Design a Jini-based Service Broker for Dynamic Service Combination Framework

Hsu, Kuo-Wei
kuowei.hsu@gmail.com

***Abstract*—The increased use of electronic service has invented a new term Service-Oriented Architecture. In a typical distributed environment consisting of many independent devices and services, it is practically required to apply a framework of dynamically combining service. The goal in developing such a framework is to add variety of functional operations as well as interaction among services, share data of software modules, and force devices and services working together in a coordinated fashion. This paper proposes a dynamic service combination framework coupled with a Service Broker for Jini, and explains how to achieve the behavior coordination of services. The major contributions of this paper are: first, the concept of combined service is abstracted into a number of cooperative services; second, it provides a programming framework that allows us not only to combine services dynamically but also to generate the combined service automatically.**

***Index Terms*—Service-Oriented Architecture (SOA), Jini, dynamic service combination framework, service broker**

I. Introduction

Service-Oriented Architecture (SOA) has gained much attention in recent years with the coming of technologies such as Jini and Web Service, which both include three elementary roles, service provider, service consumer, and service naming directory [1-8]. A typical SOA is shown in Fig. 1.

In SOA, the interface of the service is defined in a fixed way, but the implementation of the service can be changed according to future requirements. The proposed Service Broker performs a dynamic service combination framework and makes it possible to maintain environments in a spontaneous and incremental way by adopting standard protocols as well as a simple combination process. It allows developers easily to incorporate hardware devices and software modules into a system. In promoting the home networking feature, with the framework proposed in this paper, it is possible for developers logically to decompose a general information appliance into specialized functional units for reducing the development complexities.

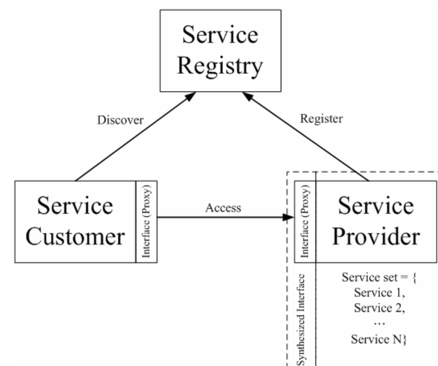

Figure 1. Service-Oriented Architecture.

This paper proposes a Jini-based Service Broker for achieving service combination. A service in Jini can be realized to represent a physical device (hardware), a program (software) or a combination of both [18-19]. Also, this paper extends the meaning of a Jini service to a set of services. In the scenario of this paper, service combination is coordinated by a special service that manages the discovery, combination and execution of the combined service. The key features of the proposed Service Broker are as follow. First, one component is treated as a service available on a network. Second, the concept of combined service is abstracted into several cooperative services. Third, it provides a programming framework that allows for the dynamic combination of services.

The organization of this paper is as follows. The second section of this paper will supply background information on Service-Oriented Architecture and a brief introduction to Jini Technology. Following that, the architectural overview of the proposed dynamic service combination framework and the Jini-based Service Broker will be discussed in the third section. Later, the fourth section will present more advanced design issues, while the fifth section will demonstrate a practical example. Finally, a conclusion will be given in section six.

II. Technical Backgrounds

A. Service-Oriented Architecture

E-service can be defined in general as an aggregation of electronic devices. Examples are Sun Microsystems' Jini [18] and Web Service [5, 7, 8]. The increased use of

© 2006 ACADEMY PUBLISHER

E-service has opened a new horizon of proliferation in software engineering and has invented a new term Service-Oriented Architecture. Here, a service is a functional module that is self-described, self-managed, and independent. Further, a Service-Oriented Architecture (SOA) is a design philosophy that is based on a collection of services, each of which communicates with others through specific protocols. In other words, SOA is a software topology that allows registered services and consumers to be in a loosely coupled relationship.

SOA is not a whole new concept. The prior technologies for SOA include CORBA [24, 25] and DCOM [26, 27]. Unfortunately, neither CORBA nor DCOM provides an efficient solution to deal with the rapidly changing networks. Contrarily, both Jini and Web Services technology give developers a better chance to handle the changeable networks and to approach the ideal plug-and-play systems. However, there is another we have to note here: Web Services technology and SOA are not synonymous. More precisely, SOA is a design principle while Web Services technology is one of the implementation technologies realizing SOA, and so does Jini.

The most important aspect of SOA is that it divides a service into two parts: "what" and "how" – the former means the interface and the latter means the implementation. A service consumer does not care about how the service works for satisfying its request. Besides service provider and consumer, a service registry plays a pivotal role in a typical SOA environment, as shown in Fig. 1. A service registry acts like a directory maintaining access information of all registered services.

