages randomly. They listen to the particular message channels that they are expecting information. A key point here is to remember that the sender application does not know what kind of an application will be receiving the messages it sends. The sender is only sure that the receiving

application is interested in the messages it sends because messaging systems have many message channels each serving a specific purpose. So both the sender and receiver applications use message channels that are suitable for their needs.

We can say that message channels are logical addresses in the messaging system. How they are implemented internally, their addressing mechanisms depends totally on the implementation product. Maybe all communicating parties are connected to each other just like a mesh network, or maybe they are all connected through a central hub like system. These kinds of details are hided from the users of the messaging system.

Messaging systems don't come with preconfigured message channels. Developers and designer decide on what kind of message channels they need for their applications. Later the installer or the administrator of the messaging middleware creates and manages these message channels. Some messaging systems also supports creating messaging channels on the fly dynamically while the applications are running but this brings a problem where the receivers would not know the name of the message channel. Maybe distributing the new channel name through some other predefined channel is a solution but still this behavior is not suggested. Applications expect the number and purpose of message channels to be predefined at deployment time [7].

There are different vocabulary used in the messaging domain. We will be using them interchangeably through out this thesis. Some may prefer "sender", "receiver", some may call it "producer", "consumer". You can also see people say "talk" and "listen". But "publisher" and "subscriber" should be treated differently because there is a slight difference there which we will discuss later.

One issue that we should keep in mind while designing applications is to know that channels are cheap but not free. Each channel requires memory and also disk space if they are persistent. So if your application requires thousands of channels you should find a scalable messaging system and test it to see if it suits your needs.

Since we said that message channels are logical addresses within the messaging system we expect that each one should have an address name. This address name depends on the implementation but in most cases channels are referenced by an alphanumeric name such as "MyChannel". Some messaging systems also support

hierarchical naming schemes which enables you to organize channels in a way that is similar to a file system with folders and subfolders. "MyCorp/Prod/OrderProcessing/NewOrders" may be an example for such a naming.

Now we may look at some examples. JMS is the Java Message Service from Sun Microsystems. We will examine it in the next chapter but we will demonstrate some examples now.

The JMS reference server can be started with the j2ee command. Message channels have to be configured using the j2eeadmin tool. This tool can configure both queues and topics.

```
j2eeadmin -addJmsDestination jms/mytopic topic
j2eeadmin -addJmsDestination jms/myqueue queue
```

Figure 2.3 : How to Create Message Channels Using j2eeadmin Tool

Once the channels have been administered (created), they can be accessed by JMS client code. The JNDI lookup doesn't create the queue (or topic); it was already created by the j2eeadmin command. The JNDI lookup simply creates a Queue instance in Java that models and provides access to the queue structure in the messaging system.

```
Context jndiContext = new InitialContext();
Queue myQueue = (Queue) jndiContext.lookup("jms/myqueue");
Topic myTopic = (Topic) jndiContext.lookup("jms/mytopic");
```

Figure 2.4 : Looking up Topic Objects with JNDI

Microsoft's MSMQ provides a number of different ways to create a message channel, called a queue. You can create a queue using the Microsoft Message Queue Explorer or the Computer Management console. From here you can set queue properties or delete queues.

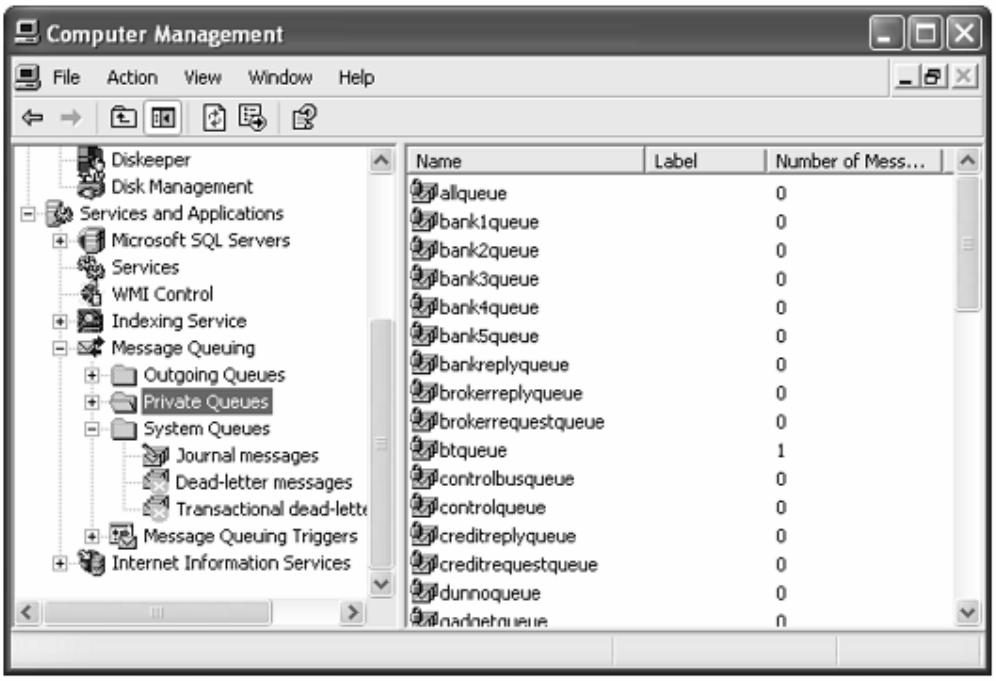

Figure 2.5 : Computer Management Console [7]

2.7.2 Message

So we have our separate applications that would like to communicate with each other through a message channel that is already defined in the messaging system. The question is what kind of data are they going to put on the message channel. It's sure that the applications data is not one big continuous stream. It must consist some units such as records, database rows, objects and so on. We can say that message channels transmit units of data.

In programming when you make a function call, the caller can pass a parameter by reference by passing a pointer to the data's address. Since the caller and the function share the same memory space, there is no problem with this approach. Similarly different threads in an application can use this method too since they are also sharing the same memory space [7].

When two separate processes need to communicate in a messaging system than there is more work to do then just passing a pointer around. There are two different memory spaces now, so the data must be copied from one to the other. What happens is the first process marshals the data into byte form and puts it on the channel while the second process read the stream from the channel and unmarshals that byte form

into its original form. Marshalling is also the operation used when making remote procedure calls (RPC).

There is a need for a simple mechanism that you could send your data easily on a message channel. The solution is to package the information into a Message, a data record that the messaging system can transmit through a Message Channel.

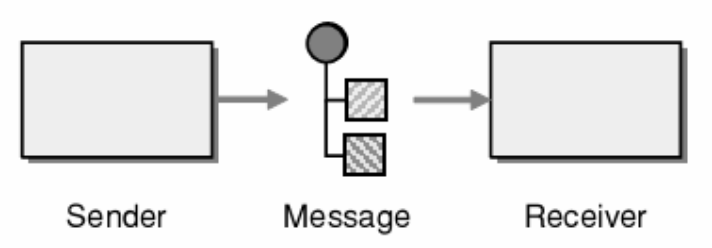

Figure 2.6 : Message being Sent Over a Channel [7]

While it may be different on many implementations messages usually consist of two main parts. The header and the body.

- Header - Information used by the messaging system that describes the data being transmitted, its origin, its destination, and so on.
- Body - The data being transmitted, which is generally ignored by the messaging system and simply transmitted as is.

To the messaging system, all messages are the same: some body of data to be transmitted as described by the header.

We will show some example of messages from various messaging systems. In the Java Message Service a message is an instance of the class message. However there are also other specialized subclasses too.

- TextMessage - The most common type of message. The body is a string, such as literal text or an XML document. textMessage.getText() returns the message body as a String.
- BytesMessage - The simplest, most universal type of message. The body is a byte array. bytesMessage.readBytes(byteArray) copies the contents into the specified byte array.

- ObjectMessage - The body is a single Java object, specifically one that implements java.io.Serializable, which enables the object to be marshaled and unmarshaled. objectMessage.getObject() returns the Serializable.
- StreamMessage - The body is a stream of Java primitives. The receiver uses methods like readBoolean(), readChar(), and readDouble() to read the data from the message.
- MapMessage - The body acts like a java.util.Map, where the keys are Strings. The receiver uses methods like getBoolean("isEnabled") and getInt("numberOfItems") to read the data from the message.

Message class is also the class name which messages are implemented on the .NET platform. A property called the body contains the body of the message as an object. Another property which is the body type contains the information of the bodies type which may be string, date, currency, another object and so.

SOAP messages can also be seen as an example for a message in the SOAP protocol. A SOAP message is an XML document which contains a header and a body section. Below is an example of a SOAP message.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Header>
       <t:Transaction
           xmlns:t="some-URI"
           SOAP-ENV:mustUnderstand="1">
               5
       </t:Transaction>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DEF</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2.7 : SOAP Message Example [7]

Messages can also carry other messages which give them a recursive nature. A SOAP message may be transmitted inside a .NET message or a JMS message as the body object itself. That way SOAP message would have been transmitted via the messaging systems own protocol and not HTTP.

2.7.3 Pipes and Filters

Think of an enterprise which has an order module which accepts messages and uses a messaging system to receive orders. Definitely there should be some pre processing operations applied on these messages. For examples the messages may be encrypted for security reasons and these have to be decrypted. Duplicate messages may come from external message sources and these should be checked and filtered also [7].

To write a comprehensive large message pre processing module would be inflexible and difficult to test. To add new functionality or remove some existing functionality for a specific application would be very hard to implement. Think of an example where the customer placing the orders resides on a private network and do not require encryption. Then we must be able to remove the decryption functionality easily. To place all of the functionality in a single software module would also make the system non – reusable. We should be creating small reusable modules and construct more complex filtering with these building blocks [7].

Pipes and filters are the solution to the problems we discussed above and shown on the figure below :

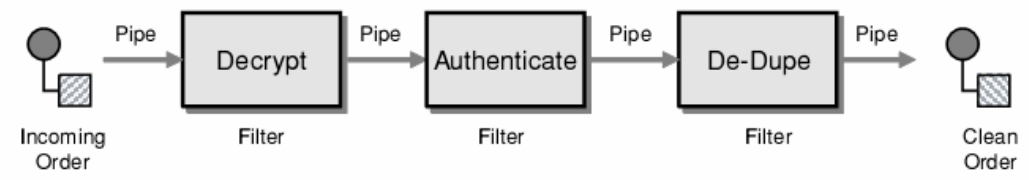

Figure 2.8 : Pipes and Filters Architecture [7]

Each filter on the figure has a very simple interface. Filters receive messages from their input pipe, processes these messages and sends them out via the output pipe. Pipes are the constructs that are connecting filters. Since all filters have the same interface different combinations of filters can easily be formed by removing or adding new filters to a chain. Even rearrangement of filters is possible. This filters

and pipes architecture discussed may be put in front of an order management system for example to pre process received order messages. Messaging systems message channels may be used to implement filters and pipes architecture.

