A Software Framework for Java Message Service Based Internet Messaging System

Hsiu-Hui Lee, and *Chun-Hsiung Tseng*
Graduate Institute of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
hhlee@csie.ntu.edu.tw,r90014@csie.ntu.edu.tw

Abstract:Java Message Service is released by Sun Microsystems as a standard way of message delivering and receiving. The purpose of this project is to implement a Java Message Service provider and provide a software framework to make some improvement on it. According to the specification of Java Message Service, Java Message Service is restricted in a pre-configured environment where each computer's IP is known in advanced and is completely connected. In this project, we design architecture to allow messaging in Internet. To integrate Java Message Service into Internet environment, there are several issues to consider:

To resolve Java Message Service destination objects without physical IP.

An efficient and stable method for message routing is required.

Load balancing should be considered.

The capability to handle heavy messaging load on Internet should be ensured.

Keywords: Java Message Service, Internet, server design

I. Introduction

Java Message Service [1] is a set of interfaces released by Sun Microsystem as a framework for enterprise messaging. There is currently some Java Message Service implementations (JMS providers) work fine in a simple LAN environment, such as "Java Message Queue" [4] and "Joram." [5] Through Java Message Service, the message sender and message receiver can both be a client. Also, they do not have to know exactly where each other is, the sender just sends messages to a "destination" and the receiver just receives messages from the same "destination." There are two types of messages supported by Java Message Service: one is "point-to-point" message and the other is "subscribe/publish" message. Before sending or receiving messages, the sender or receiver will first try to get a so called "ConnectionFactory" object, and get "Connection" object which represents real underlying connections from "ConnectionFactory" shown in figure 1. Senders and receivers will find the "ConnectionFactory" object with a pre-configured name, which is suggested to be similar with the "destination" name by the specification. However, according to the specification of Java Message Service, senders and receivers should either connect to the same Java Message Service provider or at least connect to providers having pre-configured direct connections with each other. This will be a big restriction when senders and receivers are put into an Internet. Still another problem is, senders and receivers should find their destinations through Java Naming and Directory Interface (JNDI) [4-5], which will be also an obstacle when messaging across Internet. Since JNDI will not only require a directly connected provider but also require a carefully chosen JNDI name to avoid misunderstanding and to help senders and receivers to find the correct objects.

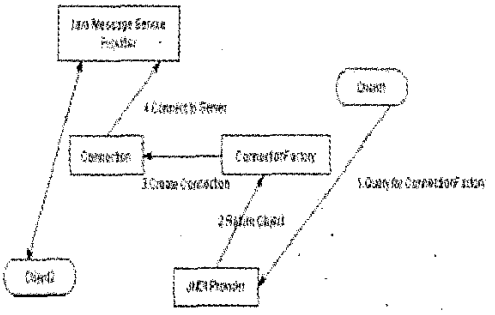

Figure 1 Traditional Java Message Service workflow

II. Design Goals

In this project, we try to refine the Java Message Service specification to be suitable in the Internet environment. We define "Internet environment" as a network environment, which may across a huge amount of WANs and LANs. In Internet environment, machines may have floating IP and connections between machines may be unstable. Therefore, to apply Java Message Service directly in such environment is impossible. We have the following design goals to achieve: design a method for resolving destination objects with floating IP, design a method for message routing in Internet, achieve load-balancing in Internet and modify the Java Message Service data structure to be applicable in a heavy messaging load environment. .

1 Resolving Destination Objects with Floating IP

A carefully designed look-up method is required if we want to use Java Message Service in Internet. Since the destination could be at any place and even with no physical IP address, only through JNDI to search the destination is not sufficient. In this project, we will design a remote look-up method can look up destinations in anywhere of the Internet. And this method should also take care of the traffic load. We don't want this cross-machine remote

lookup too expensive.

2 Remote Message Routing Method

We should design a method for remote message routing. Generally speaking, current JMS implementations require their message senders/receivers on directly connected machines, because we can only search for "destination" and "ConnectionFactory" objects in a LAN environment, which is unpractical when applying JMS to Internet. To require direct connections in that environment is quite reasonable, since it can benefit from the stability and speed of LAN environment. Traditional JMS implementations thus generally send and receive messages through direct connection. If we apply Java Message Service architecture in Internet, we have to consider message forwarding and routing, because direct connection between every machine in Internet is not reasonable. Further more, since Internet connection is far unstable than LAN connection, the method we designed here should take both routing speed and stability into consideration.

