Jini Lookup Service for Resource-constrained Mobile Devices

Chang-hoe Kim , Myounghwan Oh, Hoon Choi
Dept. of Computer Engineering, Chungnam National University
220 Kung Dong, Daejeon, 305-764, Korea
E-mail : {kchoe, mhoh, hchoi}@ce.cnu.ac.kr

Abstract

The lookup service of Jini technology helps clients find and connect to network services and information. The code size of the service is big for mobile devices which usually have limited memory resource. This study proposes a smaller design and implementation of the lookup service so that mobile devices can run it for its own Jini federation in an ad-hoc environment.

1. Introduction

Advances of mobile communication technologies and device technologies brought the mobile computing era where people use mobile devices such as a PDA(Personal Data Assistant), a cellular phone and a handheld PC(Personal Computer) for their personal or business work.

Such devices may run alone, or they may cooperate with other devices and computers to share computing resources. For instance, a salesperson may use a network printer in his office to make a hard copy of the sales record kept in his PDA. His PDA must be equipped with an appropriate printer driver for this purpose. If the salesperson visits a customer and needs to make a printout on the customer's printer, he has to install a different driver software specific for the customer's printer along with appropriate system re-configuration. This is tedious and difficult job for the most of ordinary users.

To ease the difficulty, Sun Microsystems introduced the Jini technology in 1999 that supports distribution of software-defined services and information in network environments. Jini allows configuration of devices and software being amended using a simple "network plug and work" model like the Universal Plug and Play(UPnP). The lookup service of Jini technology helps clients find and connect to network services, therefore it is the integral of Jini.

This Research was supported by SOREC(Software Research Center) of KOSEF in Chungnam National University. SOREC is a Regional Research Center Designated by Korea Science and Engineering Foundation(KOSEF).

The Jini lookup service was designed to run on a desktop level computer. However there are circumstances in which a mobile device needs to act as a lookup server. For example, a connection to the network is not available in the wireless environment thus mobile devices are isolated from the wired network, or mobile devices may want to communicate directly in a peer-to-peer method using an infrared or a bluetooth interface within a closed work group or in an ad-hoc network. Code size of the Sun's implementation of the lookup service is too big to be loaded in mobile devices for this purpose.

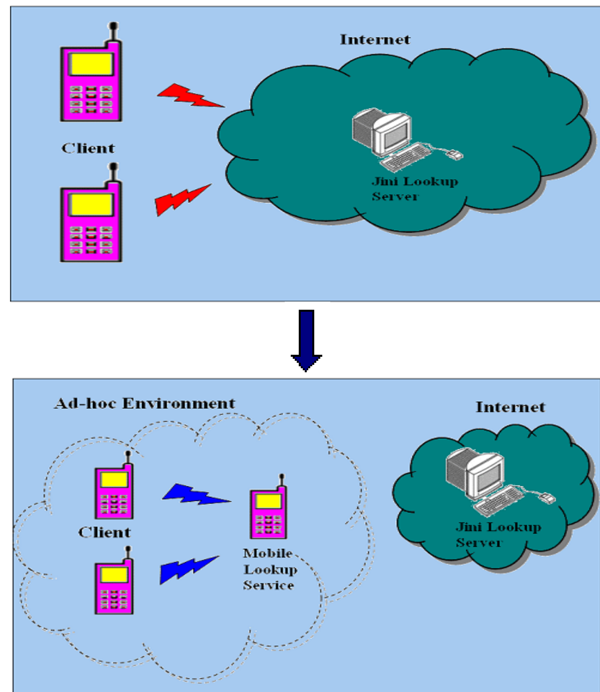

Figure 1. Use of the Mobile Lookup Service

This study proposes a smaller implementation of the Jini lookup service in order to load it into a mobile device. The basic idea is to divide the original, monolithic lookup service into multiple modules, and have a main module

dynamically load other modules on-demand. This mechanism provides complete Jini capability and enables resource-constrained devices to participate fully and transparently in Jini federation. Therefore it is possible for small devices such as mobile devices and embedded devices to host the lookup service.

On-demand downloading of lookup service modules surely slows down the lookup service. However, as long as a mobile device contains only the modules necessary for its own service, downloading of other modules is not required or takes place infrequently. As. The overhead may be tolerable considering the benefit of the mechanism, i.e., scalability of the Jini lookup service and the consequent enabling of mobile devices to participate in a wider range of mobile applications.

After a brief introduction of the Jini architecture and mechanism of the Jini lookup service in Section 2, we describe the design and a prototype implementation of lookup service for mobile device in Section 3.

2. Jini Architecture and Lookup Service

Jini technology is a simple infrastructure for providing services in a network and for creating spontaneous interactions between programs that use these services.

Devices build a Jini federation using discovery and join protocols by the following steps.

①When a service is booted on the network, it uses a discovery process to find the local lookup services. The service then registers its proxy object with each lookup service.

② A client program asks for services by the Java language type the client will use. The lookup service returns the proxy object.

③The client issues requests to service by invoking methods on the proxy object. Proxies provide all of the code needed to connect to a particular service. The Jini system does not define the protocol that the proxy and its service use to communicate with.