2.7.3.1 Pipeline Processing

If we connect software components with asynchronous message channels than each component in the chain will be operating asynchronously in its own thread. When one of the modules completes processing a message it can pass it to the next module and immediately receive the second message for processing. It need not wait for the first sent message to complete because of the asynchronous nature. This allows for multiple messages to be processed concurrently. This type of architecture is called pipeline processing and has much better performance than a sequential process [7]. You can see the difference between pipeline and sequential processing in the figures below :

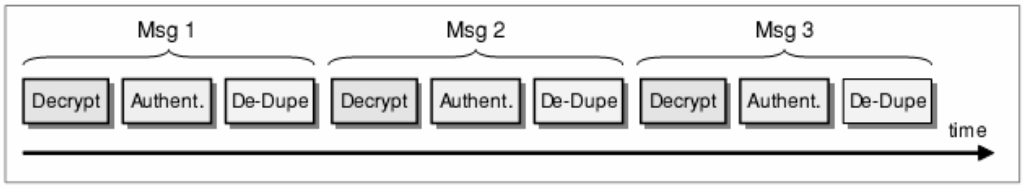

Figure 2.9 : Sequential Processing [7]

In sequential processing message 2 has to wait for all the decryption, authentication and de-duplication process to complete on message 1. Below you can see that pipeline processing takes much lesser time.

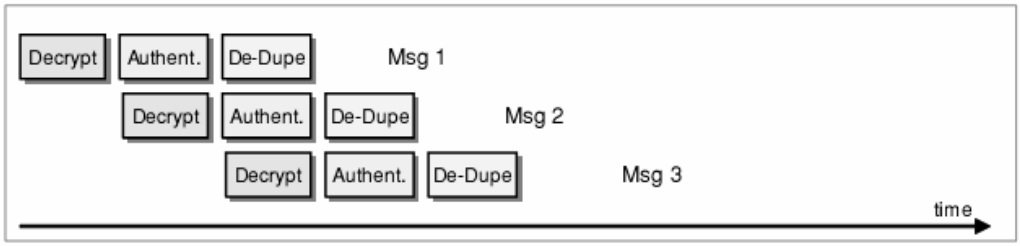

Figure 2.10 : Pipeline Processing [7]

2.7.3.2 Parallel Processing

In pipeline processing one point which is important is that the system throughput is limited with the slowest operation. For example think that the decrypt operation is working really fast and continously sending decrypted messages from its output pipe. If the authentication module is slower however, than the overall system throughput is dependent on the authentication module.

By deploying multiple instances of the slowest module we can increase the overall system throughput as shown in the following figure :

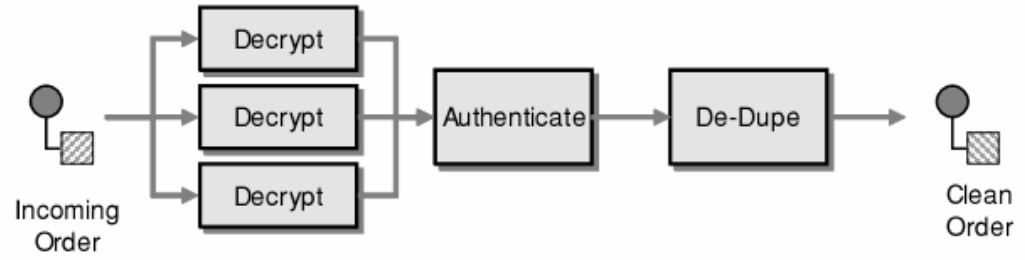

Figure 2.11 : Parallel Processing

However in this architecture we have to assure that each coming message is processed by only one of the available decrypt modules with some kind of routing mechanism. Above you can see that there are three decrypt modules before the authenticate module which are working in parallel in a load balancing fashion. Deploying multiple filters works best when the filters are stateless. Think for example the de-dupe module which eliminates duplicate messages. It maintains a list of the messages that passes through it so we can not deploy multiple instances of this module which itself is stateful [7].

2.7.4 Message Router

Filters are connected with each other through pipes in the pipes and filters architecture and there are large numbers of messages that pass through the same sequential process. However this scenario is not suitable for some cases. In some systems each message is not a part of a single, large data set. There may be messages

İncoming from different systems which require different processing. This may depend on message types or some other criteria [7].

Message routers are the best solution to use in such cases. We insert a special filter called the message router which consumes messages from a message channel and republishes it to another message channel depending on a set of conditions. Below is the figure of the architecture :

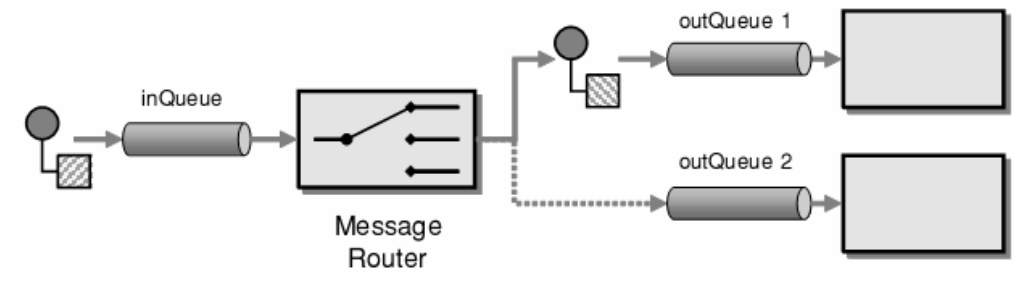

Figure 2.12 : Message Router [7]

The message router is a bit different than the standard pipes and filters architecture. The message router has more than one output port. However the components that the message router connects to are completely unaware of the existence of the message router. They simply consume messages off one channel and publish them to another. A defining property of the Message Router is that it does not modify the message contents; it concerns itself only with the destination of the message.

Another benefit of using the message router is that it maintains the decision criteria for the routing in a single location. When new message type are added or new message locations are defined in the system, only the message router needs to be

modified which brings an easy maintenance. Also, since all messages pass through a single Message Router, incoming messages are guaranteed to be processed one by one in the correct order.

However the message router brings additional processing with itself causing a degrade in performance, which may also lead to a bottleneck on the router itself. This can also be solved with using multiple routers in parallel or adding additional hardware.

2.7.4.1 Message Router Variants

There are many types of message routing architectures which operate completely differently from each other. Some message routers search the message properties and than decides on the destination according to these properties. Such kind of routers are called content based routers. Content based routers are the most common types. Another type of router variant may decide on the destination of the message by checking some criteria on the environment. We call these routers context based routers. Such routers are usually used for tasks like load-balancing, test or failover functionality. For example a context based router can check the processing module and if it is in failed state reroutes the message to another processing module which can be thought as the failover capability. Other routers split the flow of messages evenly across multiple channels to Message achieve parallel processing similar to a load balancer.

Many message routers are stateless in nature which means that they look at the current message and make the routing decision according to only that message. When the next message arrives the previous message information is not existent. Other routers however may consider information from the previous messages for example to eliminate duplicate messages. Those kinds of routers are said to be stateful.

2.7.5 Message Endpoint

Applications communicate with each other through sending messages on the message channels. The messaging system and applications are completely different sets of software developed independently from each other. Application serve specific purposes for some kind of users where as a messaging system has a more global function or service. Messaging systems are more kind of have the same class as

database management systems or web servers. So since the application and messaging system are seperate softwares they must have a way to communicate with each other [7]. What should replace the clouds in the figure below :

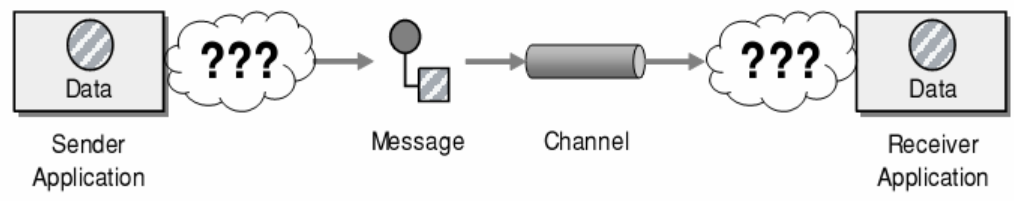

Figure 2.13 : Application Communication [7]

A messaging system can be thought of as a server in terms of its ability to take requests and respond to them. We can call such a system as a messaging server. An application which uses the messaging server for communication is the messaging client. Since the client applications of a database do not know anything about the internals of the database system, it should be same also with the messaging server. Each messaging server provides a client API which applications may use to communicate with the messaging server just like the JDBC API for communicating with databases in the Java language [7].

Each application must have some code to communicate with the messaging server and send and receive messages and so on. This code is called the message end point. This is the code that connects to the messaging system, gets the data and converts it into a message and puts it onto a specific message channel. Below is the architecture where you can see where the message endpoint resides.

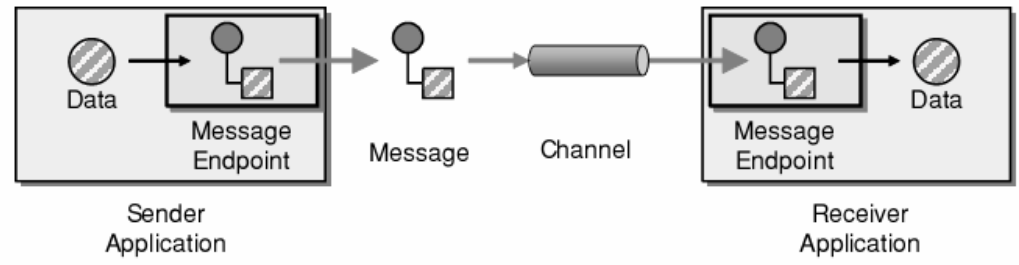

Figure 2.14 : Message Endpoint [7]

Message endpoint is also the code that accepts the messages coming from other applications via the messaging system. It encapsulates the messagins system from the rest of the application by being the bridge in between in a standard way. If the application needs to switch to another messaging system only the message endpoint code needs to rewritten. A message endpoint may send or receives messages but has single functionality at a time which means an endpoint can either send or receive messages. Also an endpoint only communicates via a single message channel so an application which needs to communicate with multiple channels require multiple endpoints.

In the JMS API the message endpoints are represented with the classes MessageProducer and MessageConsumer. An application would require and instance of these classes to send or receive messages.

2.8 Information Dissemination

The main focus of information dissemination is to transfer data from producers to consumers. The most important objectives of information dissemination are :

- Make new content available
- Update existing content
- Revoking obsolete content

Information dissemination has two main strategies, *pull* and *push*.

- In the pull category, information information consumers initiate the communication. Think of a web browser hitting the refresh button to initiate a communication with the web server and request the web page's last version.
- In the push category, the information producer is the side that is initiating the communication.

Push and pull, both categories are about who is initiating the communication, the producer or the consumer. However there is also another category which is about, on which periods or time the communication should be initiated. These categoriers are the periodic and aperiodic communication [3].