3 Load Balancing in Internet

The deploying system for JMS destination should have distribution ability. When considering messaging in Internet, too many JMS destinations on the same provider will make this Internet node a severe bottleneck. Also, if we store messages on this machine for future routing, this will drain this machine's resource. This problem is not addressed in traditional JMS implementations, since "point-to-point" messages in a LAN environment is quite efficient especially with direct connections; although "subscribe/publish" messages may require some queuing and buffering, the reliability of such network environment should greatly ease the problem. Internet environment is not as stable, so the message forwarding method will require enough buffering, this will make the problem even worse. Thus, allow users to configure and deploy their JMS destinations in distributed manner is in demand, otherwise the JMS implementation in Internet environment will be crippled.

4 Handle Heavy Messaging Load

When considering Internet environment, we may have lots of JMS destinations, which are participating in messaging process. Traditional JMS providers basically use memory alone as their storage. Although they may store their destination objects in corresponding JNDI server, the messages associated with those destination objects are directly put in memory and the real implementation of JNDI server may vary. The above situation works fine since they only work in a LAN environment where messages can be delivered as soon as possible and there are not too many destination objects. However, if we want to enhance Java Message Service to an Internet-level messaging system, we should consider more about the heavy load. In our project, we will provide a software framework that is more suitable in the Internet environment for JMS providers.

III. Architecture

In order to Java Message Service, a server adhering to the specification is required. Traditional Java Message Service specification requires only a single server as the messaging broker. However, the architecture will not work in Internet environment. For our design, there are three kinds of servers required for applying Java Message Service in an Internet environment. They are described below separately, and figure 2 is to depict the whole system architecture and the relationship of these servers. Furthermore, in order to make our messaging system applicable in Internet and keep the flexibility, we will introduce a software framework to design each server in the 5th section.

1 MainServer

Senders and receivers (JMS clients) should have direct connection to a local provider. However, we should extend the ability of traditional providers to work in Internet. We called this kind of provider a "MainServer." While sending or receiving messages, requests are first sent to MainServer, it will try to find the destination locally. If not found, MainServer will forward the request to its upper-level-server, which acts as a gateway.

2 GateServer

GateServers have two roles in our architecture. On one hand, a GateServer is a storage for the destinations shared by multiple MainServers,on the other, GateServer is also act as a router for routing requests and messages. More than one MainServers can connect a GateServer, and a GateServer itself can connect to another GateServer. Further more, in order to achieve load balancing, a GateServer can have several attachable-mini-GateServers, which will be named as "MiniGateServers." When requests from MainServers arrive, the GateServer will try to locate the destination locally first, if not found, it will try to find the destination at attached servers, if still in vain, it will forward this request to its connecting GateServer.

3 MiniGateServer

The main purpose of MiniGateServer is for load balancing. Since Internet is not a stable network environment, messages may not able to be properly sent at some time and we have to store messages before they are successfully transmitted. However, this will cause a lot of overhead, and requires many system resources. With MiniGateServer, we can easily configure it to take over some destinations originally deployed on a GateServer. This can reduce that GateServer's load. Another contribution of MiniGateServer is to enable destinations on servers in a dynamical or virtual IP environment be accessible. A MiniGateServer can dynamically attach to a GateServer, thus, even if the IP of the MiniGaServer is changed, clients through the attached GateServer can still access it.

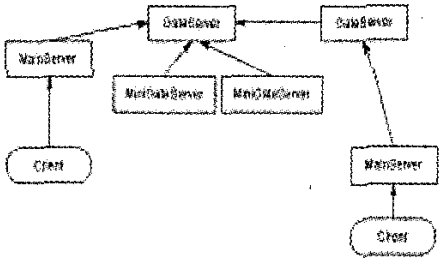

Figure 2 Server Architcture

IV. Message Flow

A messaging system should handle thousands of messaging efficiently and safely. According to the Java Message Service specification, messages are classified into two catalogs: "Point-to-Point" messages or "Subscribe/Publish" messages; we will describe how they are handled in our design below.

1 Point-to-Point Message

In Java Message Service, a point-to-point message is called a "queue" message. This type of message will have exactly a sender and a receiver.

In a virtual/dynamic IP environment, the target destination may be on a machine with no physical IP address. Thus, the target destination cannot be located in traditional JMS implementations. In our implementation, we use "MiniGateServer" to hold these destinations, and requests will be routed to these MiniGateServers through GateServers. Suppose we have a queue destination on GateServer G3, and our JMS client is connected to MainServer M1, which is connected to GateServer G1. The GateServer G1 is connected to GateServer G2 as its outlet that in turn connect to GateServer G3. This set-up will be depicted in Figure 3. Now, suppose the JMS client is sending or receiving the queue message from the queue destination. The request will first be forwarded to MainServer M1, and M1 will find the requested target is not a local destination, and then the request is forwarded to GateServer G1. G1 will check the destination again, and will still find the destination is not a local one. Then G1 will create a routing packet, set the time-to-live information, and route the request to G2. After G2 received the packet, it will check the time-to-live information to make sure this packet is not out-dated. If the packet is still alive and the target destination is still not local to G2, G2 will continue routing the packet. If the packet is successfully routed to G3, the target destination will be found. If the original JMS client requests for sending message, then this message will be stored in G3, and if the JMS client requests for receiving messages, all messages for the target destination will be return to M1 to make future receiving more efficiently.

