JINI/J2EE Bridge for Large-scale IP Phone Services

Jia Yu[†]
Monash University
Melbourne, Australia
jiayu@cs.mu.oz.au

Jan Newmarch
Monash University
Melbourne, Australia
jan.newmarch@infotech.monash.edu.au

Michael Geisler
Sun Microsystems
Melbourne, Australia
michael.geisler@sun.com

Abstract

Current IP phone architectures which bring phone services to a distributed open environment are eraltively static and do not scale well. An IP phone environment should be more dynamic. Using Jini to be middleware for IP phone services will make the IP phone environment more portable, easier to deploy and straightforward to extend. However, many global applications with complex business logic and potentially thousands of concurrent users are developed on J2EE compliant platforms. On the one hand, J2EE is a framework to support large-scale systems while Jini is for small to medium-scale applications. On the other hand, J2EE provides a centralized service whereas Jini offers loosely-coupled federations with dynamic administration. In this paper, we propose an architecture called JINI/J2EE bridge for marrying these two technologies, and allowing J2EE applications to be accessed by Jini.

1. Introduction

The development of conventional telephony systems is far behind the development of today's Internet. Centralized architectures (see Figure 1.a) with dumb terminals make exchange software and hardware very complex, but provide very limited functions. Closed and hardware property systems hinder the enterprise to choose products from different vendors and deploy a voice function to meet their business needs. Consequently, Web-like IP phone distributed architecture **[1]** is proposed to facilitate enterprises and individuals to provide their own phone services.

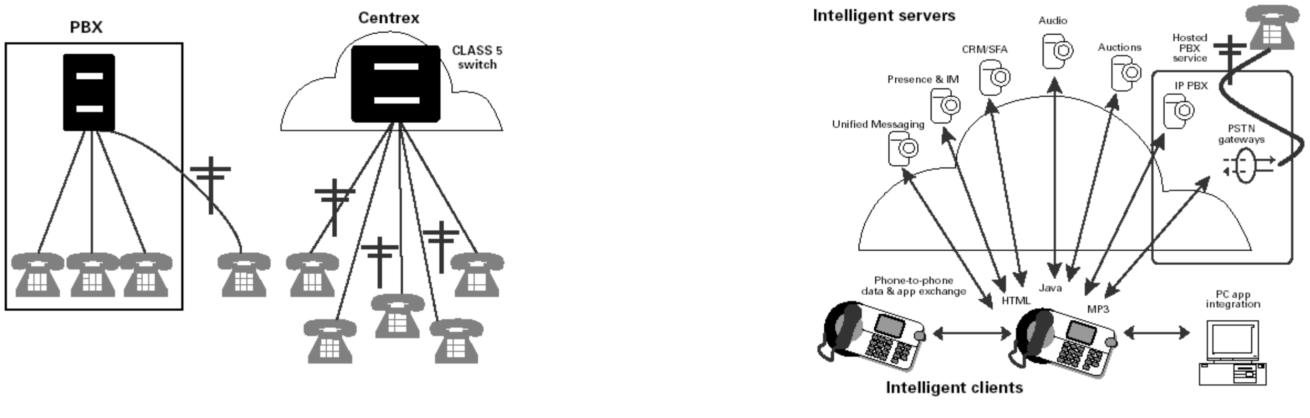
Web-like IP phone architecture (see Figure 1.b) is similar to WWW of today's Internet. An IP phone is an intelligent client. It can be a PDA, small programmable device or a desktop. Any individual or enterprise can provide their phone services. Thousands of innovative services can be deployed on IP networks for IP phones, as a huge number of websites appear on the Internet. For example, we can deploy an internal phone book service in an enterprise network. People search the phone book by their IP phones, and the phone can dial the telephone number automatically. However, how to manage and discover phone services is a challenging task.

To date, there is no standard for the phone service management and discovery. Compared with enterprise WWW service, IP phone network is more dynamic. Therefore, the underlying infrastructure for the IP phone services needs to be more portable, easy to deploy and straightforward to extend. Jini**[2]** is a good technology for device interoperability and network plug-and-play. However, the benefits of Jini are far more than for network devices; it also can contribute to network service discovery, especially for a dynamic network. By using Jini, a phone service can be discovered at run-time. Imagine if every company or institution deploys their own phone book service, when you get into a corporation, Jini phone can find its phone service and display it on the screen. Then you will easily know any staff telephone number and other information without concerning yourself with the location of the service and reconfiguration.