- In the periodic communication category the producer and consumer communicate on fixed intervals of time regularly no matter which side is initiating the communication. Teletext can be an example for this type of communication. The time interval could be fixed like one minute or it can be adapting itself according to the observed rate of data change [3].
- With the aperiodic communication neither the producer nor the consumer predetermines the times when they communicate. The communication takes places when certain events occur in the system. For example when a stock price changes a system may initiate a communication and push the event and new price values to interested consumers [3].

Push or pull, periodic or aperiodic communication models each have its advantages or disadvantages. For applications which are very dependent on timely information, aperiodic pull is not appropriate because a consumer will be needing the information updates as soon as it occurs. Periodic pull which also known as polling, forces the consumers to request information continously. To have a near event time updates the polling should be made frequently which wastes large amounts of network resources [3].

This same waste of resources also applies to periodic push, which again transmits the same data even if it is not changed. For these reasons on this thesis we will be focusing on the aperiodic push which has the following advantages:

- The sources initiate communication only if there is an update on the data, which means that the same data is not transmitted over and over again when there is no change.
- Each communication requires only a single message, where with a pull a request and reply makes 2 messages on minimum.
- Multicast messages are possible rather than only point to point messaging.
- The architecture can additionaly support aperiodic pull to support historical event data gathering for new clients.

2.9 Messaging Models

Most messaging products either point to point, publish – subscribe or both models for communication.

A point-to-point (PTP) product or application is built around the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queue(s) established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire. Below is the figure describing this model :

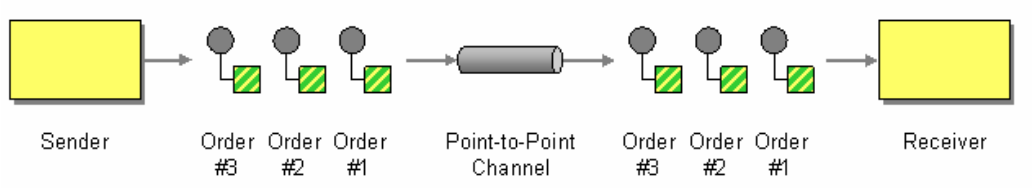

Figure 2.15 : Point-to-Point Messaging [7]

A Point-to-Point *Channel* ensures that only one receiver consumes any given message. If the channel has multiple receivers, only one of them can successfully consume a particular message. If multiple receivers try to consume a single message, the channel ensures that only one of them succeeds, so the receivers do not have to coordinate with each other. The channel can still have multiple receivers to consume multiple messages concurrently, but only a single receiver consumes any one message.

2.9.1 Publish – Subscribe Messaging Model

How can an application in an integration architecture only send messages to the applications that are interested in receiving the messages without knowing the identities of the receivers?

Integrating applications so that they receive only the messages they are interested in involves the following facts:

- The applications in an integration architecture consume different message types. For example, applications that manage customer information are interested in customer information updates. Trading applications are interested in buy and sell transactions.

- An application in an integration architecture may send several message types. For example, the application may send customer information messages and operational messages about its status. (Status is also referred to as *health* in this context). Likewise, an application in an integration architecture is usually interested only in a subset of the messages that are sent by the other applications.
- The extent to which applications let you add information to their messages varies widely. Fixed binary messages usually provide no flexibility or limited flexibility in this area. In contrast, it is usually easy to extend SOAP messages through envelope elements.
- Most integration architectures integrate proprietary applications. These applications often make strong assumptions about the messages that they use to communicate with other applications in the environment. Even with a flexible message format, it may be difficult to insert or to process message elements that the application does not know about.

Extend the communication infrastructure by creating topics or by dynamically inspecting message content. Enable listening applications to subscribe to specific messages. Create a mechanism that sends messages to all interested subscribers.

In a publish/subscribe (pub/sub) product or application, clients address messages to a topic. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.

The three variations of the *Publish/Subscribe* pattern you can use to create a mechanism that sends messages to all interested subscribers are *List-Based Publish/Subscribe*, *Broadcast-Based Publish/Subscribe*, and *Content-Based Publish/Subscribe*.

2.9.1.1 List-Based Publish/Subscribe

A *List-Based Publish/Subscribe* pattern advises you to identify a subject and to maintain a list of subscribers for that subject. When events occur, you have the subject notify each subscriber on the subscription list. A classic way to implement this design is described in the Observer pattern. When you use this pattern, you identify two classes: subjects and observers. Assuming you use a push model update, you add three methods to the subject: Attach(),Detach(), and Notify().You add one method to the observer—Update().

To use an observer, all interested observers register with the subject by using the Attach()method. As changes occur to the subject, the subject then calls each registered observer by using the Notify()method.

An observer works fine if you have created instances of objects that reify all your observers and subjects. An observer is especially well suited to situations where you have one-to-many relationships between your subjects and your observers. However, in the context of integration, you often have many observers that are linked to many subjects, which complicates the basic *Observer* pattern. One way to implement this many-to-many relationship is to create many subjects and to have each subject contain a list of observers.

If you use this object structure to implement *Publish/Subscribe*, you must write these relationships to persistent storage between process executions. To do so within a relational database, you must add an associative table to resolve the many-to-many dependencies between subject and observer. After you write this information to persistent storage in a set of tables, you can directly query the database for the list of subscribers for a topic.

Maintaining lists of published topics (subjects) and subscribers (observers) and then notifying each one individually as events occur is the essence of *List-Based Publish/Subscribe* implementations.

2.9.1.2 Broadcast-Based Publish/Subscribe

When you use a *Broadcast-Based Publish/Subscribe* approach, an event publisher creates a message and broadcasts it to the local area network (LAN). A service on each listening node inspects the subject line. If the listening node matches

the subject line to a subject that it subscribes to, the listening node processes the message. If the subject does not match, the listening node ignores the message.

Subject lines are hierarchical and may contain multiple fields that are separated by periods. Listening nodes that are interested in a particular subject can subscribe to these fields by using wildcards, if required.

Although this *Broadcast-Based Publish/Subscribe* implementation is an effective method for decoupling producers from consumers, it is sometimes useful to identify particular topic subscribers. To identify topic subscribers, a coordinating process sends a message that asks listening nodes to reply if they subscribe to a particular topic. Responses are then returned by each listening node to the provider to identify the subscribers.

Because all messages are sent to all listening nodes, and because each node is responsible for filtering unwanted messages, some authors refer to this as a publish/subscribe channel with reactive filtering.

2.9.1.3 Content-Based Publish/Subscribe

Both *Broadcast-Based Publish/Subscribe* implementations and *List-Based Publish/Subscribe* implementations can be broadly categorized as topic-based because they both use predefined subjects as many-to-many channels.

Publish/Subscribe implementations have recently evolved to include a new form—*Content-Based Publish/Subscribe*. The difference between topic-based and content-based approaches is as follows:

In a topic-based system, processes exchange information through a set of predefined subjects (topics) which represent many-to-many distinct (and fixed) logical channels. Content-based systems are more flexible as subscriptions are related to specific information content and, therefore, each combination of information items can actually be seen as a single dynamic logical channel. This exponential enlargement of potential logical channels has changed the way to implement a pub/sub system.

The practical implication of this approach is that messages are intelligently routed to their final destination based on the content of the message. This approach overcomes the limitation of a broadcast-based system, where distribution is coupled to a

multicast tree that is based on Transmission Control Protocol (TCP). It also gives the integration architect a great deal of flexibility when deciding on content-based routing logic.

2.9.1.4 Applying Publish - Subscribe

The figure below shows an integration solution that consists of four applications. The sender (also called a *publisher*) uses a topic-based approach to publish messages to topic A and to topic B. Three receivers (also called *subscribers*) subscribe to these topics; one receiver subscribes to topic A, one receiver subscribes to topic B, and one receiver subscribes to both topic A and to topic B. The arrows show messages flowing from the publisher to each subscriber according to these subscriptions.

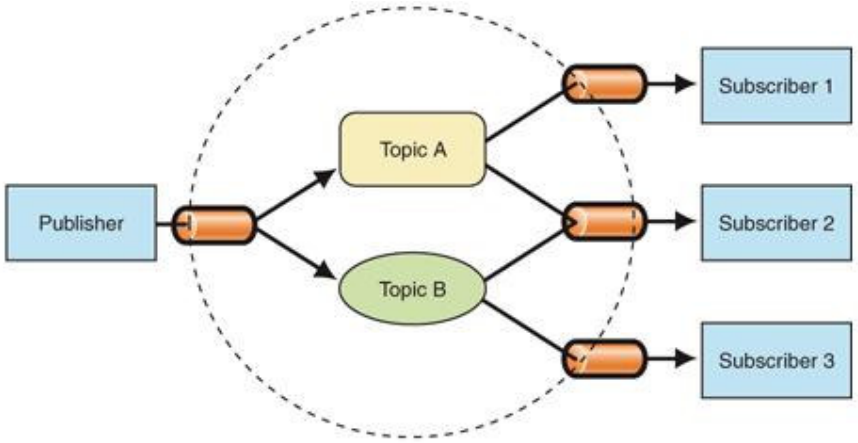

Figure 2.16 : Publish – Subscribe Model

Implementing *Publish/Subscribe* usually affects the messages, the integrated applications, and the communication infrastructure.

First, you must identify the topics or the content of interest to the receiving applications. This translates into partitioning the set of message types into different subsets. For example, consider the types of messages that are sent by a trading system. Some trading applications track buy transactions, some track sell transactions, and other applications track both types of transaction. Separating the

message by creating a buy topic and a sell topic partitions the trading system messages into subsets aimed at these applications.

Next, you must add information to the messages that indicates the topic or that identifies specific content information. Sometimes you can store the topic-related information in an unused message field. Alternatively, you may be able to add a new field for the topic. For example, you may be able to insert a new element in a SOAP header. If you can neither use an existing field nor add a new one, you must find other ways to encode the topic into the message, or you must use a content-based approach instead.

You must then extend the communication infrastructure so that it delivers messages according to each subscriber's subscription. The approach that you use depends on the topology of the integration solution. For example, consider the three common topologies. For bus integration, you can implement the subscription mechanism in the bus interface. For broker integration, you can implement the mechanism through subscription lists to the broker. For point-to-point integration, you can implement the mechanism through subscription lists in the publisher.

Finally, you must modify the integrated applications. The publisher must add the topic-related information to each message that it publishes. For example, if the topic is encoded as a header element, the publisher must insert the topic-related information into the appropriate element. Likewise, the subscriber must specify the topics of interest.

Subscriptions can be either fixed or dynamic. With fixed subscriptions, the integration architect sets the topics that an application subscribes to. Applications have no control over their subscriptions. Usually, the subscriptions are specified when each application is added to the integration solution. The figure below shows a fixed subscription to Topic A.