Before analyzing the detail of SOA, it is important to first explore software architecture, which describes components of a system and the way they interact with one another. These components are abstract software modules deployed as a functional module on servers, and further services are self-describing components offered by service providers. They are responsible for obtaining and maintaining the implementations of services and providing descriptions for services. Service descriptions provide the basis for the binding, discovery and combination of services.

SOA promotes loose coupling between service consumers and service providers [5]. Coupling refers to the number of dependencies among modules. Loosely coupled modules have a few well-known dependencies. Contrarily, tightly coupled modules have many unknown dependencies. If a service consumer knows the details of a service provider, they are more tightly coupled. On the other hand, if the consumer does not need the detailed knowledge of the service before using it, they are more loosely coupled. The consumer does not depend directly on the implementation of the service but only on the contract (interface) the service supports. Although the coupling between service consumers and service producers is loose, the implementations of services can be tightly coupled with implementations of other services, for example, a set of services is tightly coupled if they have access information about implementation.

The main idea behind SOA is to take an appropriate modularity so developers can build a system in which components are not tightly coupled with others. In the programming methodology of SOA, a service consumer is not tied to a particular service provider, instead, the service providers are interchangeable [3, 6]. One characteristic of SOA programming is that services are dynamic in nature and they can be registered to and unregistered from the service naming directory at any moment and service consumers have to prepare to cope with this situation. The second one is that a service consumer must be prepared to cope with situations where no services are found or multiple matching services are found. Finally, the third characteristic is that service consumers do not know whether they are interacting with a common service implementation or other particular implementations.

B. Jini Technology

Jini is designed to turn a network into an easily managed environment on which consumers can find services in a flexible and robust fashion. Jini is unique in that it introduces the idea of transmitting a proxy object (remote reference) to the consumer where it is used as an agent to the corresponding service entity. The service simply has to encode its well-known location within its proxy. Any client (service consumer) can call methods on the proxy of interested service entity without having any knowledge of the actual location of that service. Fig. 2 illustrates operations in a Jini federation.

A collection of Jini services forms a Jini federation where services coordinate with others. One main module is the Jini Lookup Locator (JLL in Fig. 2) providing the Lookup Service (LS in Fig. 2), which maintains dynamic information about all available services in a Jini federation. More precisely, the JLL maps interfaces of services indicating the functionalities provided by services to several sets of objects that implement services. In addition, descriptive attributes associated with a service allow more accurate selection of services according to attributes. Every service has to discover one or more JLL before it can be allowed to enter a federation. The location of the JLL could be built-in or discovered using multicast at run time (line 1 and 3 in Fig. 2). When a Jini service wants to join a federation, its provider discovers at least one JLL first and then uploads the service implementation code, i.e., a set of Java classes, to the JLL (line 2 in Fig. 2).

© 2006 ACADEMY PUBLISHER

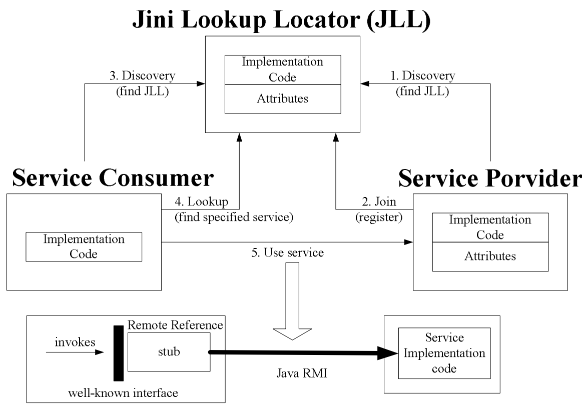

Figure 2. Operations in a Jini federation.

In a Jini federation, a service consumer uses a service proxy to contact some service entity. A consumer attempting to search for a service first launches a multicast-based query to find out the JLL in the network. If a JLL exists, the corresponding JLS service proxy is downloaded into the consumer's JVM (Java Virtual Machine). The consumer then uses this proxy to run the lookup procedure and find out wanted services.

Jini adopts Java RMI (Remote Method Invocation) to implement such a service proxy. Java RMI is based on remote reference, which is the local representative of the remote object involved in the method invocation [20-23]. Rather than making a copy of the implementation object in the receiving JVM, Java RMI passes a remote reference for a remote object. The client-side remote references are "stubs", while those in the server are "skeletons".