In addition to Jini services which are for small to medium-scale applications, large-scale phone services are also needed in IP phone system, for example monitoring service and customer information retrieval. Java 2 Enterprise Edition (J2EE)**[3]** is targeted at large-scale applications, integrating clients, web components, EJB components and database. However, J2EE provides a centralized service whereas Jini offers loosely-coupled federations with dynamics of administration. In dynamic phone environment, we need to access J2EE large-scale applications such as trouble reporting system, not only by Web and certain Java clients, but also by Jini, so that an IP phone can discover and access a J2EE application at run-time. Since there is no J2EE server supporting Jini access, in this paper we propose an architecture for marriage of these two technologies called JINI/J2EE

[†]Current contact address: Department of Computer Science and Software Engineering, Melbourne University

Bridge to facilitate the discovery of J2EE service on the Jini federation.



a. Traditional telephony architecture **b. Web-like phone architecture**

Figure 1. Telephony architecture[1]

The rest of this paper is organized as follows: Related work within an IP phone environment is presented in Section 2. The detailed system design is described in Section 3. Section 4 describes implementation with class diagram. A use-case study is presented in Section 5. We discuss the benefit and drawback of the system in Section 6. We conclude in Section 7.

2. Related work

The primary feature of voice application is that it is extremely delay-sensitive rather than error-sensitive. There are several approaches that have been developed to support delay-sensitive applications on IP networks. In the transport layer, UDP can be used to carry voice packets while TCP may be used to transfer control signals, as long delay is caused by TCP by its retransmission and three-handshake mechanism. The Real-Time transport protocol (RTP)**[4]** is a compensative protocol for real-time deficiency on packet networks by operating on UDP and providing mechanisms for real-time applications to process voice packets. The Real-Time Control protocol (RTCP)**[4]** provides quality feedback for the quality improvement and management of the real-time network.

Several signaling protocols have been proposed for IP phone applications. MGCP**[5]** and MEGAC **[6]**are old centralized models, while H.323**[7]** and SIP**[8]** are peer-to-peer protocols. H.323 is published earlier by ITU and gained more support in today's products. However, being simple and similar to HTTP, SIP will bring the benefits of WWW architecture into IP telephony and readily run wherever HTTP runs. It is a gradual evolution from existing circuit-switched networks to IP packet-switched network. Server gateways that convert VoIP signaling and circuit-switched network signaling are required. Potentially, MGCP and MEGAC are believed to work along with SIP. MGCP or MEGAC acts as a protocol for gateway control whereas SIP is used as call-signaling protocol.

Intelligent phone terminals are the basic component for Web-like phone system. Pingtel xpressa**[9]** is 100% Java-based IP phone. The xpressa is based on SIP and hence leverages its extensibility, scalability and ease of deployment. Several Java APIs for audio applications are also available on desktop, including Java Telephony API (JTAPI)**[10]**, Java Speech API (JSAPI)**[11]** and Java Media Framework API (JMF)**[12]**.

Integrated Network APIs for the Java Platform (JAIN)**[13]**, developed by a community of companies led by Sun Microsystems, is a set of integrated network APIs for the Java platform. JAIN architecture divides a network-related software system into several layers and puts forward open APIs between two adjoined layers. The JAIN architecture is targeted toward networks, signaling and services layers of wired, wireless and packet networks. Open standard APIs will enable different companies to provide solutions for different portions of the systems and hence allow customers to choose products from multiple vendors at different levels based on their requirements. Open standard APIs will also hide the complexity of networks and signaling protocol for high-level application developers, so a huge opportunity for new services on IP phone systems will be achieved.

OSS/J**[14]** is a set of standard Java APIs for Operation Support systems (OSS). OSS covers the software used by a service provider to create, administer and bill services on its network. Standard OSS APIs are expected to promote multi-vendor interoperability. OSS/J APIs are shown in Table 1.