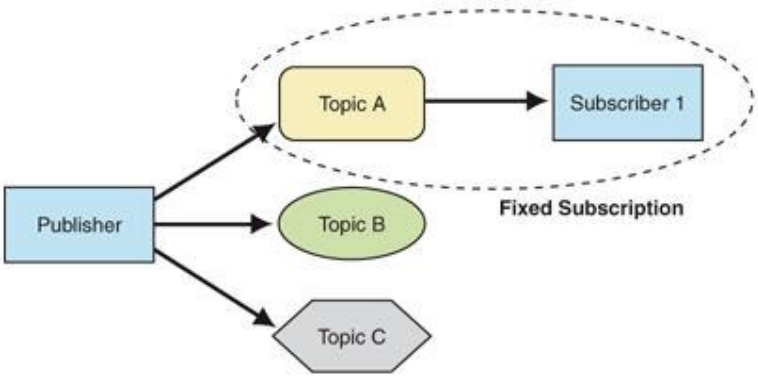

Figure 2.17 : Fixed Subscription

In contrast, dynamic subscriptions enable applications to control their own subscriptions through a set of control messages. Applications can remove existing subscriptions by sending messages to the communication infrastructure that remove the application from the subscription list. Applications can add new subscriptions by sending messages to the communication infrastructure that add the application to a subscription list. Most communication infrastructures that have *Publish/Subscribe* capabilities provide this feature. However, supporting dynamic subscriptions is not a requirement.

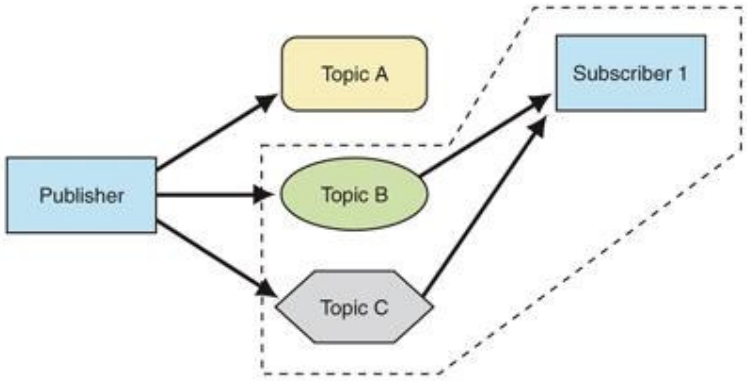

Figure 2.18 : Dynamic Subscription

The figure above shows how dynamic subscriptions function. The top part of Figure 3 shows the initial subscription to topic A. The application then sends a message that removes it from the subscription list for topic A. The application then sends two messages that subscribe the application to topic B and topic C. The bottom part of the figure above shows the final subscription after these control messages are sent. Using *Publish/Subscribe* has the following advantages :

- Lowered coupling. The publisher is not aware of the number of subscribers, of the identities of the subscribers, or of the message types that the subscribers are subscribed to.
- Improved security. The communication infrastructure transports the published messages only to the applications that are subscribed to the corresponding topic. Specific applications can exchange messages directly, excluding other applications from the message exchange.
- Improved testability. Topics usually reduce the number of messages that are required for testing.

The following are the disadvantages of such a model :

- Increased complexity. *Publish/Subscribe* requires you to address the following:
    - You have to design a message classification scheme for topic implementation.
    - You have to implement the subscription mechanism.
    - You have to modify the publisher and the subscribers.
- Increased maintenance effort. Managing topics requires maintenance work. Organizations that maintain many topics usually have formal procedures for their use.
- Decreased performance. Subscription management adds overhead. This overhead increases the latency of message exchange, and this latency decreases performance.

3. JAVA MESSAGE SERVICE (JMS)

JMS is the solution of Sun Microsystems in the area of messaging. So we can start with the question what is JMS. Java Message Service (JMS) is a set of standard interfaces which form a Java API for applications to create, send, receive and read messages while using any kind of server implementation that conforms to the JMS API specification. The JMS API is similar to the JNDI and JDBC API's from Sun which only has a set of interfaces. All the implementation is left to the specific vendor. Advantages of this approach :

- Maximum application portability
- Powerful set of messaging features
- Reduced dependency on a specific implementation
- Low learning curve
- Learn once and use on every implementation and platform

Another big side effect of this approach to design the API itself is to increase the competition between the vendors to provide better support, scalability and performance, rather then focusing on introducing new features which may not benefit the whole community.

JMS API provides communication with two main characteristics which are it being loosely coupled, asynchronous and reliable :

- Asynchronocity as discussed in the messaging chapter means that the receiver does not need to actively request for the messages that it should receive. This is similar to the postal services we use. We just provide an address to them and wait for the package. We dont check with the postal service everyday asking for our delivery everyday.

- Reliable communication means that we can be assured that out messages will be delivered 100% and the messages will also be delivered once and only once. Consider an online shopping system using messaging. In a non reliable communication we could either be getting duplicate orders, or our order would not reach the shop at all.

3.1 When to Use JMS API

If the below conditions are present an enterprise application would most likely choose the JMS messaging API instead of a tightly coupled RPC communication [8].

- Components should not now the interfaces of other components in order to communicate, so that the components should easily be replaced with new components developed inhouse or bought from a third party.
- The application should be able to run still when all components are not up. Communication should still possible when some components are down or at maintenance.
- The business logic does not require that a component should wait for a response from the component which it communicates and rather continue its execution.

We can think of an example enterprise application from a manufactoring factory which produces electronic equipment.

- The inventory component which is in charge of the produced goods may send a message to the factory component to produce more LCD monitors when the number of LCD monitors gets low in the inventory. This message could be sent at weekends when there are sales to shops but the factory is not working. So on a weekday when the factory starts working again and online, the message sent on the weekend could be processed [8].
- Similarly the factory component sends a message to the parts component so it can organise the necessary parts of the electronic equipment.

- The parts component may also communicate with its own part inventory component to check with the parts in stock and also communicate with the order module if its necessary to order new parts from the suppliers.
- The whole system can publish updated catalog and electronic goods information to its sales offices around the world even.

The following figure may show the messaging network which is completely asynchronous in an enterprise application.

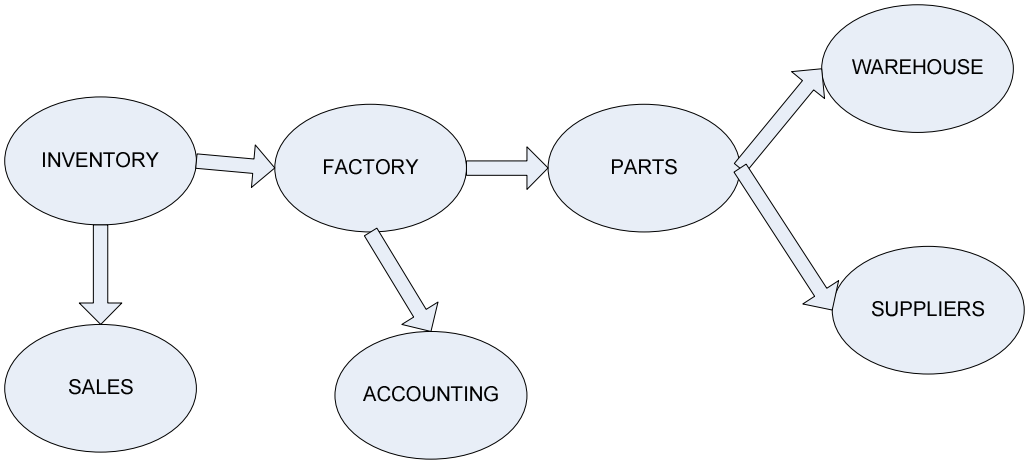

Figure 3.1 : Messaging in an Enterprise Application

3.2 Basic JMS Concepts

We can say that a basic JMS application has 4 main parts :

- JMS provider : A JMS provider is a messaging system (server) which implements all of the JMS API interfaces, and has utilities software to manage the messaging system and create administrative objects and so on. Sonic MQ server, WebLogic JMS Server, IBM MQ series and Sun J2EE reference implementations after version 1.3 are examples of a JMS provider [8].
- JMS clients : All kinds of software or j2ee components written in the Java language which can consume or produce messages by connecting to the JMS server can be JMS clients.

- Messages : These are objects that are transmitted between the provider and clients.
- Administered Objects : These are objects that reside at the server side. The administrator of the JMS server creates these via the tools provided by the JMS server. These objects are preconfigured by the messaging system so the clients just look up these using JNDI, obtain a reference and use them. The two kinds of administered objects are connection factories and destination objects [8].

Below you can see where the four main parts fit and interact in the JMS architecture :

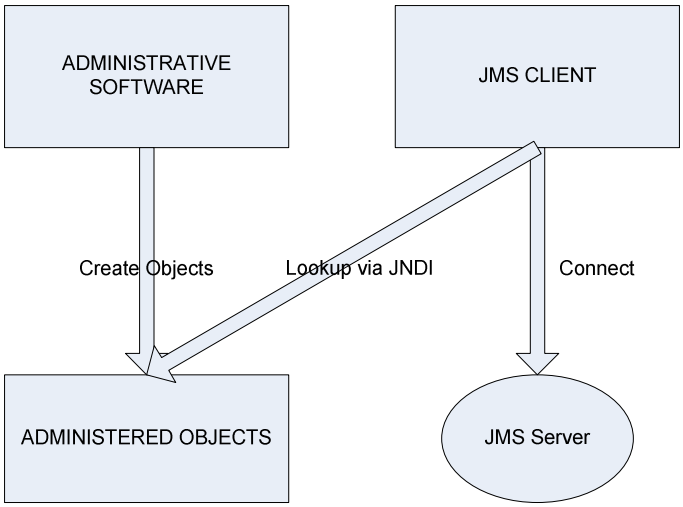

Figure 3.2 : JMS Architecture

3.3 Messaging Domains

The JMS messaging system can be divided into two main domains :

- Point–to–Point Messaging Domain
- Publish/Subscribe Messaging Domain

While standalone JMS providers can implement only one of the domains, a standard J2EE providers must implement both domains to be considered a J2EE compliant server. While you can develop software for both domains using the JMS API, after JMS version 1.1 the API also supports common interfaces so that you can program in a single way for both of the domains.

