Name of workshop

B Workshop on Computer Networks

Title of the paper

Applying Java Message Service in Virtual/Dynamic IP Environment

Abstract

The purpose of this paper is to implement a Java Message Service provider and to make some improvement on it. According to the specification of Java Message Service released by Sun, Java Message Service is restricted in a LAN, a single computer, or at least a pre-configured environment where each computer's IP is known in advanced and is completely connected. Here, we design architecture to allow more flexible usage of Java Message Service. For example, we allow messaging in the internet and even in a virtual-IP environment. More over, we add some modern server technology, like load balancing on our architecture. After all, we implement the entire architecture in pure java.

Author

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

tel: (02)-23625336 ext 511

fax: (02)-23628167

Keywords

Java Message Service, virtual IP network, server design

Applying Java Message Service in Virtual/Dynamic IP Environment

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

Keywords: Java Message Service, virtual IP network, server design

The purpose of this paper is to implement a Java Message Service provider and to make some improvement on it. According to the specification of Java Message Service released by Sun, Java Message Service is restricted in a LAN, a single computer, or at least a pre-configured environment where each computer's IP is known in advanced and is completely connected. Here, we design architecture to allow more flexible usage of Java Message Service. For example, we allow messaging in the internet and even in a virtual-IP environment. More over, we add some modern server technology, like load balancing on our architecture. After all, we implement the entire architecture in pure java.

1. Introduction

Java Message Service [1] is a set of interfaces released by Sun Microsystem as a framework for enterprise messaging. Currently, there are some Java Message Service implementations (JMS providers) which 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 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.

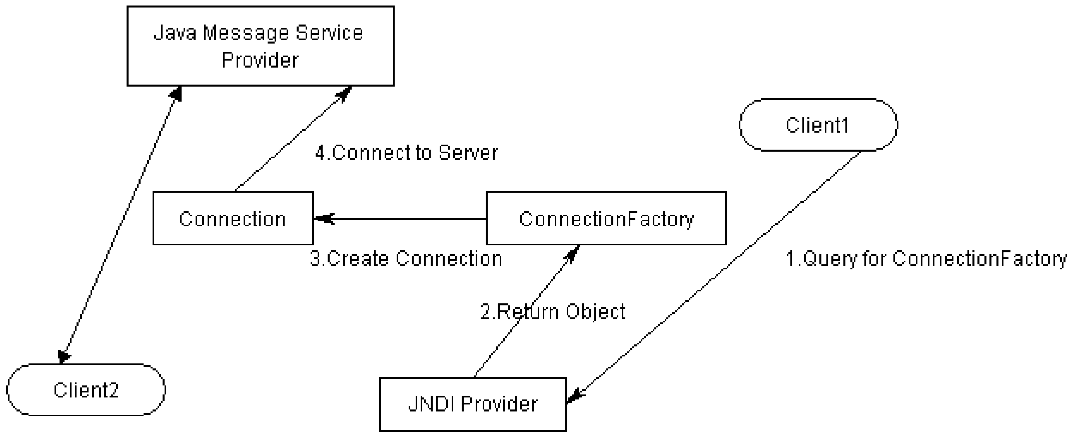

Figure 1 Traditional Java Message Service workflow.

2. Design Goals

2.1 Remote Look-up Method

To look-up an object in a virtual/dynamic IP environment is more difficult than that in a simple LAN environment. In a LAN environment, a simple naming and lookup method is sufficient since the name of each object just takes effect in a restricted network environment. Furthermore, to avoid name conflicting is much easier and since all naming is applied just in a certain domain (for example, the same JNDI provider), the look-up method can only take care of only a few machines. However, if we want to use Java Message Service in a virtual/dynamic IP environment, since the destination could be at any place, only through JNDI to search the destination is not sufficient. In this paper, we will design a remote look-up method can look up destinations in internet anywhere. And this method should also take care of the traffic load, because we don't want this cross-machine remote lookup too expensive.

2.2 Remote Message Routing Method

2.2.1 Message Routing in Traditional JMS Systems

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 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.

2.2.2 Message Routing in JMS Systems within Virtual/Dynamic IP Environment

If we apply Java Message Service architecture in a virtual/dynamic IP environment, we have to consider message forwarding and routing, because direct connection between every machine in that environment may not be possible. Further more, since connection is far unstable in that environment than in LAN, the method we designed here should take both routing speed and stability into consideration.