Table 1. Summary of OSS/J APIs[15]

API	Application
Service Activation API	Provisioning a service
Quality of Service API	Monitoring service life-cycle
Trouble Ticketing API	Detecting and fixing faults
IP Billing API	Billing and discounting
Inventory API	Service, network and customer info retrieval

Current Jini implementation using dynamic classloading cannot be deployed in small devices, due to their limited resources. Jini surrogate architecture**[17]** provides proxy-based solution to address this problem. Other projects such as JMatos**[18]** and JiniMe**[19]** also provide solutions to implement Jini on small devices.

Emerging networking, signal standards and intelligent terminals enable high performance Web-like VoIP system. Layered standards for phone software and operation support systems facilitate vendor interoperability and productivity of innovation phone services. Solutions for Jini on small devices contribute to produce Jini-based phone device. As compared to related works above, we focused on combining Jini and J2EE technology, so that IP phone can access large-scale phone services, such as OSS/J applications, without concern of their location.

3. JINI/J2EE Bridge

Our goal is to facilitate the discovery of J2EE services on the Jini federation. Through an independent program running outside J2EE application server -a Jini/J2EE proxy for a J2EE application- the J2EE application can be accessed by a Jini client. However, it is not a good idea that people have to run the proxy separately after deploying their J2EE application on a J2EE server. It is better that JINI/J2EE proxies can be run automatically. For example, when a J2EE application is dbbeployed on a J2EE application server, the corresponding Jini/J2EE proxy runs and registers this J2EE service to the Jini lookup service. Once this application is undeployed from the J2EE application server or disconnection occurs between the proxy and the application server, that proxy should disappear from the Jini network until the application appears on the J2EE server again.

3.1. Architecture

The architecture of the JINI/J2EE Bridge is illustrated in Figure 2. Every J2EE application has its proxy and runs in the bridge. A JINI/J2EE proxy represents a J2EE application on the Jini federation and can communicate with its J2EE application by Remote Method Invocation (RMI)**[22]** or Java Message Service (JMS)**[21]**. There are two major entities of the JINI/J2EE Bridge, a JINI/J2EE proxy manager and a proxy container.

- The JINI/J2EE proxy container holds resources and provides a runtime environment for JINI/J2EE proxies, and organizes JINI/J2EE proxies. The container can allocate and release resources for a JINI/J2EE proxy.
- The proxy manager is responsible for the proxy management. The manager is capable of communicating with J2EE applications running on a J2EE server; it manages proxy creation and destroying, according to the status of the corresponding J2EE application.