3.3.1 Point-to-Point Messaging Domain

Despite we have discussed this type of messaging in the previous chapter, we still will have a look at it here in terms of a JMS domain. Point-to-Point (PTP) defines a messaging domain in JMS and its based mainly on queues, senders and receivers. In a PTP application, each message is sent to a queue and the receiver applications extract their messages from the queues they are interested in. Queues are persistent which means that queues hold the messages until they are extracted by the client or they are expired. Message queues operate with the *First In First Out* (FIFO) rule which means that the messages leave the queue in the same order that they were sent. Neither the sender knows the consumer, not the consumer knows that producer of the message [8]. Message queue is the common point of exchange for the sender and receiver applications. Below you can see the figure for this architecture :

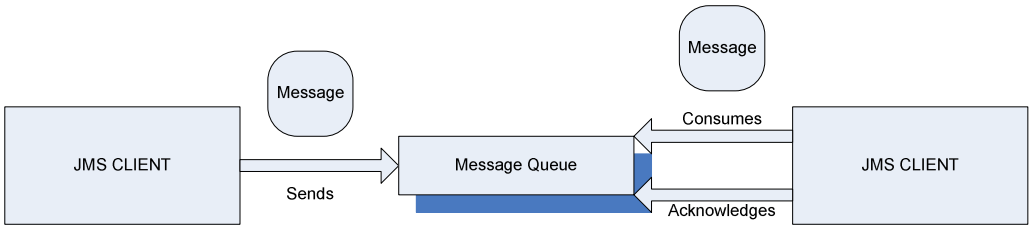

Figure 3.3 : JMS Point-to-Point Messaging

3.3.2 Publish/Subscribe Messaging Domain

In the publish/subscribe domain, there may be more than one receivers of a single message. This is the biggest difference between publish/subscribe and point-to-point domains. Instead of queues, there are topics in this architecture where the messages are published instead of sent. Topics are very similar to queues but multiple consumers can share a topic where only one consumer can consume messages on a queue. Topics are also in the first in first out scheme. There may be more than one producer also publishing messages on a single topic. Consumers are called the subscribers in this domain. Topics are again persistent like queues so they retain the messages until they expire or the last receiptent gets his message [8]. Three important things to keep in mind in publish/subscribe model are :

- Multiple consumers can receive a single message.
- Subscribers will only receive the messages that are published after their subscription. This is not the case in point-to-point messaging. In PTP a newly consumers can receive all the messages present on the queue. However in the publish/subscribe domain there is a time dependency between subscription date and message date. Think of a case where a publisher publishes a message and the two subscribers who are client 1 and client 2 receive the message. If a client 3 later comes and subscribes to the same topic, he will not receive the message.
- Subscribers in general must be online to receive the messages. However durable subscriptions are an exception to this, where topics act like a persistent queue.

Below you can the figure for the publish/subscribe architecture :

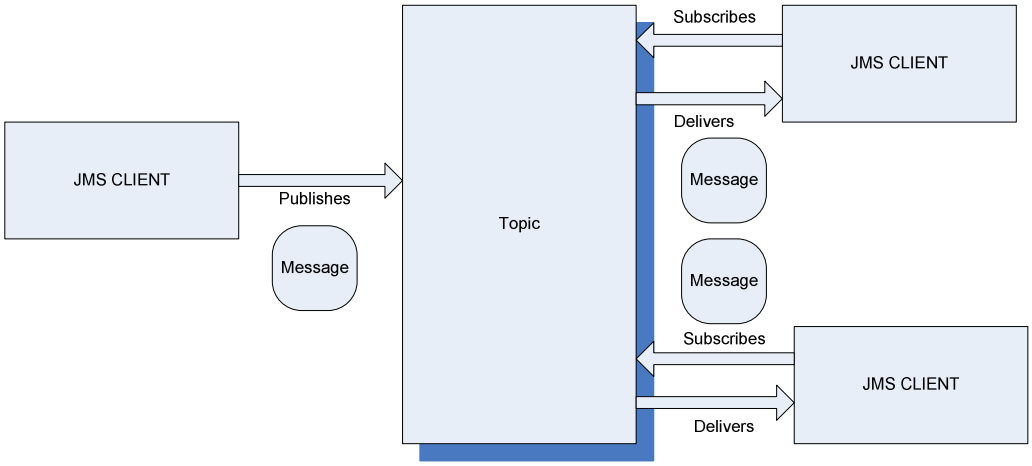

Figure 3.4 : Publish/Subscribe Messaging

3.4 The JMS API Programming Model

A JMS application can be built using the building blocks listed below :

- Administered Objects
- Connections
- Sessions
- Message Producers
- Message Consumers
- Messages

These objects are the main JMS related objects that are necessary to any JMS application. In some specific order applications either create these objects or look them up via JNDI lookup. Creation of an object may require another object as a parameter to its constructor. As a result the application will be calling methods on these objects to send and receive messages. These objects form the message endpoint concept discussed in the previous chapter.

Before discussing these objects you can see the architecture figure where all objects fits below :

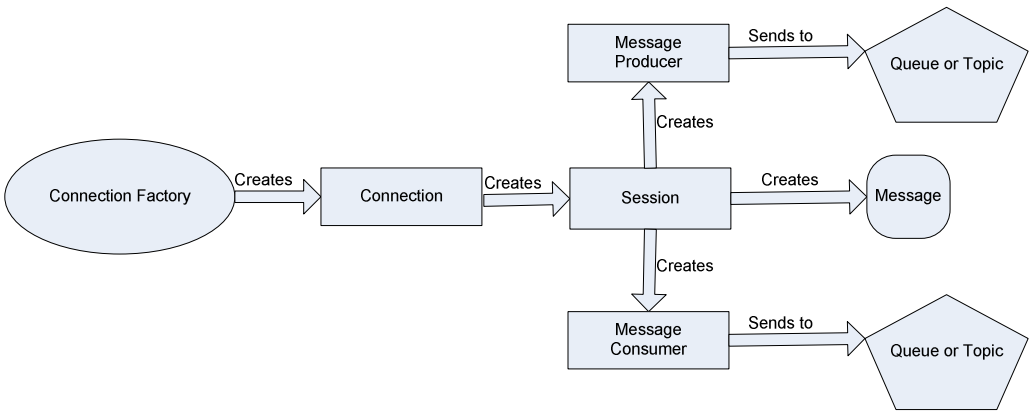

Figure 3.5 : The JMS API Programming Model

3.4.1 Administrative Objects

These are the objects which are maintained outside the JMS programs. They are called administrative objects because they are created or removed by an administrator who is using the administrative tools of the relevant JMS implementation. For example Sun's J2EE 1.3 Reference implementation has a tool called the *j2eeadmin* that can be used to administer JMS [8].

Another reason why this objects are administrative is that the underlying technology of these objects will vary from vendor to vendor therefore the administration of it is different from provider to provider. But still all different provider objects implements the JMS interfaces so the software we develop can use these objects with a standard interface to them. The JMS clients usually looks up for these objects using the JNDI API.

Two different kinds of administrative objects are the Destinations and Connection Factories :

Connection factories are the objects that the client programs use to connect to the JMS server. This object encapsulates some connection specific parameters defined by the administrator. Each connection factory can be an instance of one of the three listed classes :

- ConnectionFactory
- QueueConnectionFactory

- TopicConnectionFactory

On the initialization part of every JMS application, you should perform a JNDI look up of a connection factory and cast it into a ConnectionFactory instance. The following figure shows how to obtains a connection factory object :

```
Context ctx = new InitialContext();
ConnectionFactory connectionFactory = (ConnectionFactory)
  ctx.lookup("jms/ConnectionFactory");
```

Figure 3.6 : Obtaining a Connection Factory

Destination objects are the second kind of objects which are administered. They are used to determine the location where the producers send their messages to and the receivers get their messages from. They may be queues in the point-to-point messaging domain, or topics in the publish/subscribe domain. Again destinations are retrieved via the JNDI API and casted onto appropriate destionation objects. Below are two examples of retrieving destination objects :

```
Destination myDest = (Destination) ctx.lookup("jms/MyTopic");
Queue myQueue = (Queue) ctx.lookup("jms/MyQueue");
```

Figure 3.7 : Obtaining Destination Objects

3.4.2 Connections

This is a generic term used for the connection between the client and the JMS provider. It is a socket connection actually between the client and server. The client developers do not deal with low level socket programming though. Rather you can obtain a connection from the ConnectionFactory class using the *createConnection* method. Once a connection is obtained, it can be started with the *start* method. The service can be started and stopped without losing the connection. Stopping a connection means that the client can not consume the messages. Since connections use system resources it should be closed after use with the *close* method [8].

Closing a connection also closes its sessions and their message producers and message consumers.

Normally one connection is enough for an application, but in rare cases where an application has to consume messages from a JMS server and produce and publish

messages on another JMS server there is a need to have two connections. One per JMS server interacted with. Below is the sample code for creating and closing a connection.

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

Figure 3.8 : Creating a JMS Connection

3.4.3 Sessions

Session objects provide the transactional context for sending and receiving messages. With the session object you can create necessary objects to send and receive messages which are message producers, consumers and the messages itself. There are two kinds of sessions which are TopicSession and QueueSession. Session objects implement the *session* interface [8]. After creating a connection object you can obtain a session from it with the following code :

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

Session session = connection.createSession(true, 0);
```

Figure 3.9 : Creating a JMS Session

3.4.4 Message Producers

You can create a message producer object through the session object. Message producers implement the MessageProducer interface. You can send messages to destinations. These message producers have the *send* method to send messages. Below is the figure where you can see how to create a producer for a given queue and topic.

```
MessageProducer producer = session.createProducer(myQueue);
MessageProducer producer = session.createProducer(myTopic);
```

Figure 3.10 : Creating Message Producers

3.4.5 Message Consumers

You can create a message consumer object through the session object. Message consumers implement the MessageConsumer interface. You can receive messages from destinations. Message consumers have the *receive* method to receive messages.

Below is the figure where you can see how to create a consumer for a given queue and topic.

```
MessageConsumer consumer = session.createConsumer(myQueue);
MessageConsumer consumer = session.createConsumer(myTopic);
```

Figure 3.11 : Creating Message Consumers

When a message consumer is created it is said to be in active state where you can consume messages. By calling the *close* method on the consumer you can switch it to an inactive state where message consuming is not possible. You can receive messages only synchronously by using the receive method after calling *start* method as shown below.

```
connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive(1000);
```

Figure 3.12 : Consuming a Message Synchronously

In order to consume messages asynchrnously, we need to use message listeners. Message listeners are objects which implement the *MessageListener* interface. This interface has a single method which is the *onMessage* method. In this method you can define the actions that will execute when a message is received. After creating a message listener, we have to associate it with a message consumer with the *setMessageListener* method of the consumer as shown in the figure below.

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

Figure 3.13 : Creating and Registering a Message Listener

When message delivery occurs, the JMS provider will automatically call the *onMessage* method of your message listener. This method also has a *Message* parameter which your *onMessage* method can cast to any message type.

3.4.6 Message Selectors

Some applications are not interested in all the published messages to a destination, so they need to filter some messages. We can achieve this functionality by using the JMS message selectors. Message selectors are the constructs that a message consumer use to represent the messages it is interested in.

A message selector is an actual string which contains an expression. The syntax is based on a subset of SQL92 syntax. The following message selector helps the consumer to select only the messages which have a property of NewsType which is equal to *finance* or *art*. *NewsType = 'Finance' or NewsType = 'Art'* [8].

3.4.7 Messages

Messages are the objects which wrap the data that our applications want to produce and receive. A JMS message is formed of the following parts :

- Message Headers
- Message Properties
- Message Body

Among these parts only the Message Headers part is mandatory while the others are optional.

3.4.7.1 Message Headers

A JMS message header contains some fields which are mandatory so that the client and the provider use them to identify and route messages. For example the *JMSMessageID* field is the unique identifier of the message where as the JMSDestination field represents the queue or destination the message has been sent to. Each header field has its own associated getter and setter methods as defined by the message interface. Some of these methods are automatically set by the send or publish methods while some should be set by the client. Below is the table for all the JMS message header fields [8].

Table 3.1 : JMS Header Fields [8]

Header Field	Set By
JMSDestination	send or publish method
JMSDeliveryMode	send or publish method
JMSExpiration	send or publish method
JMSPriority	send or publish method
JMSMessageID	send or publish method
JMSTimestamp	send or publish method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	JMS provider

3.4.7.2 Message Properties

You can create custom message properties in your messages. For example in an application which publishes News data, a property field names NewsType might be helpful. These properties can also be used in message selectors too.

3.4.7.3 Message Body

The JMS API has six message formats which are shown in the table below. The message body contains the actual data that is to be transmitted. You can choose a message body type according to your applications need.

Table 3.2 : JMS Message Body Types [8]

Message Type	Body Contains
TextMessage	A java.lang.String object (for example, the contents of an Extensible Markup Language file).
MapMessage	A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

Below you can see the code to create and send and receive a TextMessage :