A representative Java RMI architecture is represented in Fig. 3. Fundamentally, the stub or skeleton layer includes the client-side stub and the server-side skeleton. The remote reference layer handles the behavior of remote reference, such as invoking an object. The transport layer initializes and manages connection, the remote reference, and the remote object tracking. A stub implements the same interface as the remote skeleton it relates to. Further, it forwards method invocations received from clients to the appropriate skeleton provided by the service provider. A skeleton waits for a remote method invocation, and sends it to the corresponding objects. Moreover, the service consumers can use the service stub to contact the Jini service and invoke methods on that service.

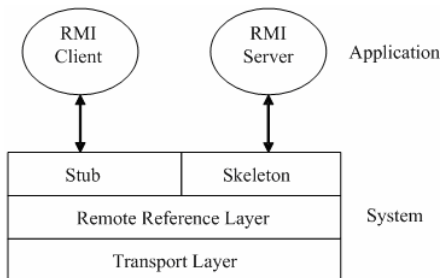

Figure 3. RMI architectural overview.

Because service consumers only interact with the service stubs based on Java, the nature described previously allows various types of services to be accessed in an uniform fashion. For searching for a service, a consumer first launches a multicast-based query to find out the JLL in the network (line 3 in Fig. 2). If a JLL exists, the corresponding LS service stub is downloaded into the consumer's machine. The consumer then uses this stub to run the lookup procedure and find out the wanted service (line 4 in Fig. 2). The service consumer makes a request for some services by specifying a Java type, specific attributes, or the interface that wanted services have to implement. Besides, in Jini, service discovery is done by Java interface or attribute matching. Once a candidate service is found, the stub registered by the service provider is copied to the consumer's machine using Java RMI (line 5 in Fig. 2). Consumers take the use of the stub to interact with the service.

III. Architecture Overview

A. Dynamic Service Combination Framework

Service combination is an important and active area of research, and has been studied widely for wired as well as wired and wireless networked services. Many researchers are working on distributed computing with SOA. Their outstanding achievements show that SOA is a practical approach for combining components. Using SOA, distributed systems can share various components and processes. Developers can utilize SOA to build distributed systems more easily.

Jini's dynamic nature enables services to be added or withdrawn from a federation, a group of services. The proposed process of combining component consists of the following steps:

1. Specify possible combination rules, at design time.
2. Specify components as services, at design time.
3. Make them participate into a Jini federation, at run time.
4. Generate the combined service when appropriate ingredient services available on the Jini federation, at run time.
5. Mount the newly generated virtual service in the Jini federation, at run time.
6. Allow the combined service to be discovered and accessed by other Jini-enabled applications or even services, at run time.

In this scenario, service combination is coordinated by a special service that manages the discovery, combination and execution of the combined services.

Here are two significant features. First, the concept of a service is extended from an entity to a set of entities – a combined service is actually a set of cooperative services or components. To clarify, the proposed framework enlarges the service provider from the solid-line box to the dash-line box in Fig. 1 such that various services are bundled as a service seamlessly and transparently. Therefore, from the service customer's view, a combined service acts just like an ordinary service. On the other hand, from the viewpoint of a service registry, a

© 2006 ACADEMY PUBLISHER

combined service is not different from a normal service. The second feature is that, it provides programming interface that allows for dynamically combining services at run time.

From a device developer's or software architect's point of view, the combination of components or services in a complex distributed environment will face many design issues and development problems. Examples are protocols compatibility, absence of the API of the required service on time of compilation. Also, there will be problems of configuration and dynamic deployment, a set of problems related to the absence of certain information about services or components in a distributed environment at run time, and many others.

The reason we adopt Jini instead of Web Services technology here is that, a web is a program that hides behind a web server. It is activated by a web server when a request is coming. So, it is safe to say that Web Services technology is stateless, which means when a request comes, a service wakes up, takes the responses accordingly, and finally returns to the waiting status. Because of such a characteristic, using Web Services technology to combine services dynamically and then generate the combined service is harder or more expensive than using Jini to do those. Jini views each service as an object so we can generate a combined service as we create a new object and further we can monitor each service by its state.

Another reason we do not employ Web Services technology but Jini is that, Jini has built-in security mechanism since it was born. Further, from version 2.0, Jini supports a more comprehensive security mechanism. To explore this more, a security manager is any class whose ancestor is the *jaba.lang.SecurityManager* class. A customized security manager allows developers to build a user-defined security policy for a specified service. Once the security manager is activated, every access will be checked with the security policy, and every invocation will be allowed only after the customer gets the permission from the service provider. In addition, a Jini service could specify which customer is allowed to invoke which method it provided. The security policy is used to grant "stub" (dynamically downloaded code) permission to do something, or not to do anything, which means it is the access control, and a solution to the authorization problem.