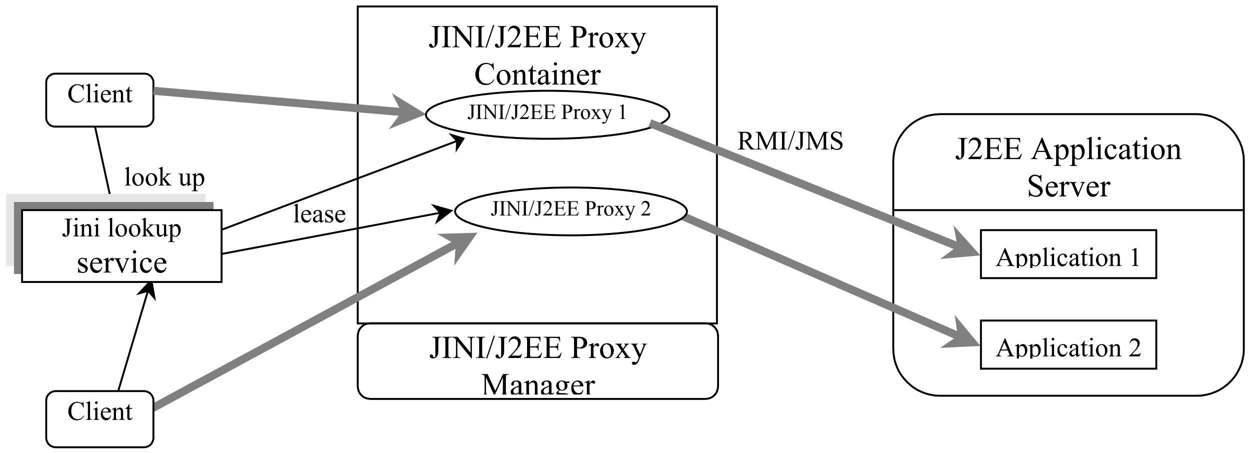


Figure 2. JINI/J2EE bridge architecture

3.2. Discovery mechanism

How can the bridge know there is a proxy needing to run? And how can an application be aware that there is a JINI/J2EE bridge available and send its proxy to the bridge? In order to let bridge and application discover each other, two types of messages are developed, query message and response message. As shown in Figure 3, the initiation of the discovery procedure starts from the bridge, since JINI/J2EE Bridge needs to be run all the time. The bridge sends a query message, asking "do you want to provide a Jini service?" When a query message is received by an application, if it wants to provide a Jini service, it will send a response message, saying "yes, my Jini proxy is there, please run it for me." In the response message, the URL of the proxy Jar file must be indicated. The Jar file contains the main class of representing proxy for the J2EE application and other supporting classes. Hence the manager can know where to download the proxy's Jar file.

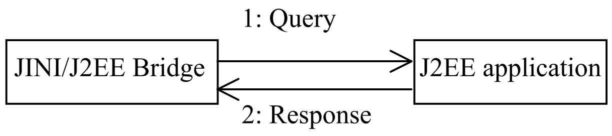


Figure 3. Discovery procedure

The proxy manager of the bridge is responsible to handle query message sending and response receiving, while the J2EE application is able to process query messages and send response messages by a message-driven bean called JINI/J2EE message-driven bean. In the deployment stage, this type of message-driven bean can be added along with other business logic beans into one .ear file.

3.3. Message models

In order to let the bridge talk to a J2EE application, query and response information needs to be wrapped into a J2EE message system such as JMS. We use different message models for processing query message and response message

For the query message, the publish/subscribe messaging model is used, because these query messages should be simultaneously passed to many applications, which want to provide Jini services. As shown in Figure 4, the JINI/J2EE manager sends a query message periodically to a certain topic, whose JNDI name is JINIServiceQueryTopic. After a J2EE application is deployed, its JINI/J2EE message bean subscribes to this topic and the query message can be received. In this way, any application going to offer a Jini service can be informed that the proxy bridge is active.

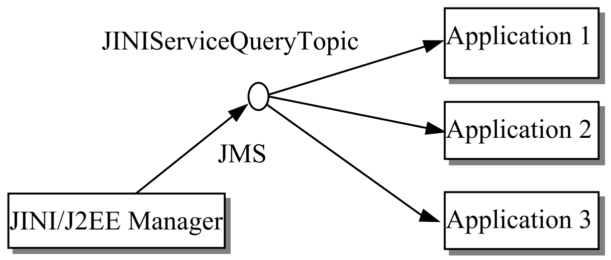


Figure 4. Query message model

For response message, given only one receiver, JINI/J2EE proxy manager, the point-to-point messaging model is considered. As shown in Figure 5, once an application receives a query message, it sends a response message to a specified queue named JINIServiceRequestQueue. The JINI/J2EE manager is a listener to that queue and hence the response message can be received.

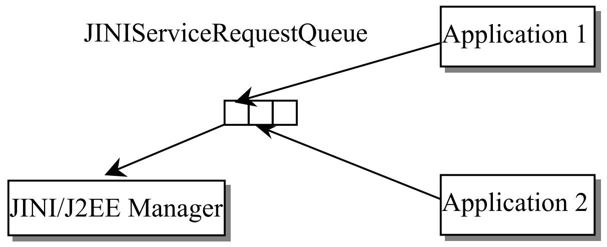


Figure 5. Request message model

3.4. JINI/J2EE proxy retrieval

We presume that the JINI/J2EE bridge does not know anything about a proxy before the proxy runs, otherwise, an administrator would be assigned to collect proxy class files from application providers and deploy them into the bridge. The bridge should download the Jar file of a proxy code automatically. Jar file retrieval for the bridge can be done with aid of an HTTP server. Every proxy Jar file is put into the HTTP server and its URL is indicated in response messages sent to the bridge. On receiving the response message, the bridge downloads the Jar file from HTTP server and activates the proxy, if a proxy is not running in the container.

3.5. Liveness

A JINI/J2EE proxy manager is responsible to maintain the lifetime of every proxy running in the container. The manager should keep in touch with the J2EE application; if the connection is lost, the manager must destroy the representing proxy and then ask the container to release all resources allocated for the proxy.