2.3 Distributed JMS Objects

The deploying system for JMS destination should have the distribution ability. Since we have to consider load balancing and stability in a virtual/dynamic IP environment, too many JMS destinations on the same provider will make this 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. Our target environment may not be 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 are in demand.

3. Architecture

In our opinion, there are three kinds of servers required for applying Java Message Service in an internet environment. There are MainServer, GateServer, and MiniGateServer. Below we describe them in detail. Figure 2 depicts the whole architecture.

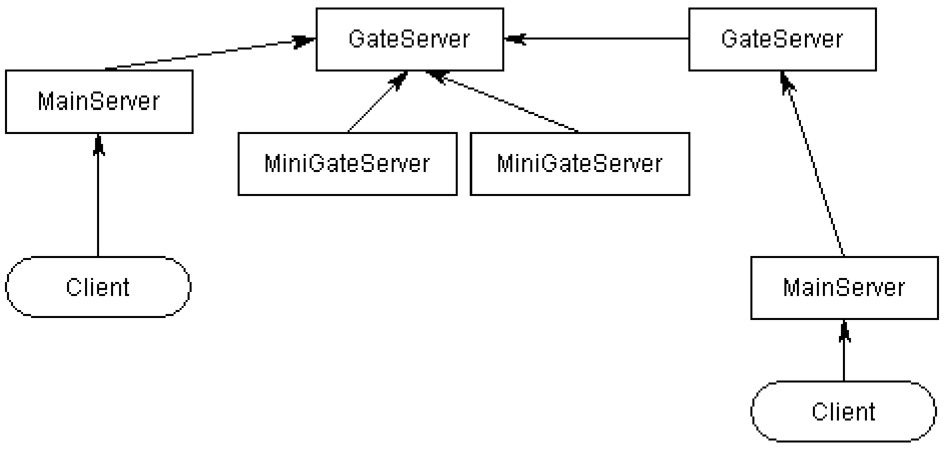

Figure 2 System architecture.

3.1 MainServer

Senders and receivers (JMS clients) should have direct connection to their local provider. Here, we should extend the ability of traditional providers to work in internet. We call 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 it is not found, MainServer will forward the request to its upper-level-server i.e. the GateServer.

3.2 GateServer

GateServers have two roles in our architecture. On one hand, GateServer is a storage for the destinations shared by multiple MainServers; on the other, GateServer also acts 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 its attached MiniGateServers, if still in vain, it will forward this request to its connecting GateServer.

3.3 MiniGateServer

The main purpose of MiniGateServer is for load balancing. Since a virtual/dynamic IP environment may not be 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 require a lot of overhead and 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 MiniGateServer is changed, clients through the attached GateServer can still access it.

4. Message Flow

Below, we will describe how a remote message is forwarded in our design.

4.1 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. We will go deep into this issue in section 5.

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.

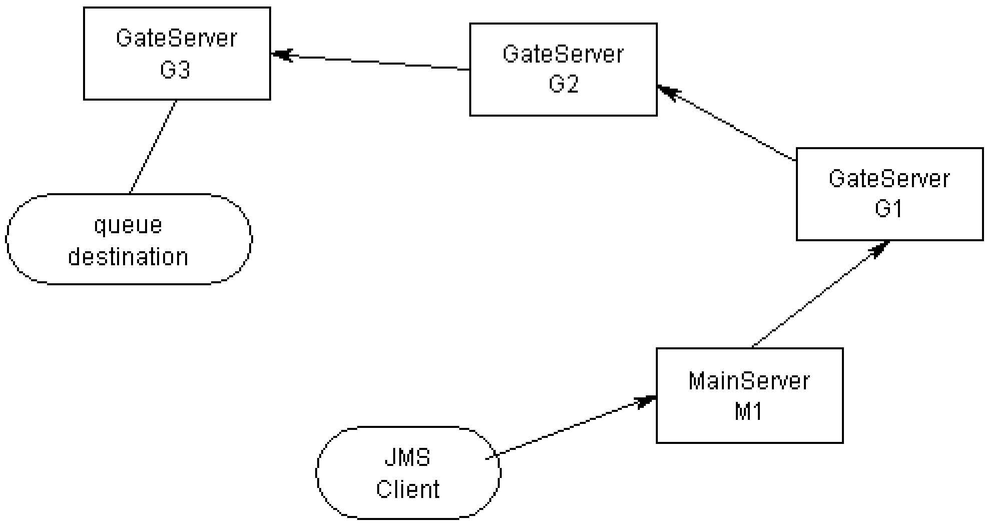

Figure3 The Point-to-Point message set-up.