B. Service Broker

The spurt of SOA has increased the importance of service combination. Service combination enables the users to integrate existing services to satisfy complex requests that require cooperation of multiple services. One direction of SOA research aims in developing architectures that enable service composition [8, 14, 15]. These architectures presume a workflow specification of a composite service and perform the task of discovering, integrating and executing the services.

Service combination refers to the technique of creating complex services with the help of smaller, simpler and easily executable services or components. In a home network, for example, a Bluetooth-enabled Digital Video Recorder (DVR) carrying video files could be a service providing AV (audio/video) data in various formats. The notion of integration has been proposed to describe such dynamic combinations of devices and services. It embodies both software and physical concepts of composition and de-composition. In particular, it supports combination of heterogeneous services and highly dynamic shifts in which services are available; properties which are characteristic of ubiquitous computing. So, a service combination is of the use-oriented (service-on-demand) concept: a combination of services or devices is considered as a conceptual whole by its user. In a distributed system supporting plug-and-play connectivity, first, the centralized design approach of the combination engine in a wired network is prone to single point of failure, especially since all nodes are not so reliable due to their inherent nature. Second, service topology (distribution of services on various ad hoc nodes) changes because of dynamically entering and exiting the network. The service combination architecture should be able to use the spatial distribution of services to optimize service combination and execution.

This paper introduces a process of brokering to combine services. In this paper's scenario, a union service is a combined service consisting of various component services, and further, a component service provides specific functions and is an ingredient of a union service. For brokering services, initially the broker finds all component services and then employs them to create a union service according to the corresponding combination rule defined in a database or a knowledge base. In this scenario, it suggests the automatically generated union service is a virtual service in reality. A service consumer directly accesses a union service and is not aware that underlying component services cooperate to get the work done.

The proposed architecture mainly consists of a Broker Engine as well as a Knowledge Base and is presented in Fig. 4. The Broker Engine is responsible for combining component services and automatically generating a union service with the help of the Knowledge Base that keeps track of rules of combination processes.

In Fig. 4, the *usManager* class manages the union services. The *BorkerService* interface provides a proper definition of the brokering service, while the *BrokerAgent* class is the core element that performs the combination process. Further, an instance of *usThread* class corresponds to a union service, which is a virtual but workable Jini service. The *ServiceFinder* class is a helper class used to find the specified component services.

The *BrokerService* interface defines two methods, *addUnionService* and *delUnionService*. The former is used to add a new union service and the latter is used to delete an existing union service. To realize the *BrokerService* and make it a standard Jini service, an implementation class named *BrokerServiceImpl* is provided. Surely the stub of an instance of *BrokerServiceImpl* can be downloaded by consumers attempting to use the brokering service.

© 2006 ACADEMY PUBLISHER

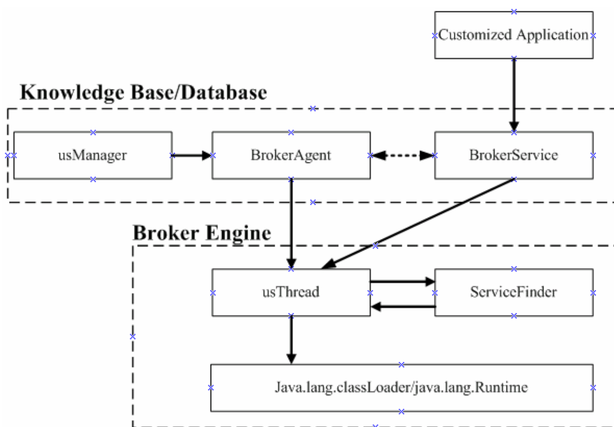

Figure 4. Proposed Jini-based Service Broker

The *BrokerAgent* class manages several data structures to keep track all the union services. Also, it exports the current status of the brokering service into a file, and importing the previous status from a file as well. Such a capability can encourage the automatic initialization through default setup file.

The *UnionService* class is a super class of all union service implementations. To make this virtual service look like a real Jini service, an instance of *UnionService* has to properly set the value of related attributes. Further, each instance of *usThread* class presents a thread running a union*Service* instance. The *usThread* class is not restricted to any particular union service. Operations taken in the *run* method of the *usThread* class are straightforward. In the first step, it finds all component services in a rolling manner. In the second step, it creates and then initializes the corresponding *UnionService* object with the help of Java reflection API and *java.lang.ClassLoader* that loads specified Java classes dynamically.