A time-based solution can be used to maintain the lifetime of activated proxies. Every proxy is set an expiry time when it is created in the container. When the proxy expires, the bridge will deactivate it. The manager publishes query messages periodically and these messages trigger JINI/J2EE message-driven beans to send a response. For a running proxy, its expiry time is extended on receiving its response message. So if an application disappears, there will be no response message from it for a long time, and then the proxy will be expired.

4. Implementation

We implemented a JINI/J2EE proxy manager and used surrogate host implementation from Madsion project[23] to simulate the container. The Madison surrogate host provides a Jini service running environment and export server from which Jini clients can download the Jini service resources.

The class diagram of the JINI/J2EE proxy manager is illustrated in Figure 6. Classes are explained below:

- JINIJ2EEProxyManager

It is responsible for proxy creation and interaction with the container.

- QueryTask

The purpose of QueryTask is to establish a connection to a certain topic on a J2EE application server and publish query messages to the topic on a J2EE application server.

- ResponseListenerThread

It is an asynchronous message listener, an object of RequestListener, to the queue on a J2EE application server, which is used to pass through Jini service response messages sent by a Jini/J2EE message-driven bean in the J2EE application. RequestListener realizes the interface *MessageListener* defined in JMS.

- IncomingResponse

This is used to parse response messages from a J2EE application and take action according to the content of the message and the status of its representing proxy.

- ProxyRec

This class holds the information of a proxy, which is running in the container.

- ProxyExpirationThread

This thread is used to maintain a proxy table in the manager. This proxy table records the information of all active proxies in the container.

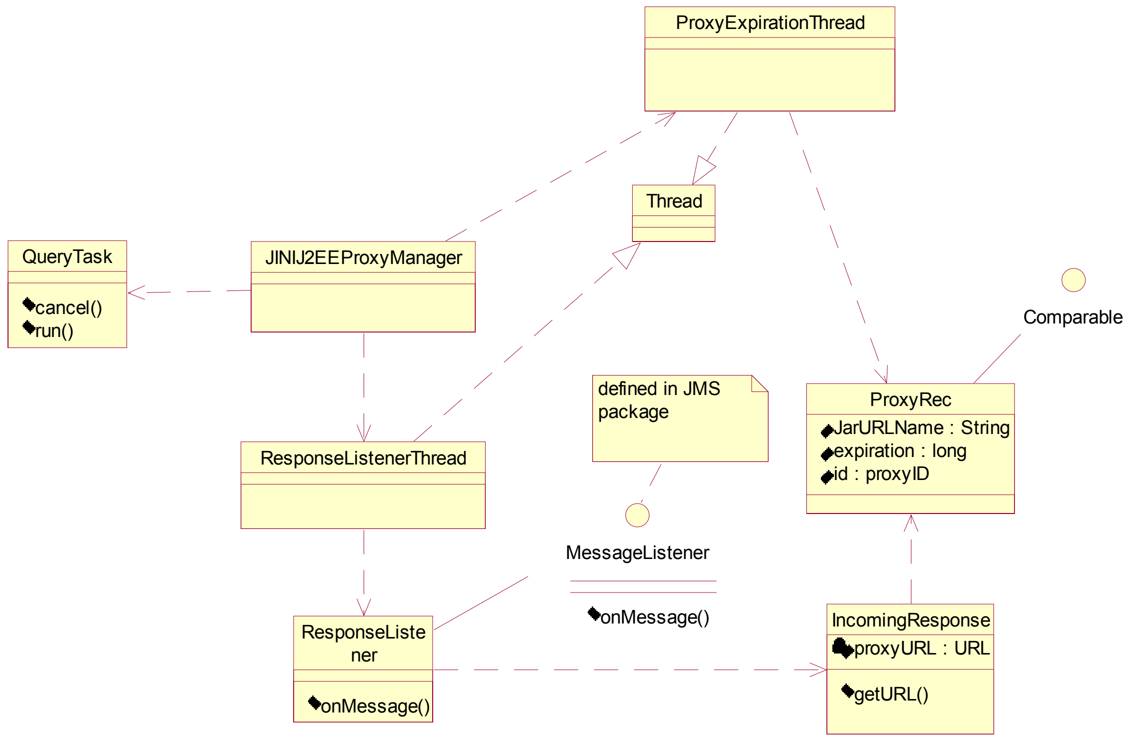


Figure 6. Class diagram of the JINI/J2EE proxy manager

5. Case Study

As introduced in the related work section, OSS software is a large-scale application for service providers. OSS/J is built on J2EE, and provides a well-defined interface in Java as shown in Figure 7. In this section, we present JIINI/J2EE proxy for OSS/J trouble ticket implementation [16] in the IP phone environment.