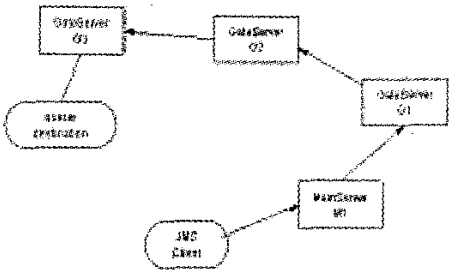

Figure3 The Point-to-Point message set-up

2 Subscribe/Publish message

In Java Message Service, subscribe/publish messages are also called "topic" messages. Unlike queue message, which is "client request for receiving", the JMS provider should automatically transmitting messages to subscriber clients without clients' request.

Traditional JMS implementations still have problems when the target destination does not have physical IP address. Furthermore, The nature of Subscribe/Publish message will also make implementing Java Message Service in a virtual/dynamic IP environment much more difficult, since the subscribers may come from every place with dynamic IP address. If we just record the subscribers' current IP address and forward messages to all client machines, and check the client's existence at the same time, this will produce incredible overhead. To overcome that problem, we design a "multi-level forwarding method." We will describe this method below.

Suppose the topic destination is at GateServer G2, and a GateServer G1 connect to G2, and a MainServer M1 connect to G1. Several subscribers connected to M1 subscribe to the topic destination. The set-up will be depicted in figure 4. When M1 receives the request, it will register itself as a "proxy entry" in G1. G1 will first check if the topic destination is a local one and after G1 assure the destination is a remote one, it will route the request to G2 and register G1 itself as a "proxy entry" in G2. There may be thousands of MainServers connected to G1 and they may all subscribe to that topic destination. When G2 publish messages, it will publish to those "proxy entries" instead of publishing to all subscribers. Of course, after messages arriving the MainServers, MainServers should still forward messages to all subscribers. However, we assume MainServers is as close to clients as possible, thus this "multi-level forwarding method" will decrease the numbers of packets which transmitting messages along a long Internet path. Furthermore, our publishing thread will automatically schedule the publishing interval. Thus if we have only a few messages to publish, it will increase the publishing interval and try to collect as many messages as possible in a publishing session. This will also increase the performance.

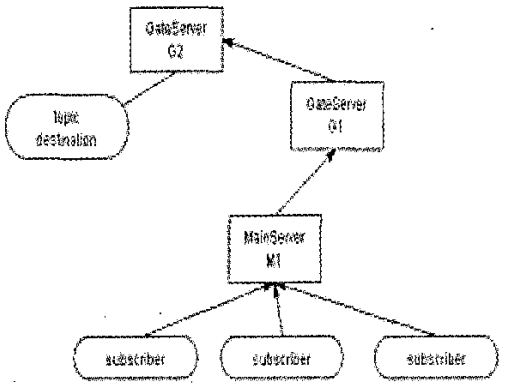

Figure 4 The Subscribe/Publish message set-up

V. Heavy Load Destination Storage

Up to now, we have described the server architecture and the message flow of our messaging system. In our design, the server is able to handle messaging hierarchically, and we have discussed load-balancing issues. However, for each server, we have assumed that all destination objects and messages are stored in memory until now. This may not be the case when we consider a real Internet world. In order to handle the heavy load elegantly, we may want to store JMS destinations and their associating messages into different location, different disks, and even different database. And due to the unreliability of Internet connection, transferring messages may fail, and we may want to ensure messaging of some JMS destinations more strictly. That is, we want to have different mechanisms to manage different JMS destinations and their associating messages. Furthermore, traditional JMS providers only support TCP connection, but we may need different connection type for different group of JMS destinations, such as UDP, HTTP, and SSL... etc. Below we will provide a software framework to achieve the goals.

1 JMS Server Framework

Java Message Service is a standard established by Sun Microsystems as a messaging framework. However, according to the specification, a Java Message Service server is a prerequisite for using Java Message Service. Of course, there are many commercial or non-commercial products now, but if we want to use Java Message Service in a particular situation, for example, using it in an embedded system or using it to facilitate internal messaging between components in an application, we still need to build our own Java Message Service server. In our project, we have several levels of servers working together to form a complete messaging system. To write a Java Message Service server is a quite tedious work, moreover, different servers may work on different host in different environment, optimization for each case is quite difficult. Thus, we have designed a flexible architecture for easier building process of Java Message Service server.