IV. Advanced Issues

A. Leasing Mechanism

In distributed systems, the most common failure scenario is one in which some system modules cannot be accessed. Therefore, Jini introduced distributed leasing to help developers address the resource management issues that partial failures present. To ease the development of applications built on the proposed architecture, we make good use of the Jini leasing mechanism. For the proposed architecture, the leasing model is demonstrated in Fig. 5.

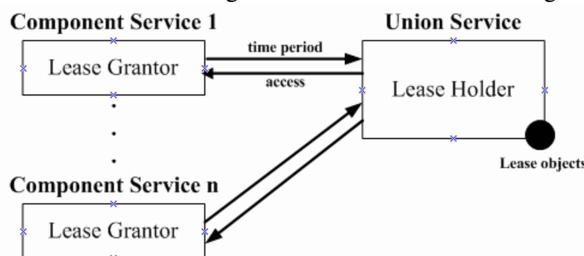

Figure 5. Leasing mechanism

The leasing mechanism enables distributed components to explicitly limit the duration of their agreed cooperation. This removes any possible ambiguity about when such agreements are terminated and allows modules to safely reclaim resources that had been associated with them. In the Jini network, a lease is used to establish an agreement about how long the JLL will maintain a record in the naming directory that is for the corresponding service. Since the resources belong to it, the JLL grants the lease to the developer's service. Besides, the duration of each agreement can be specified independently. The responsibility for initiating a lease renewal request belongs to the original lease requestor. If a service would like the JLL to continue registering its availability, it must request the JLL renew its service registration lease before its expiration. A lease requestor can also cancel a lease. If a lease is neither renewed nor canceled before its specified duration time passes, the lease automatically expires. Here, the lease grantor and lease requestor are both freed from responsibilities associated with the lease.

In the proposed architecture, all implementation codes of component services should implement the *ComponentServiceInterface* interface. By combining with leasing, a component service is able to control its resource usage or public access actively. On the other hand, a union service should perform a resource allocation mechanism in order to manage the situation, such as when the union is no longer maintained due to the disappearance or unavailability of its dependent component services. The leasing policy should be implemented in the *getLease* method of *ComponentServiceInterface* interface. The *getLease* method returns a Lease object that can be transmitted over the network. Therefore, an instance of *usThread* has to find the minimum value of the expiration time of all its dependent component service for deciding the expiration value of itself. Moreover, by adopting the Jini leasing model, the union service decides which specific component service in the union can change its accessibility. Also, the *usThread* instance can manage the status of the union service, and further provide users an advanced service allocation administration.

B. Event Model

The Jini distributed event model allows one distributed component to register interested events and receive notifications of events generated by another component. In its simplest form, an object that wants to be notified of an event's occurrence first registers its interest to the event generator. The event notification registration is leased. The event generator will maintain the notification registration for the notification receiver. Also, it will send event notifications to the receiver as long as the receiver continues to renew the lease. If a partial failure prevents the event receiver from being able to renew the lease, the event generator will remove the event notification registration and discontinue notifications.

Fig. 6 illustrates the event model design for the proposed architecture and the event dispatching process.

© 2006 ACADEMY PUBLISHER

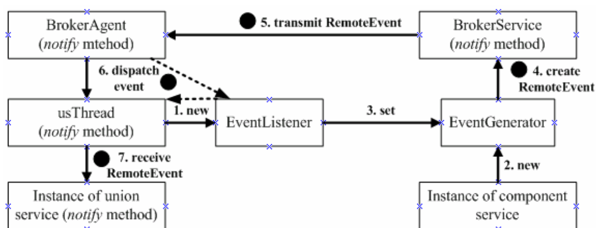

Figure 6. Event model and delivering process

The proposed architecture adopts a simple but effective design, the generator-listener event model, where every *ComponentService* instance creates an *EventGenerator* object to send events and an *usThread* object representing a union service creates an *EventListener* object to accept related events.

When an event appears (e.g., network topology or service status changes, a service detaches form Jini federation), it is reported by an *EventGenerator* instance and later passively received by an *EventListener* instance. The *EventListerer* then passes this event to the corresponding *usThread* instance and later the related *UnionService* instance. The *EventGenerator* interface defines only one method, *setEventListener*, which is used to set the listener for events. The *EventListener* class follows the standard Jini and Java RMI design pattern and also implements the *RemoteEventListener* interface that defines the *notify* method. For posting events, an *EventGenerator* instance invokes the *notify* method of the *EventListener* instance that previously has been set.