The class diagram is shown in Figure 8. A proxy can access trouble ticket application by invoking methods on the objects implementing OSS/J Trouble Ticket API. A suitable OSS/J Jini service interface (TTService in Figure 8) is defined, so that the proxy can be found by matching that service interface on Jini federation. Trouble ticket API is defined in TTProxy and Proxy is interface defined in the container through which the container can manage proxy.

After an instance of TTProxyImpl runs in the proxy host, it registers TTService to the look up service. An object of TTServiceImpl is exported to a Jini client as soon as the client asks for trouble ticket service and then client can create a trouble ticket by invoking createTroubleTicket( ). We create a program that can monitor IP phone status and generate a trouble ticket when the IP phone is not online. The trouble ticket reports the location and reason and the ticket is lodged into the database. The sequence diagram of trouble ticket creation is illustrated in Figure 9.

We deployed the Jar file of the proxy in a HTTP server. We also implemented a JINI/J2EE message-driven bean and deployed along with OSS/J trouble ticket implementation on a Weblogic server[20], J2EE application server. The message-driven bean is used to discover the JINI/J2EE Bridge, and send response message indicating the URL of the Jar file. When trouble ticket application is deployed, the bridge downloads the Jar file and activates trouble ticket proxy. After putting trouble ticket proxy access code into the phone's monitor program, the trouble ticket is successfully generated when the phone is switched off. In addition, the trouble ticket proxy can be also deactivated when the message-bean is undeployed from J2EE server.