```
TextMessage message = session.createTextMessage();
message.setText(msg_text);
producer.send(message);


Message m = consumer.receive();
if (m instanceof TextMessage) {
  TextMessage message = (TextMessage) m;
  System.out.println("Reading message: " + message.getText());
} else {

}
```

Figure 3.14 : Creating, Sending, Consuming a TextMessage

3.4.8 JMS Exceptions

The JMS API defines also the exceptions that should be thrown while doing JMS operations. The root exception where all exceptions derive is the JMSException so catching the JMSException provides a generic way of handling all exceptions related to the JMS API. The JMSException subclasses are listed below.

- IllegalStateException
- InvalidClientIDException
- InvalidDestinationException
- InvalidSelectorException
- JMSSecurityException
- MessageEOFException
- MessageFormatException
- MessageNotReadableException
- MessageNowWriteableException
- ResourceAllocationException
- TransactionInProgressException
- TransactionRolledBackException

4. NEWS FEEDER

In this chapter we will look at the news feeder application that is developed in this thesis study. The News Feeder consists of two seperate applications which are the client and the admin applications. News Feeder is developed in the Java programming language using the Eclipse platform as the IDE. Sonic MQ Server is used as the JMS implementation. The screens and usage and code snippets of how the relevant functionality achieved will be shown.

4.1 Sonic MQ Server and Management Console

Version 6.1 of the Sonic MQ Server was used as the necessary JMS implementation in this study. In order for the news feeder application to run, sonic server should be started. The following console should be seen after you start the Sonic server on the Windows XP platform.

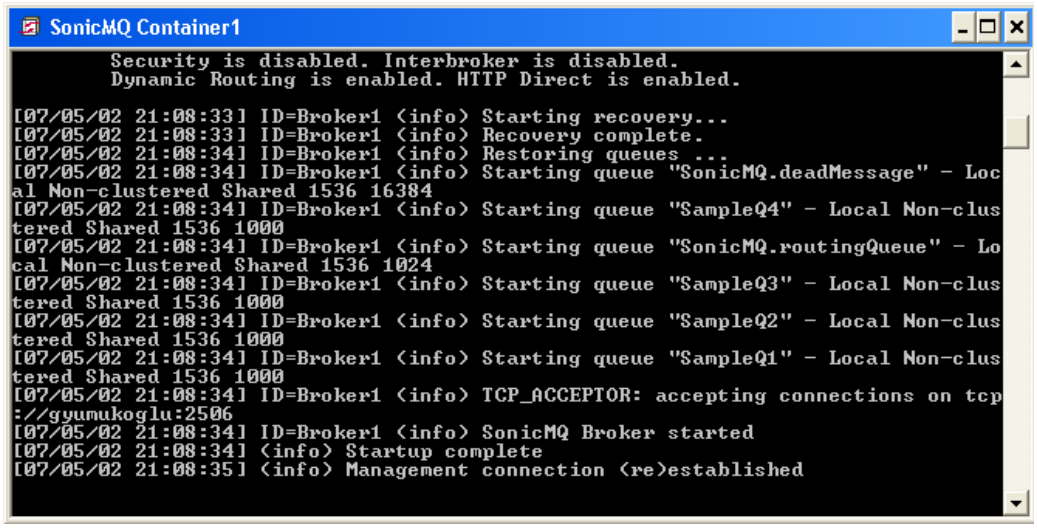

Figure 4.1 : Sonic Startup

After starting the server the administered objects should be configured at least the Connection Factories so that our News Feeder applications can look it up and connect to the Sonic MQ server. The administrative tool Sonic MQ provides is called the Management Console. You can find the shortcut on the start menu if it is installed on the Windows platform.

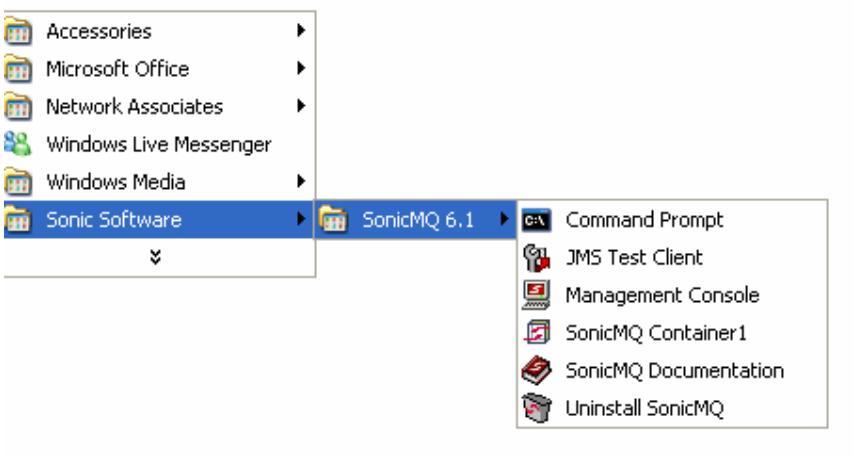

Figure 4.2 : Menu Items for Sonic MQ Server

You can start the management console with the shortcut *Management Console* and you can start the actual JMS server with the shortcut *SonicMQ Container.* We can also define the destination topics with the management console but this functionality is also included in the admin module of our news feeder. So below is the screenshots on how to define a connection factory and destination topic and which menu item you should click to open the administrative objects management screen.

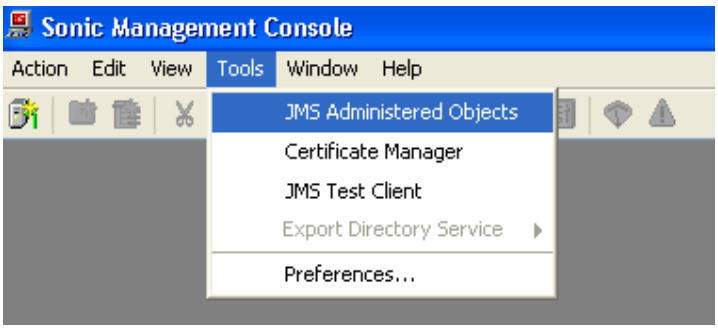

Figure 4.3 : Menu Item for JSM Administered Objects

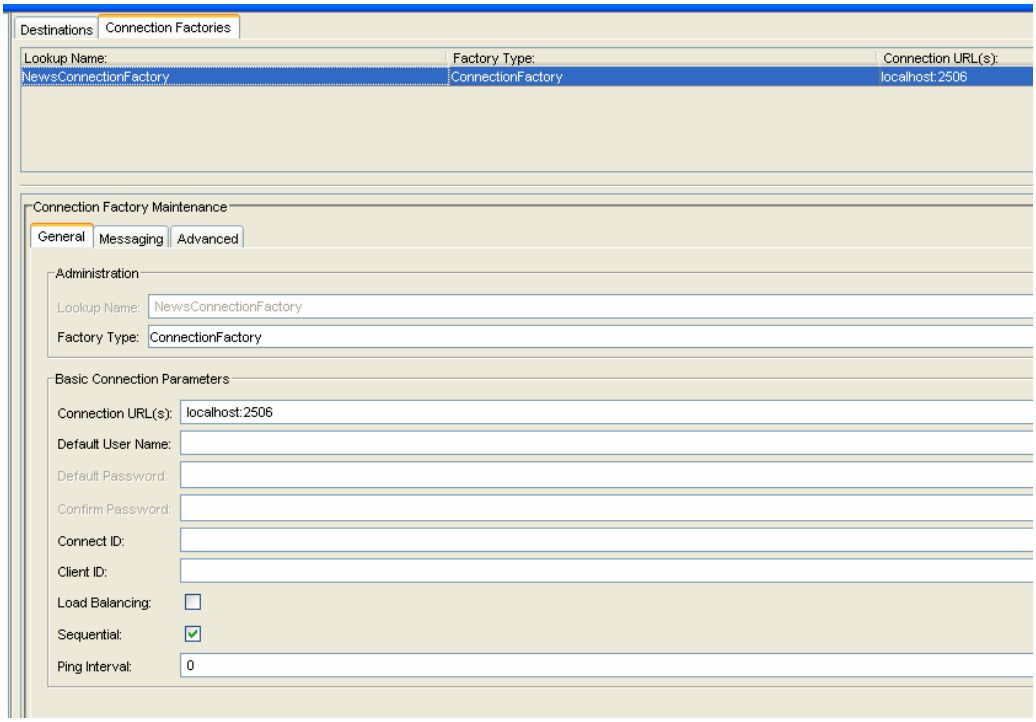

Figure 4.4 : How to Define a Connection Factory

As can be seen from the figure above, we define a connection factory names *NewsConnectionFactory* using the administrative tool. We could similarly define destination topics on the destination tab. Below is the screenshot for how to define a destination.

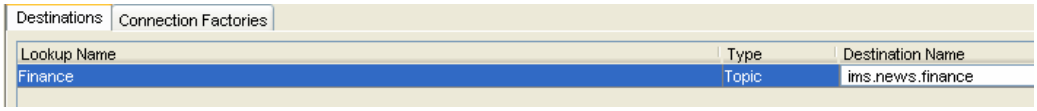

Figure 4.5 : Define the Finance Topic

4.2 News Feeder Application

After having configured the administered objects we can start the admin module by double clicking the *admin.bat* file. The admin module will pop up a login dialog on startup which asks for a profile name (to store profile information) and username and passwords for connecting to the JMS server.

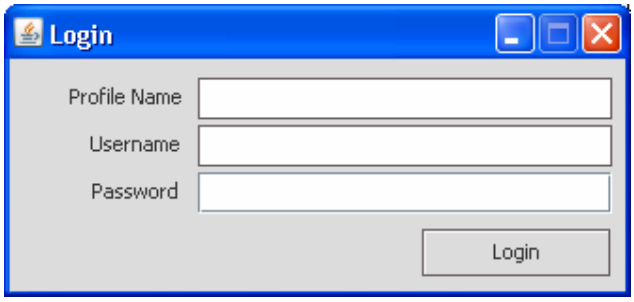

Figure 4.6 : Login Dialog

After logging into the system the admin module main screen opens. The admin module has two main functionalities which are provided in the two main tabs. With the admin module you can manage the topic objects which are news topics such as Finance, Art, Sports and so on and you can publish messages or web links to the topics. Below is the figure where you can see the topic management screen. The admin actually connects to the JMS server and programmatically defines topics on the server. You can define new topics or remove a defined topic from the server.