V. Practical Example

Here, the example is a virtual "copy machine" service that is composed of a physical "scanner" service and a physical "printer" service. If a company has installed M scanners and N printers, logically there could be M×N copy services (virtual copy machines) - of course, some physical restrictions and the differences between hardware specifications will reduce this number. When all real copy machines are busy or occupied, those M*N copy services could provide an alternative solution. Thanks to SOA, the existing investments (in legacy systems or devices) could be protected. Through the help of the proposed framework, for example, we can build a mobile office, which is filled with mobile devices and wireless printers so employees are permitted to do the copy job at anywhere, at anytime. However, since our target here is not to develop a fancy business copying solution, we simplify the problem to one scanner plus one printer. Also, we concentrate on the behavior of the combined service, i.e., how the framework generates it, and how developers comply with the framework' programming interface.

Fig. 7 illustrates all significant Java classes participating in this example. In Fig. 7, both *ScanService* and *PrintService* are located on the servers or devices actually. They play the role as an RMI server defined in Fig. 3, while *Agent* is a standard RMI client.

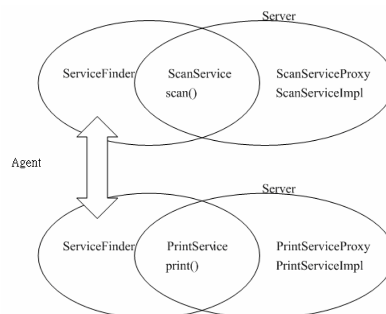

Figure 7. Classes in the example.

As we described before, an RMI client communicates with some RMI server via the connection built between the stub and the skeleton. In Fig. 7, *ScanServiceImpl* provides the stub for *ScanService* as well as the implicit skeleton which is automatically created by Java RMI compiler. So does *PrintServiceImpl*. Moreover, *ScanServiceProxy* is a special Java interface extending *java.rmi.Remote* and *ScanService*. Similarly, *PrintServiceProxy* extends *java.rmi.Remote* and *PrintService*. Last but not least, *ServiceFinder* in Fig. 7 is a utility used to find some Jini service by specifying the corresponding interface.

The abstract model for the example consists of an *Agent* instance and a database, as represented in Fig. 8. Here, service A and B resisters to JLL respectively. Of course, at the same time, both let their stubs be hosted on JLL. After it discovers service A and B via the standard Jini LS, the *Agent* in Fig. 8 obtains stub of A and B, later generates the combined service X according to the related combination rule stored in a database, and finally registers service X to JLL. Thus, *Agent* in Fig. 8 corresponds to *BrokerAgent* in Fig. 4.

Implementing this abstract model is straightforward. First, it gets the stub of *ScanService* (A) and *PrintService* (B) respectively. Next, it concatenates corresponding methods to perform the cooperative service, *CopyService* (X). The most important operation in the example is the *copy* method defined in the *CopyService* interface. These two finding steps look redundant, but in reality they are necessary. Although the stub of the *CopyServiceImpl* class exists in the JVM which hosts the *Agent* class that goes through similar steps, they are different objects and the interactions between them should be as simple as possible. Another reason for finding service *ScanService* (A) and *PrintService* (B) again is that, both of them have the possibility of disappearance since every Jini service can join and leave Jini federation arbitrarily.

© 2006 ACADEMY PUBLISHER

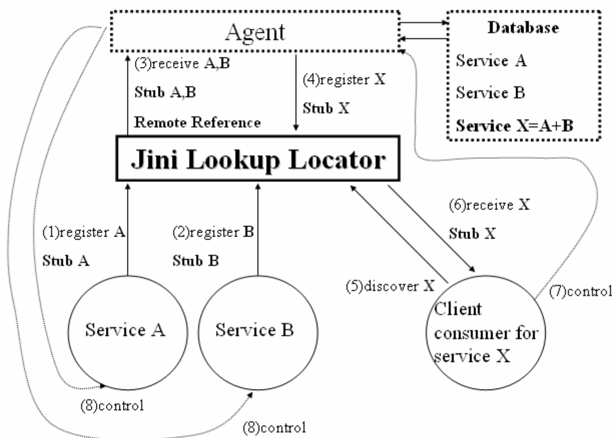

Figure 8. Combining service with remote objects.