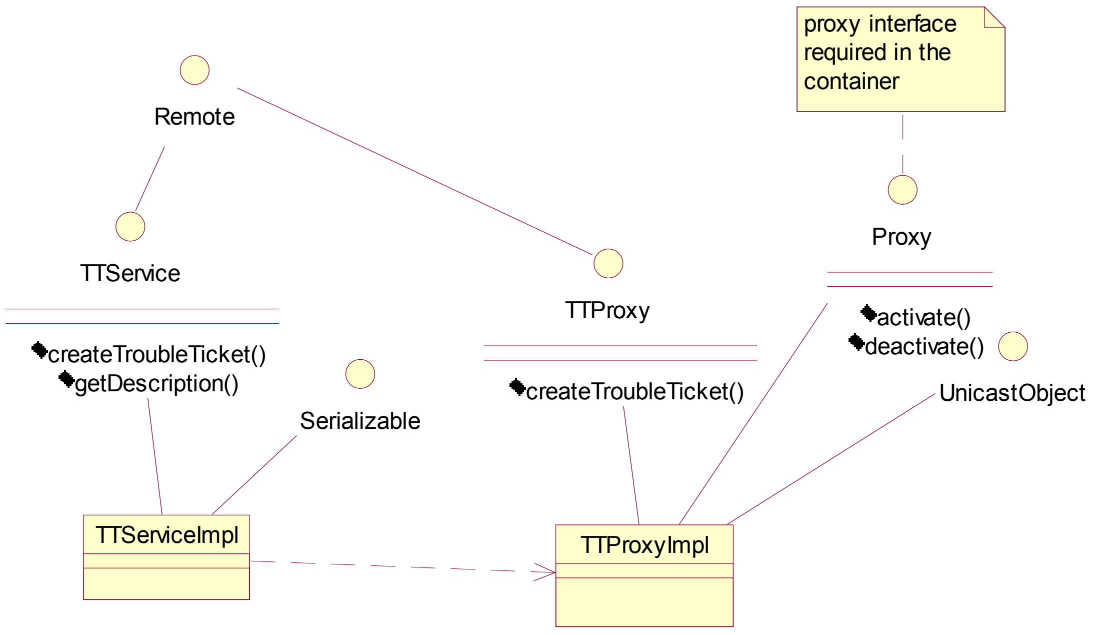


Figure 8. Class diagram of proxy for OSS/J Trouble Ticket system

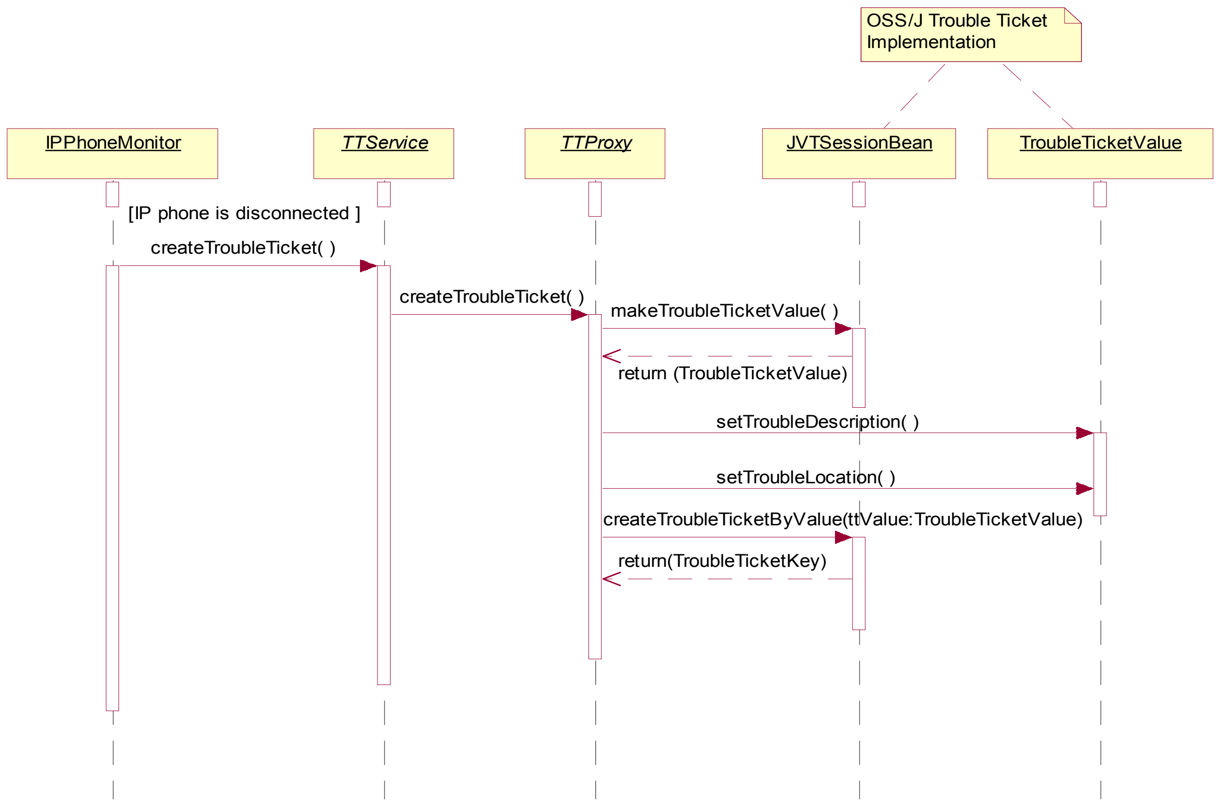


Figure 9. Sequence diagram of trouble ticket creation

6. Discussion

This JINI/J2EE bridge architecture, a container with a proxy manager, can be deployed with any J2EE-compliant application server such as Weblogic server. Developers can write a proxy program and deploy it into a HTTP server. As soon as the corresponding application is up on the J2EE server, the bridge retrieves and activates the proxy. The bridge enables J2EE applications to provide Jini access other than RMI and JMS type communication. In the case of Trouble Ticket API, on one hand, managers or customer care centers can create Trouble Ticket through Web or a J2EE client; on the other hand, network components, such as IP phone device and server, can generate trouble tickets automatically by Jini. Therefore, the developers of equipment can put trouble report for every possible error condition in their code without need for reference to the location of the trouble ticket system, as trouble ticket service can be discovered at run-time.