Proxies are similar to device drivers in that they allow an application program to interact with a service while shielding it from the details of that service. But Jini proxies are installed by the services themselves and are dynamically downloaded when clients wish to use a service. Clients need not know the implementation details of these proxies.

Services can join or leave the network in a robust fashion and clients can rely upon the availability of visible services. Jini is self-healing because if the network is recovered when a network failure isolated a service from a lookup service, the service will receive a message from the lookup service and rejoin.

The Java virtual machine provides that objects can be moved around the network in a consistent and trustable manner. These properties enable a system built on dynamic service proxies, moving object state and implementation to the most useful parts of a system when they are needed. Each Jini system is built around one or more lookup services.

The Jini lookup service is a directory service and it has information for all services in a Jini federation. The Jini lookup service interprets Java types, so it can retrieve a proxy that implements a particular Java interface and find a super class or super interface of the proxy. The internal implementation of the lookup service is hidden for user. Users only know the lookup service's proxy that implements ServiceRegistrar interface.

The ServiceRegistrar interface is used to implement all the lookup services. Users do not need to know how it communicates between proxy and a lookup service in detail. The Java RMI(Remote Method Invocation), socket or message passing are usually used for the communication between them.

The main role of the lookup service is the Jini registrar. It receives requests for registration from services. Each registration is made by a service item which consists of proxy object of the service, service identifier and a set of attributes associated with the proxy.

Other functions of lookup services can be listed as:

- starting a Jini federation
- managing databases which permanently store service items, lease, registration
- supporting self-healing network
- generating a unique service identifier
- merging multiple Jini federations by importing or exporting a lookup service

Functions of a lookup service must implement the ServiceRegistrar interface by RMI proxy or by using a similar mechanism. When clients or service providers access a lookup service, they download ServiceRegistrar interface object and then invoke a method of the object.

The code of the lookup service implemented and distributed by Sun Microsystems is made for stationary computers, and all functions of the lookup service have been implemented into one object. It is too big for small, limited mobile devices to run.

3. Jini Lookup Service in Mobile Devices

It is not necessary that all the functions are ready always. Some functions may not be used at all, and some lookup services may be sufficient to provide only a subset of the lookup service depending on the usage of the particular application. For example, we may want a

lookup service that registers a set of specific service items and it need not provide event or log management functions. In this case, carrying all the functions is a waste of memory.

In order to flexibly compose various scale of lookup service capabilities, we propose an approach that breaks down the lookup service functions into multiple, downloadable modules and then dynamically loads, removes modules on-demand.

(1) Core Module: It implements three protocols that comprise the discovery process, i.e., the unicast discovery protocol, the multicast request protocol and the multicast announcement protocol. This module monitors packets of the multicast request protocol from new startup clients/services for the groups that it handles. This module also implements the registrar proxy. At the end of a successful discovery, the requesting client/service holds one or more lookup service registrar proxies.

(2) Lookup Module: This module looks after and persists the service items of the registered services. And it supports a template search based on a combination of three criteria: service identifier, the type that the service supports, and associated attributes.

(3) Service Management Module: This module manages services by storing and removing service items.

(4) ID Generator Module: This module creates the service ID.

(5) Event Module: This module manages event solicitation by storing a reference to a listener object from the client.

(6) Lease Module: If the lease under negotiation is granted by the lookup service, then a listener object is put into storage.

(7) Log Module: It supports persistent logging in a recoverable manner.

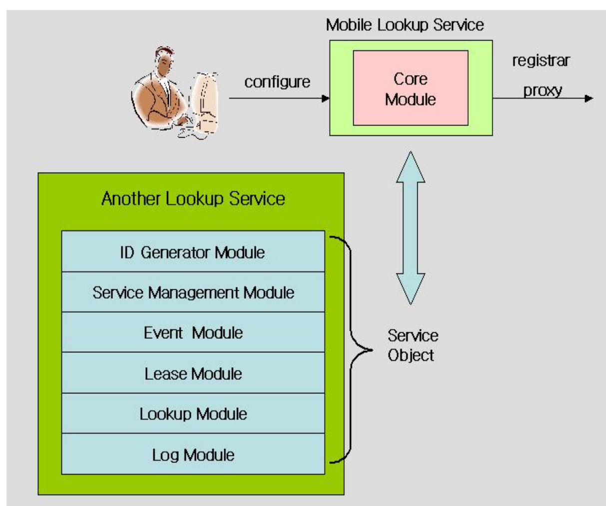

Figure 2. The Lookup Service Modules

The core module resides in a device at all times. Other modules can be optionally downloaded to the mobile device and dynamically re-composed with the core module. Optional modules are required to be registered as the service items in other lookup servers, possibly in a backend, monolithic lookup server in a wired network. When the lookup service of a mobile device receives from its client a request for which the lookup service needs modules that the device does not carry, the lookup service can either reject the client's request or it can connect to a backend lookup service to download the required modules. Therefore, the mobile device can be a Jini client as well as a server.