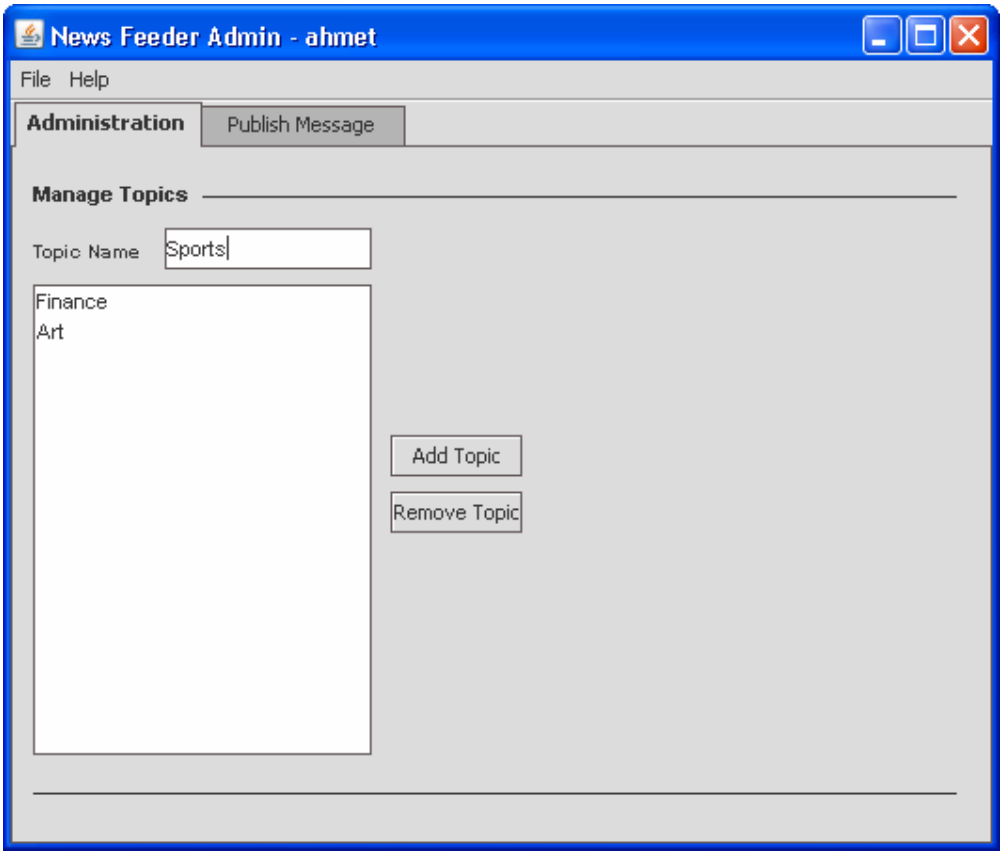

Figure 4.7 : Defining Topics via Admin Module

On the publish message tab, the admin can publish messages to the defined topics. The user selects the topic from a combo box, enters the subject into the subject textfield. The user either enters the news data or the link to the news in the body text area. If the body is a hyperlink, the user should check the web link checkbox. When the producer publishes the message via this screen it is sent to the JMS server middleware software thus providing the asynchronous communication. The clients could be offline and not available but the admin module shown here does not initiate a connection with the client so the operation is asynchronous.

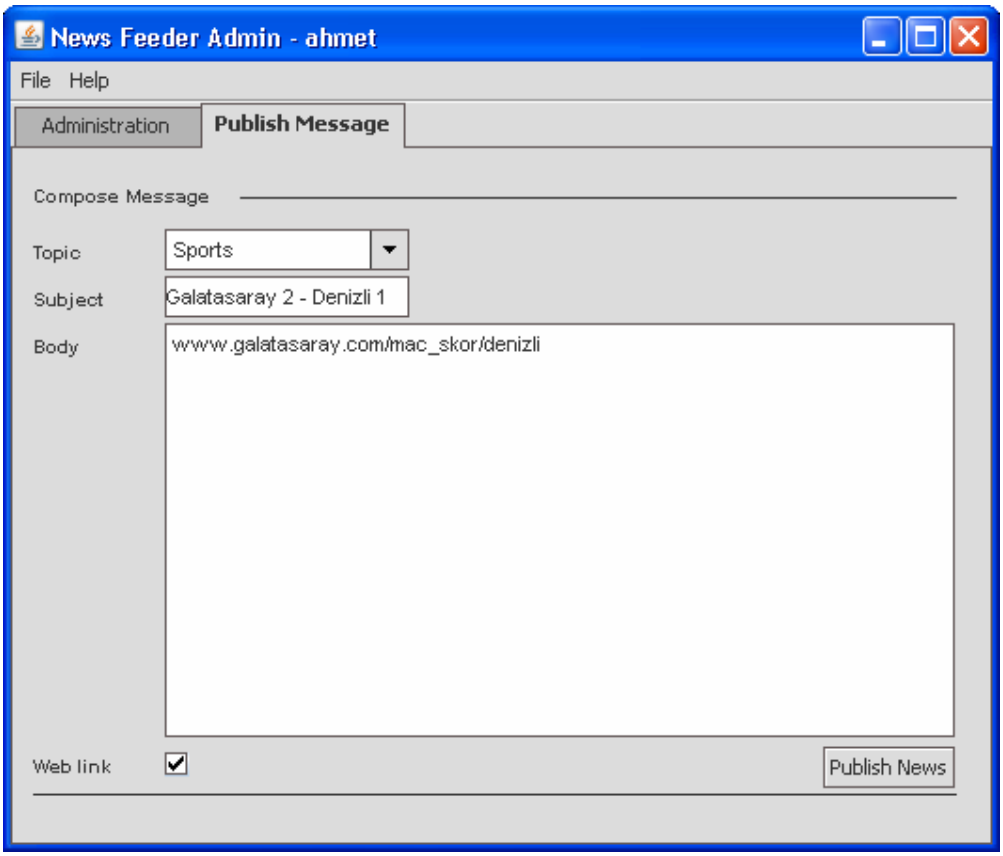

Figure 4.8 : Publishing a Message to the Sports Topic

The client module is the application which consumes messages from the subscribed topics. The client application has four main functionalities which are :

- Subscribing and unsubscribing to topics.
- Configuring message settings.
- Configuring message selectors for topics.
- Deleting messages.

Below is the dialog where users can subscribe or unsubscribe to topics. Via this screen the client is showing its interest in the events being published on a specific topic. This screen is where the client only needs to perform an action to receive messages and do it only once.

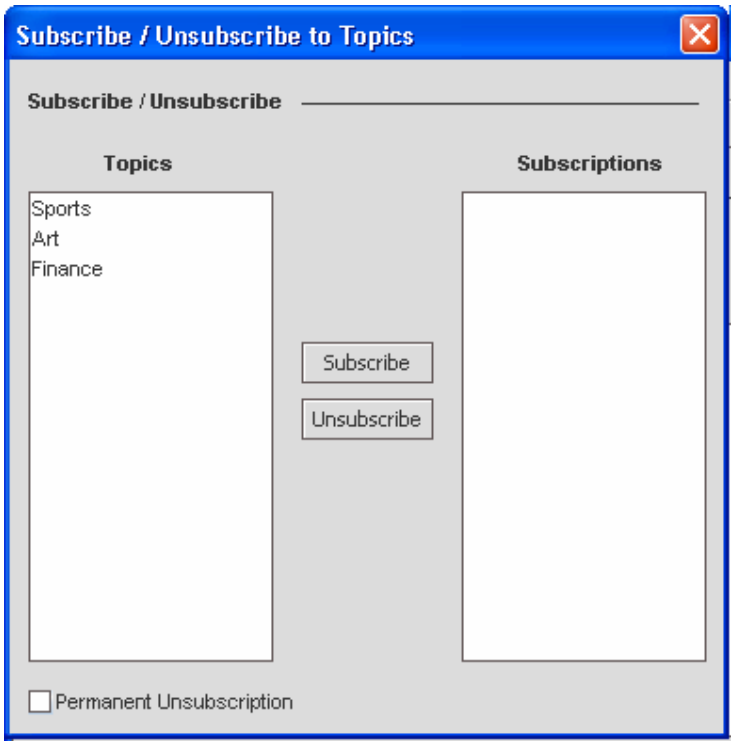

Figure 4.9 : Manage Subscriptions Dialog

On this dialog the client looks up via JNDI to the available topics and subscribes to them by selecting a topic from the list and pressing the subscribe button. One important detail is when unsubscribing the user must click the permanent unsubscription checkbox to remove the subscription permanently. Otherwise only the subscription will be closed. Below is the screen where the user can configure the message font and color settings.

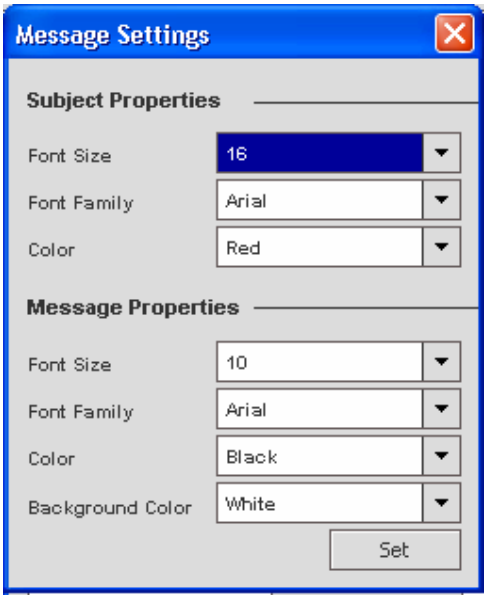

Figure 4.10 : Message Settings Dialog

Another functionality is to define message selector keywords for the topics. The clients can maintain a list of keywords for each topic the client has a subscription for. This way the client will only receive the messages if the subject property of the message contains a word from the keyword list. Below is the dialog to maintain the message selectors. The message selection is processed on the server side so bandwidth usage is limited this way.

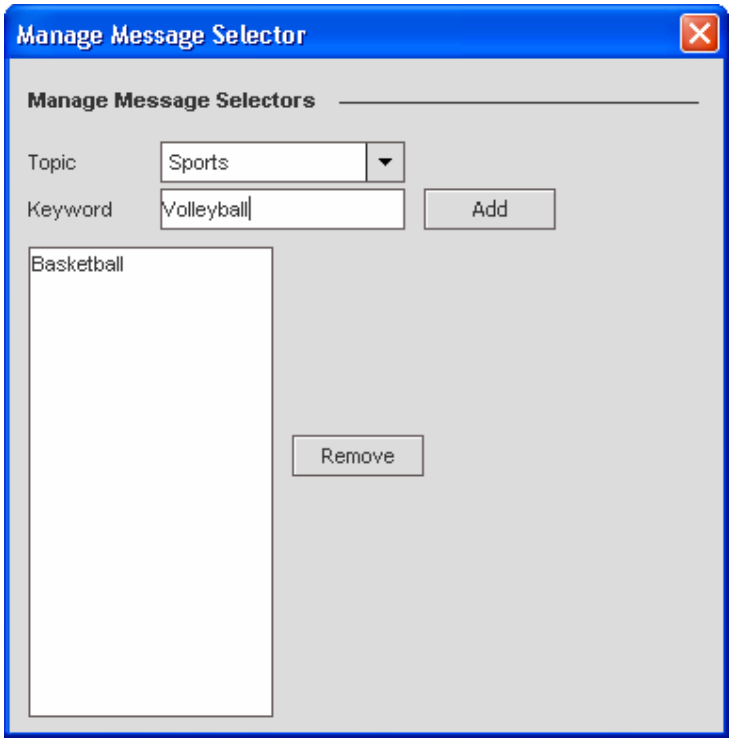

Figure 4.11 : Message Selectors Management Dialog

And below you can see a screenshot where the client is displaying a received news message. As the messages are published, if the client is online than the message will be pushed to the client the time the event occurs, so we can achieve our goal which was about the clients not performing any look ups.

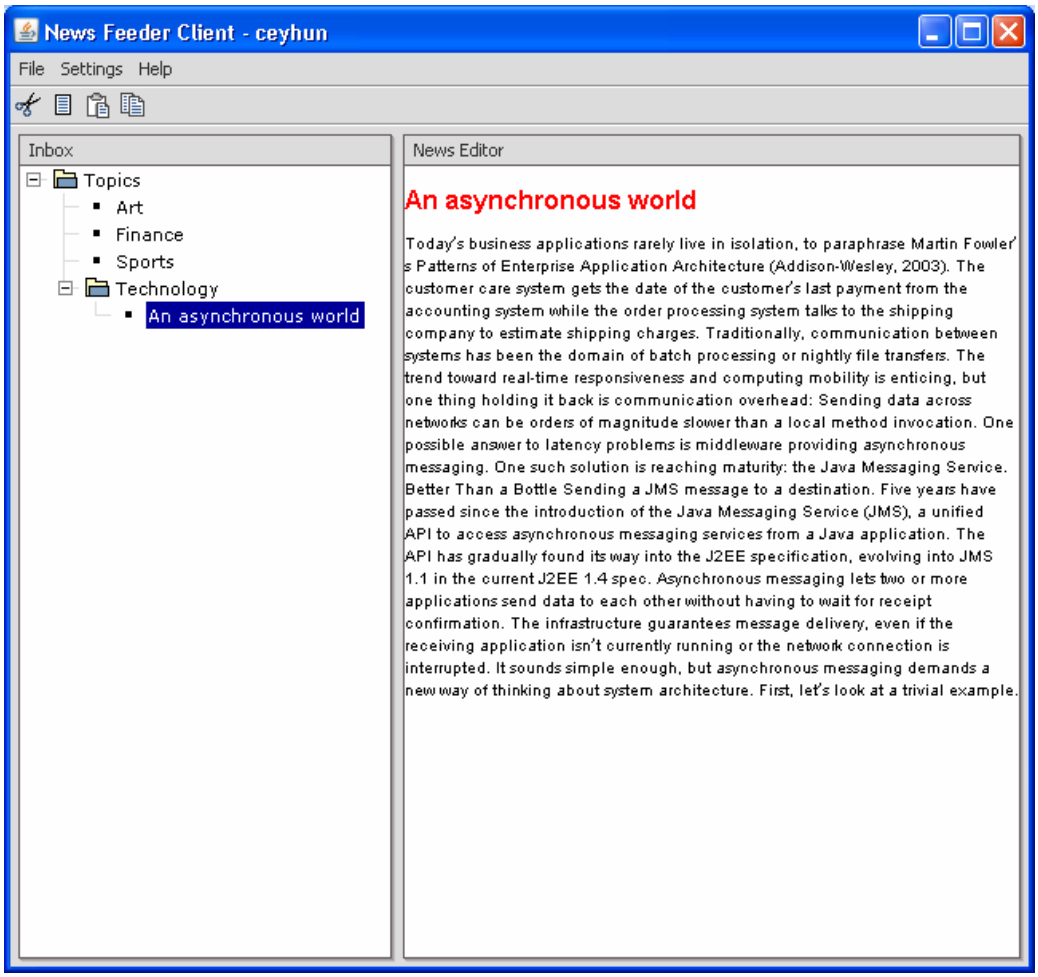

Figure 4.12 : News Feeder Client

4.3 Code Samples from the Application

Below are two code snippets from the news feeder client application where the JMS initializations takes place and asynchronous message receiving takes place.

The method shown below is the method used to perform all the initialization operations. As can be seen in the code, the first operation of every JMS application is to obtain a context object. Through a look up on the context we can get a reference to a connection factory object defined on the server side. This factory object will provide us to connections and sessions which we will be using to send and receive messages. After obtaining the required objects, the method retrieves the topic objects from the server side again with look ups. For each subscription the operations are repeated. Using these topic objects new subscribers are created and stored in a hash table. The need to store these subscribers is the need that they should be closed in case of an unsubscription operation. Also the message listeners of the subscribers are set. This means that which part of the application will deal with the incoming messages.