In this case, *Agent* has to help construct a standard Java RMI stub for the service X. The service X is a virtual service, but it still acts like a standard Jini service. However, the execution of the service X is different form the previous one. The consumer accesses service X in the way of invoking the remote methods carried by the stub of X after downloading it. Note that, the skeleton of service X exists in *Agent* since it is hosted on the same machine as on which the *Agent* runs. Therefore, the combined operation (A plus B) is executed on the machine that hosts the *Agent* class. For the *Agent* class, the code segment of the *main* method defined in is presented in Fig. 9.

The *Agent* class first finds the "scan" (A) and the "print" (B) service respectively. Later it initializes the "copy machine" service (X). To simplify this example, the combination rule demonstrated here is hard-coded. Nevertheless, a flexible design is to put such rules in a database. On the other hand, for the client, it directly finds the desired "copy machine" service (X), *CopyService*. Next, it invokes the *copy* method of the stub obtained from *CopyService*. When searching *CopyService* (X), the client need not trouble about the *ScanService* (A) and *PrintService* (B) since both are invisible for the client application. Fig. 10 shows the code segment of the *main* method defined in the *Client* class.

Next, let's put the focus on the *copy* method that is defined in *CopyService* interface and implemented in *CopyServiceImpl* class. The former is an interface and the latter is the implementation for the former. The code segment of the *copy* method is shown in Fig. 11.

```
Agent.java – main()

// Step 3 for A (find scan service)
ServiceFinder ssf = new ServiceFinder(ScanService.class);
ScanService ss = (ScanService) ssf.getObject();

// Step 3 for B (find print service)
ServiceFinder psf = new ServiceFinder(PrintService.class);
PrintService ps = (PrintService) psf.getObject();

// Step 4 for X (initialize copy machine service)
CopyServiceImpl.main(null);
```

Figure 9. Method main of Agent.

```
Client.java – main()

// Step 5 (find copy machine service)
ServiceFinder csf = new ServiceFinder(CopyService.class);

// Step 6 (get stub of copy machine service)
CopyService cs = (CopyService) csf.getObject();

// Step 7 (invoke the copy method)
cs.copy();
```

Figure 10. Method main of Client.

```
CopyServiceImpl.java – copy()

// Step 7, 8
public void copy() throws java.rmi.RemoteException {
   System.setSecurityManager(new RMISecurityManager());
   ServiceFinder ssf = new ServiceFinder(ScanService.class);
   ScanService ss = (ScanService) ssf.getObject();
   ServiceFinder psf = new
     ServiceFinder(PrintService.class);
   PrintService ps = (PrintService) psf.getObject();
 ps.print(ss.scan());
}
```

Figure 11. Method copy of CopyServiceImpl.

Straightforwardly, what the method *copy* does is nothing but getting the remote references from *ScanService* and *PrintService* respectively and then calling *scan* and *print* sequentially. As we talked previously, the Jini provides built-in authorization and security mechanism. However, developers are welcome to add their own business logic here. They can enhance the authorization by applying some resource allocation algorithm, or take stricter security model through the help of encryption algorithms.

VI. CONCLUSIONS

The subject of this paper is how to realize the service combination framework in a programmatic way. For brokering services, the proposed Service Broker finds all component services first and then employs them to create a union service according to the corresponding combination rule defined in a database or a knowledge base. This suggests a service consumer directly accesses a virtual union service and is not aware that underlying real component services cooperate to complete the work. We make full use of Java RMI and Jini, so as to provide developers a service combination framework for dynamically creating an intelligent device that have the capability of automatically generating combined or collaborative services.

Specifically, the contribution of this paper is twofold. First, it explains the abstractions used to comprehend such combinations of services, as are inherent to distributed, pervasive or ubiquitous computing. Second, it is argued that the concept of a combined service should be realized through an explicit programmatic construction that represents it at design time and at run time. Moreover, developers can create a more intelligent service or device,

© 2006 ACADEMY PUBLISHER

which can automatically make functional modules or services combined and further configure a set of services flexibly. Consequently, this paper extends the meaning of a Jini service to a set of services.

The future works include research on combination rules and the knowledge base that manages them. Ideally, an intelligent system should evolve new rules form existing combination rules. One approach is to introduce machine learning techniques into the proposed framework. Also, the knowledge base as we discussed earlier is worthy of our continued study since it can provide a flexible programmability and a strong support for the evolvement of rules.

REFERENCES

[1] Thomas Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*, Prentice Hall PTR, August 2005.

[2] Dirk Krafziq, Karl Banko, and Dirk Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices*, ISBN: 0131465759, Prentice Hall, 2004.

[3] Munindar P. Singh and Michael N. Huhns, *Service-Oriented Computing : Semantics, Processes, Agents*, John Wiley & Sons, January 2005.