The modules of the lookup service shown in Fig. 2 are not the remote proxy to communicate by RMI, but they are the local proxy that contains service objects themselves. The size of modules gets bigger than remote proxy, but the services need not communicate with the backend server anymore, once downloaded.

Before the service is operational, the core module and some optional modules are loaded into a mobile device by off-line operation to compose a mobile lookup service. Deciding which of optional modules to load depends on the scale of lookup service functionality that the mobile device likes to provide or depends on the hardware capacity of the mobile device. If hardware permits, the

```
public class RegistrarImpl implements Registrar {

        RegistrarImpl() throws IOException {

        }

        cnuUnicaster = new UnicastThread();                     \
        cnuLocator = new LookupLocator();                        } Supporting
        cnuMulticaster = new MulticastThread();                  } client's requests.
        cnuAnnouncer = new AnnounceThread();                    /
        proxy = new RegistrarProxy();                           \
        cnudiscovery = new LookupDiscoveryManager();             } Exporting
        cnujoiner = new JoinManager(proxy,...);                 / Registrar Proxy

        if( moudule 1)                                          \
                registrar = lookup( service module1 );           } Finding Service
        if( moudule 2)                                           } Modules
                registrar = lookup( service module2 );           }
        if( moudule 3)                                           }
                registrar = lookup( service module3 );          /
                .
                .

        select Func(){    } Selecting
        }                 } Modules
}
```

Figure 3. The Skeleton of the Dynamic Module Loading Mechanism

mobile device can carry all of the optional modules, then this lookup service becomes a full-scaled lookup service. As long as the mobile lookup service carries the functions that clients need, it looks as if it is a full-scaled lookup service at client's side.

After testing our prototype implementation, we confirmed the ability of the mobile lookup service to dynamically integrate with the optional modules on-demand while providing the full service of the Jini specifications. Table 1 shows the sizes of compiled class and run-time memory of the modules and the class size of the monolithic lookup service measured from the Sun's reference implementation. The size of the monolithic lookup service is 220 Kbytes whereas the core module's is 143 Kbytes.

The original monolithic lookup service also requires much bigger space than the mobile lookup service with respect to the run-time memory. For example, when we load the core module, the size of required memory is only 1746 Kbytes, whereas the monolithic lookup service requires at least 3694 Kbytes. Even if we add modules, the mobile lookup service use smaller memory. For example, if we need the lease management function additionally, then we configure the mobile lookup service with two modules. The class size of this case is only 60.27 Kbytes and the size of memory to run it is 1806 Kbytes which is much smaller than the monolithic lookup service's requirement.

The optional modules that may not be used anymore can be removed immediately from the device, if the user chooses. This will further save the memory space, resulting in the ability to run our implementation on smaller and more various types of devices in a various service scale.

We tested our implementation on the J2SE(Java2 Standard Edition) environment. If our implementation is tested on the J2ME(Java2 Micro Edition), we expect the run-time memory requirement to become even smaller.

Table 1. Class Size and Memory Requirement (Kbytes)

Module	Class Size	Memory Requirement
Core Module	143	1746
Event Module	10.9	60
Log Module	31.1	120
Lease Module	8.77	18
Lookup Module	257	110
Service Management Module	15	30
ID Generator Module	2	45
All of the Modules	233.1	2102
Reference Implementation	220	3694

As we mentioned earlier, this mobile lookup service is designed for a small device and for a case that the lookup service does not have to provide full functionality of the specification. Then the core module and possibly one or two additional modules are quite enough for a lookup service and saves memory resource.

4. Conclusion

The mechanism described in this paper makes Jini lookup service be able to run on a small sized mobile or embedded device while it is fully compatible with the Sun's reference implementation. Inevitable cost is to search and download one or more additional modules from a backend lookup service in case that the requested function is not already loaded in the mobile device. It will result in some communication cost and delay. But if it does not happen very frequently, the overhead may be acceptable.

Our mobile lookup service is beneficial when the full-scaled lookup service is not available or when a mobile device has to provide a lookup service to other devices that do not have capability to directly access the full-scaled, backend lookup server in the Internet. By providing a way for small devices to participate in and to form Jini federations, our mechanism can contribute the Jini technology to be more useful in mobile computing era.

It is possible to make finer grain modules so that wider range of different Jini functions can be loaded into a memory at the same time. Then we can reduce the chance of downloading necessary functions from the backend lookup server, thus obtain better performance. Users can compose a customized lookup service based on his device capability and his need of the lookup service.

5. References

[1] Ken Arnold, Bill Joy, The Jini Specifications(2nd Ed.), Addison-Wesley, 2001.

[2] Sing Li, Professional Jini, Wrox, 2000.

[3] W. Keith Edwards, Tom Rodden, Jini Example by Example, Prentice Hall, 2001.

[4] Jan Newmarch, A Programmer's Guide to Jini Technology, Apress, 2000.

[5] W. K. Edwards, Core Jini,(2nd Ed), Prentice-Hall, 2001.

[6] Sun Microsystems Inc., The Remote Method Invocation Specification, 2001.

[7] www.sun.com/jini/specs

[8] www.upnp.org/resources.htm/