```
// Initialize JMS connections, sessions, topics, etc ...
private void setJMSConnections() throws AuthenticationException {

    try {

        context = new InitialContext(getInitialEnvironment());
        connectionFactory = (ConnectionFactory) context.lookup(CF_LOOKUP_NAME);
        connection = connectionFactory.createConnection (userName, password);
        connection.setClientID("ID_1");

        subSession = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

        if (subscriptions.size() > 0) {

            Topic topic                = null;
            MessageConsumer subscriber = null;

            for (int i = 0; i < subscriptions.size(); i++ ) {

                topic = getTopicObject(context, subscriptions.get(i));
                subscriber = subSession.createDurableSubscriber(topic, subscriptions.get(i) +
                             "_" + profileName,getSelectorString(subscriptions.get(i)), true);
                subscriber.setMessageListener(this);
                subscribers.put(subscriptions.get(i), subscriber);
            }
        }

        connection.start();
    } catch (AuthenticationException aue) {

        throw aue;
    } catch (Exception e) {

        e.printStackTrace();
    }
}
```

Figure 4.13 : JMS Initializations

The method below is the method that every class must implement if the class is implementing the interface MessageListener. This message is called by the server when a new message is received by placing the receive message as a parameter to the method. This method is also called in a seperate thread so receiving a message is also asynchronously done.

```
// Method that handles received messages
public void onMessage(Message aMessage) {

    TextMessage msg = (TextMessage) aMessage;

    String topic                      = null;
    DefaultMutableTreeNode node       = null;
    DefaultMutableTreeNode parentNode = null;

    FeederMessage feederMsg           = null;

    feederMsg = new FeederMessage(msg, this);
    messages.add(feederMsg);

    topic = feederMsg.getTopic();
    node = new DefaultMutableTreeNode(feederMsg);
    parentNode = (DefaultMutableTreeNode) rootNode.getChildAt(getNodeIndex(topic));
    parentNode.add(node);
    treeModel.reload(parentNode);
    expandTree();
}
```

Figure 4.14 : Asynchronously Receive Messages

5. CONCLUSION

We can clearly see the ongoing change on software and internet after this study. We think that our current use of the internet and applications will slightly change as we have more tools at hand. Messaging systems and asynchronous communications with aperiodic event based push capabilities will be the main focus of some web 2.0 applications. More and more information which only we are interested will be pushed to our computers with us spending minimum effort.

The developed News Feeder application is an example to this kind of aperiodic data publishing. The news feeder operates asyncrhonously which means that whenever the producer publishes a message, the consumers do not need to be online. The messages published are stored on a persistent store at the server side on the JMS server. So whenever the intended recipents come online, the messages are sent to the user with the order of being sent. This kind of behaviour removes the necessity of being online at the same time. Also since the message is pushed by the server, the client application need not pull any data, or perform any action on the GUI to receive the message. This contradicts with the current usage of web browsers where the user continously either clicks links or submit web forms to receive some information. Another important goal of the thesis was to achieve event based communication in the developed project. What we mean by this is that, the producer publishes an event the moment it occurs. So the event publish time is actually very close to event occurence time. This way the receivers get the information at a time very close to its occurence. This way a client does not have to poll a flight arrivals page at an airline companies web site, but rather the arrival of a flight will just be pushed to the interested clients at the moment it occurs.

JMS technology was selected to develop the news feeder application where JMS technology provides the asynchronous communication, reliability and loosely coupled architecture. The news feeder application is actually two different modules (applications) which do not know about each others existence. They do not know the web addresses of each other in order to communicate but rather they only know the URL of the JMS server. Another characteristic is that the applications are loosely coupled. They do not know any of each others API's or classes or methods to communicate. No interfaces are provided for communication also. Only the message format is the standard both application require to be able to communicate. Reliability also is handled in the JMS technology by the JMS server. While sending messages JMS uses various protocols and handshake mechanisms to guarantee that every recipent receives the message. So as a developer you can focus more on the business requirements rather than dealing with complex communication issues.

In the future we beleive that these kind of communication features will be embedded into the standard application we use like the web browser and the mail clients where we will be able to configure its properties and we will not have to adapt to completely different applications but rather use the applications which are familiar.

REFERENCES

[1] **Roberto S. Silva Filho, Max Slabyak, David F. Redmiles,** Web-based Infrastructure for Awareness based on Events, *Information and Computer Science, University of California.*

[2] **Amir Padovitz, Seng Wai Loke, Arkady Zaslavsky,** Using the Publish-Subscribe Communication Genre for Mobile Agents, *School of Computer Science and Engineering, Monash University*

[3] **Gero Mühl, Andreas Ulbrich, Klaus Herrmann, Torben Weis,** 2004. Disseminating Information to Mobile Clients Using Publish-Subscribe, *IEEE*

[4] **Victor Budau, Guy Bernard,** 2003. Runtime Support for changing communication model in large scale applications, *Institut National des Telecommunications, IEEE.*

[5] **Patrick TH. Eugster, Pascal A. Felber, Rachid Guerraoui, Anne-Marie Kermarrec,** 2003. The Many Faces of Publish/Subscribe, ACM Computing Surveys

[6] **Huang, Y. Garcia-Molina, H.** 2001. Publish/Subscribe in a Mobile Environment, *MobiDE*

[7] http://www.enterpriseintegrationpatterns.com/

[8] http://java.sun.com/j2ee/1.4/docs/tutorial/doc/

RESUME

Gökhan Büyükyumukoğlu was born in İstanbul on the 10.09.1979. After graduating from İzmir Türk College on 1998, he studied Computer Engineering at Dokuz Eylül University, İzmir. He started his graduate degree on Computer Engineering at İstanbul Technical University at 2003.