2 Architecture

The architecture can be categorized into following parts:

1. Server

A Server is the control center of the whole framework. It will contain several pairs of ConnectServers and ConnectOperators and manage their lifecycle. A Server will also maintain a DestinationStore as storage of Java Message Service destinations.

2. ConnectServer and ConnectOperator

For flexibility, our Java Message Service Server Frame work will not enforce a specific protocol for client connection. Instead, we use ConnectServer/ConnectOperator pairs to fulfill connection work. A ConnectServer may choose any proper protocol for client connection; for example, a ConnectServer can be designed as a HTTP server to interact with clients from internet, and a ConnectServer can also act as a mail server to receive clients' request as e-mail. However, a ConnectServer should only handle connections, that is, it should not directly process request from clients. A ConnectServer will forward any client request to its corresponding ConnectOperator, thus, a ConnectOperator can be used by more than one ConnectServer. ConnectOperators will have reference to DestinationStore for Java Message Service operations.

3. DestinationStore and DestinationOperator

A DestinationStore is actually a hash table storing destination/DestinationOperator pairs. In our design, Java Message Service destinations and its corresponding messages can be stored at any storage. Thus, we can use arrays, files, and even databases to store those messages. For flexibility, different storage can be accessed through different DestinationOperator, thus we can use storage with higher speed for destinations with high loading and use storage with grow-able size for destinations may have to store huge amounts of messages in future.

4. DriverPool and Driver and DriverConnection

Clients for traditional Java Message Service Server typically rely on JNDI only to search their target destinations. However, since our frame can be applied on various environment even embedded systems, developers using our framework may want to use other methods for searching. We have wrapped searching-object methods as Driver; clients can get DriverConnection from DriverPool, and use DriverConnection as a standard interface to search objects. The underlying searching methods is hidden in Driver, thus provide both flexibility and convenience.

5. System-independent JMS libraries
6. System-dependent JMS libraries

We have separated our implementation of JMS libraries into two types: system-independent and system-dependent. This is because some JMS objects, like Message object, Destination object, are quite general, while others, like Connection object may be totally different according to different environment. By separating these two types of JMS libraries, we can have better re-usability in our implementation.

Figure 5 will depict the UML class diagram of our software framework:

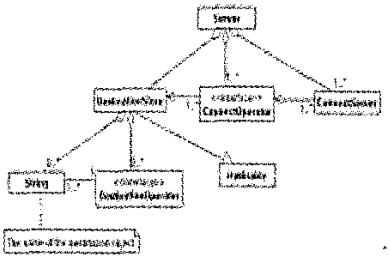

Figure 5: JMS server framework UML class diagram

VI. Testing

The following is the testing data. First of all, we test the performance for messaging on a single machine. And then, we test messaging through Internet with a 512K ADSL network to see the difference. The testing scenario is to send 50 TextMessage, 50 ObjectMessage, and 50 BytesMessage (the message format is according to Java Message Service specification) through a Queue. We run the test five times for both single machine case and Internet case. We measure the elapsed time in milliseconds and illustrate the result in figure 6.

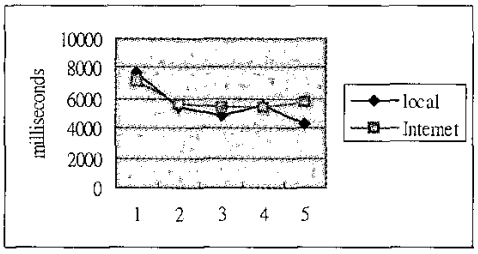

Figure 6: Testing data

VII. Conclusion and Future Work

We have designed a "maximized" JMS provider, and it can be used in an Internet environment. We have considered the performance and stability in Internet, and have developed some methodology to meet Internet constraint. Problems like virtual-IP, dynamic-IP, and load balancing are also handled in our provider. In the future, we still have many features to consider: increasing transmitting speed with compression method, more efficient broadcasting method, etc.

References

[1] Java Message Service version 1.0.2 specification, November 9,1999, Sun Micro System.

[2] The JNDI Tutorial by Rosanna Lee.

[3] The SLAPD and SLURPD Administrator's Guide,30 April,1996,University of Michigan.

[4] JORAM Tutorial,http://www.objectweb.org/joram

[5] http://wwws.sun.com/software/Developer-products/iplanet/jmq.html