4.2 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.

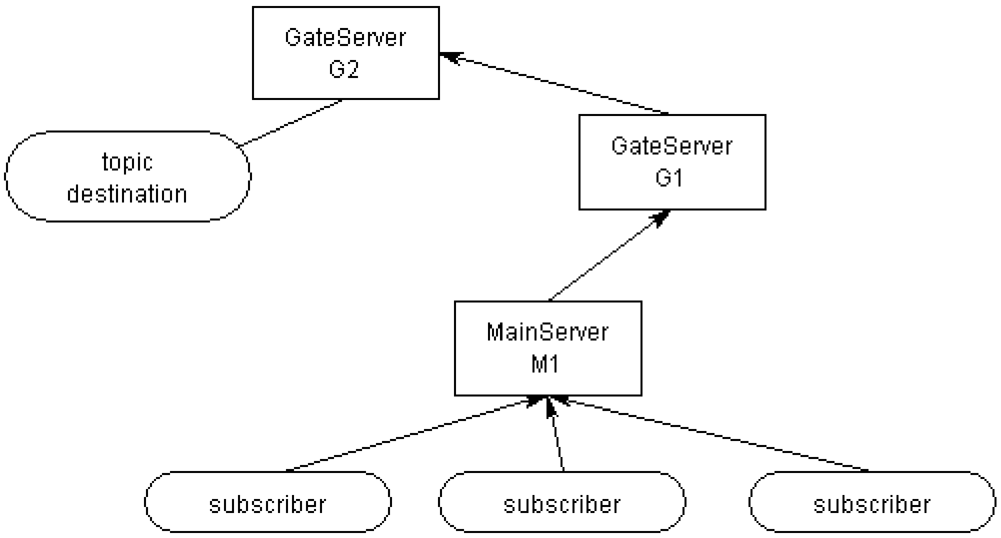

Figure 4 The Subscribe/Publish message set-up.

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.

5. Virtual/Dynamic IP

5.1 The Scenario

The "Virtual/Dynamic IP" problem is a big challenge for Java Message Service provider development. This is because JMS client will typically locate their service provider with a network address and a port number. Furthermore, traditional JMS providers will require a JNDI provider to register destination objects and ConnectionFactory objects. Those JNDI providers will need a physical IP address since clients will search for objects through them. Generally speaking, according to the specification of Java Message Service 1.0.2, a physical IP address and a phone number will identify all services provided in a Java Message Service environment. This will restrict JMS to be only used in a LAN environment, since virtual/dynamic IP is almost everywhere outside the LAN.

5.2 Our Solution

However, our design can solve the virtual/dynamic IP problem at a certain degree. Through our "hierarchical searching method", clients need not to depend on JNDI to locate the destination objects and ConnectionFactory objects. With a properly designed canonical name, client can find every object with the hierarchical searching method. Of course, our JMS provider still allows searching objects through JNDI, this will obviously be faster and less error prone. But, if the JNDI provider is in a virtual/dynamic IP environment and cannot be identified by physical IP address, our design stands there to solve the problem.

Moreover, if the JMS provider itself is in a virtual/dynamic IP environment, this may even prevent clients from accessing the services provided by the provider. To this problem, our design also renders a part solution. In our architecture, a GateServer can have several MiniGateServers attached and provide service. Thus, JMS providers in virtual/dynamic IP environment can attach themselves to some GateServers with physical IP address. Clients can access those providers indirectly. Although this will certainly put some drawback on performance, those JMS providers are shielded from some attack since clients do not directly access them. However, our solution in this aspect is, of course, a part solution, since a universal GateServer with physical IP address is still required.

6. Conclusion and Future Work

We have designed a "maximized" JMS provider, and it can be used in a virtual/dynamic IP environment. We also consider the performance and stability in a virtual/dynamic IP environment, 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 will add more features like applying JMS in embedded systems, decreasing message-transfer load by compression, or enhancing the deployment interface, *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, April 30, 1996,University of Michigan.
4. JORAM Tutorial,http://www.objectweb.org/joram
5. Java Message Queue implementation http://wwws.sun.com/software/Developer-products/iplanet/jmq.html