[4] ZapThink and Jason Bloomberg, *The SOA Implementation Framework White Paper: The Future of Service-Oriented Architecture Software*, ZapThink, LLC, April 2004.

[5] ZapThink and Jason Bloomberg, *Growing an Agile Service-Oriented Architecture White Paper: Achieving Reuse & Loose Coupling through Web Services Delivery Contracts*, ZapThink, LLC, September 2003.

[6] Zoran Stojanovic and Ajantha Dahanayake, *Service-Oriented Software System Engineering Challenges and Practices*, Idea Group Publishing, April 2005.

[7] Gerhard Wiehler, *Mobility, Security and Web Services: Technologies and Service-oriented Architectures for a New Era of IT Solutions,* Wiley-VCH, August 2004.

[8] Thomas Erl, *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, Prentice Hall PTR, April 2004.

[9] Joost N. Kok and Kaise Sere, "Distributed service composition", in *Technical Report No. 256, Turku Centre for Computer Science*, Finland, March 1999.

[10] D. Chakraborty and A. Joshi, "Dynamic Service Composition: State of-the-Art and Research Directions", Technical report TR-CS-01-19, University of Maryland Baltimore County, December 2001.

[11] Gerald C. Gannod, Sudhakiran V. Mudiam, and Timothy E. Lindquist, "Automated support for servicebased software development and integration", *Journal of Systems and Software*, 2003.

[12] Gerald C. Gannod, Sudhakiran V. Mudiam, and Timothy E. Lindquist, "An Architecture-Based Approach for Synthesizing and Integrating Adapters for Legacy Software", in *Proc. of the 7th Working Conference on Reverse Engineering*, November 2000, pp. 128–137.

[13] Paolo Predozani, Alberto Sillitti, and Tullio Vernazza, "Components and data-flow applied to the integration of web services", in *Industrial Electronics Society, The 27th Annual Conference of the IEEE*, volume 3, 2001, pp. 2204–2207.

[14] J. Suzuki and T. Suda, "Middleware Support for Super Distributed Autonomic Services in Pervasive Networks", in *Proc. of the IEEE Workshop on Service Oriented Computing, in conjunction with the 2004 IEEE International Symposium on Applications and the Internet*, Tokyo, Japan, January 2004.

[15] Stephen Langella, "Distributed Data Management and Integration, The Mobius Project", *The Global Grid Forum (GGF11) Semantic Grid Applications Workshop*, Honolulu, HI, Jun 2004.

[16] R.H. Katz, Eric. A. Brewer, and Z.M. Mao, "Fault-tolerant, scalable, wide-area internet service composition", in *Technical Report, UCB/CSD-1-1129, CS Division, EECS Department, UC. Berkeley*, January 2001.

[17] Pierre-Antoine Queloz and Alex Villazon. "Composition of services with mobile code", in *Proc. of First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, Palm Springs, California, 1999.

[18] W. Keith Richards. *Core Jini*. Prentice-Hall PTR, 1999.

[19] UC Berkeley Computer Science Division, *The Ninja Project*, http://ninja.cs.berkeley.edu.

[20] Sun Microsystems, *JSR 78 - RMI Custom Remote Reference*, http://java.sun.com/aboutJava/communityprocess.

[21] Sun Microsystems, "Java Remote Method Invocation Specification, Rev. 1.50", *Sun Microsystems*, Mountain View, California, October 1998.

[22] Esmond Pitt, Kathleen McNiff, and Kathy McNiff, *Java.rmi: The Remote Method Invocation Guide*, ISBN: 0201700433, Addison-Wesley, 2001.

[23] William Grosso, *Java RMI*, ISBN: 1565924525, O'Reilly, 2001.

[24] Fintan Bolton, *Pure Corba*, ISBN: 0672318121, Sams, 2001

[25] Marku Aleksy, Axel Korthaus, and Martin Schader, *Implementing Distributed Systems with Java and CORBA*, ISBN: 3540241736, Springer, 2005.

[26] Frank E., III Redmond, *Dcom: Microsoft Distributed Component Object Model*, ISBN: 0764580442, John Wiley & Sons, 1997.

[27] Nathan Wallace, *COM/DCOM Blue Book*, ISBN: 1576104095, Coriolis Group, 1999.

Hsu, Kuo-Wei received a BS in electrical engineering from National Chung-Hsing University, Taichung, Taiwan, and an MS in computer science and information engineering from National Taiwan University, Taipei, Taiwan. His primary research focus is software engineering, particularly the development of distributed system.

© 2006 ACADEMY PUBLISHER