To date, the JINI/J2EE proxy manager of the bridge was implemented to work along with only one J2EE server. It is better to let the bridge support multi-J2EE servers. It can be achieved by making a property file for the bridge, in which the URLs of multi-J2EE servers are pre-configured, and then when the bridge starts, it reads all URLs from the property file and sends query messages to all URLs.

7. Conclusion

Centralized and closed system architecture hinders the development of telephony systems. Web-like phone architecture is expected to strengthen a lot of phone services provided by enterprise or individuals. We propose to use Jini as phone service middleware, due to the dynamic nature of phone environment. However, J2EE is a framework for global enterprise applications. Additionally, new standard APIs (OSS/J) for operation support systems and business support system are built on J2EE. It is clear that IP phone systems need to support both dynamics of administration and global access. In this paper, we presented architecture for marriage two technologies called JINI/J2EE Bridge.

The bridge can accommodate many J2EE application proxies on Jini federation. These proxies are automatically managed by the bridge according to the status of their represented J2EE applications, with the aid of a HTTP server. This general bridge architecture works along with a J2EE compliant server and provides Jini access for J2EE applications. A proxy for OSS/J trouble ticket, which is built on J2EE, was also discussed. By providing Jini access, an IP phone monitor can report trouble to the trouble ticket software without the need for reference to the service location. The success of the implementation of the JINI/J2EE proxy for the trouble ticket application certainly proves that this bridge architecture is practical.

8. Acknowledgment

We would like to thank Sun Microsystems, Melbourne for their support for this project. We also want to thank Rob Gray for his proof-reading for the paper.

9. References

[1] Pintel Corp., Next Generation VoIP Services and Applications Using SIP and Java, Technology Guide, http://www.pingtel.com/docs/collateral_techguide_final.pdf.
[2] Jini Network Technology, http://www.sun.com/ software/ jini
[3] Java 2 Platform, Enterprise Edition, http://java.sun.com/ j2ee
[4] Schulzrinne H., Casner S., Frederick R., and Jacobson V., *RTP:* A Transport Protocol for Real-Time Applications, RFC 1889, January 1996
[5] Arango M., Dugan A., Huitema C., and Pickett S., Media Gateway Control Protocol(MGCP) Version 1.0, RFC 2705, The Internet Society, October 1999
[6] Cuervo F., Greene N., Rayhan A., Huitema C., Rosen B., and Segers J., Megaco Protocol Version 1.0, RFC 3015, November 2000
[7] ITU-T, H.323: Packet-based Multimedia Communications Systems, International Telecommunication Union, 1998
[8] Rosenberg J., Schulzrinne H., Camarillo G., Johnston A., Peterson J., Sparks R., Handley M. and Schooler E., SIP: Session Initiation Protocol, RFC 2543, The Internet Society, February 21, 2002
[9] Pingtel Corp., http://www.pingtel.com
[10] Java Telephony API, http://java.sun.com/products/jtapi/
[11] Java Speech API, http://java.sun.com/products/java-media/speech/
[12] Java Media Framework, http://java.sun.com/products/java-media/jmf/
[13] Sun Microsystems, The JAIN APIs: Integrated Network APIs for the Java Platform, White Paper, May, 2002, http://java.sun.com/products/jain/WP2002.pdf
[14] OSS through Java Initiative, http://java.sun.com/products /oss
[15] OSS through Java J2EE Design Guidelines, http://java.sun.com/products/oss/Com-arch-dg.1.1.pdf.zip
[16] OSS Trouble Ticket API, http://www.jcp.org/aboutJava/ communityProcess/first/jsro91/
[17] Jini Technology Surrogate Architecture Specification, http://surrogate.jini.org/sa.pdf
[18] JMatos Software, http://www.psinaptic.com/oem/

[19] JiniME: Jini Connection Technology for Mobile Devices, WhitePaper, http://www.cs.rit.edu/~anhinga/Whitepapers/JiniME
[20] BEA WebLogic Server Overview, White Paper, http://www.beasys.com/products/weblogic/server/wls-70-ov-wp-04021.pdf
[21] Java Message Service API, http://java.sun.com/ products/jms/
[22] Remote Method Invocation, http://java.sun.com/products/ jdk/rmi/
[23] Madision project, contributed implementation for Surro-gate host, http://ipsurrogate.jini.org/specs.html