A NOVEL ARCHITECTURE FOR DYNAMIC LEAST COST ROUTING

Peer Hasselmeyer
Information technology Transfer Office,
Darmstadt University of Technology
Wilhelminenstr. 7, 64283 Darmstadt, Germany
E-mail: peer@ito.tu-darmstadt.de

Abstract: *Private branch exchanges offer a constantly increasing number of features which are traditionally implemented on an embedded processor within the PBX. With the advent of computer telephone integration they now offer methods to communicate with the outside world and let certain features be implemented on external workstations. In this paper we describe how Jini can be used to make legacy PBXs ready for the service age. We use least cost routing as an example service and describe an architecture which allows dynamically exchanging least cost routing data and logic while the PBX keeps running. We also present the experiences gained during implementation and trials.*

KEYWORDS: Computer Telephone Integration, Least Cost Routing, Jini, dynamic components, spontaneous networking.

1. INTRODUCTION

Private branch exchanges (PBXs) offer an ever-increasing number of features which are not directly related to their core functionality, which is connecting telephone users. They may support call forwarding, voice mailboxes, etc. Traditionally, all these functions have been considered an integral part of the PBX and have been implemented on an embedded processor within the PBX. While this solution was necessary as strict real-time constraints had to be observed, the recent explosion of computing power in desktop systems as well as increasing network bandwidth have made it possible to move parts of this functionality out of the closed box of a PBX and onto a connected workstation. This results in advantages for developers as well as for customers. Services can be developed in the familiar environment of a workstation or a PC, and they can be tested more easily as they can be run “in-place” without moving them to the embedded system for deployment. New services and updates of the software can be installed much easier as replacing some piece of software on a desktop computer is much simpler than exchanging a ROM or a hard disk of the PBX (which usually requires the visit of a technician).

While the aforementioned advantages pertain mainly to PBX vendors, the opening of PBXs to the outside world helps other companies as well. A number of different architectures were developed that abstract specific PBX makes and models and allow applications to be written in a PBX-independent style. Third party

companies can therefore write applications that work with a large number of different PBXs without having to know the internal details of any one of them. The most important of these architectures in use today are Novell's TSAPI [3], Microsoft's TAPI [2], and Sun Microsystems' Java-based JTAPI [6]. All of these frameworks allow the development of telephony-related applications in a PBX-independent way. In TAPI and TSAPI the application part is still platform-dependent and usually runs on either a Unix workstation or a Windows PC. In JTAPI, applications are platform-independent thanks to the use of Java.

An important service of a PBX is least cost routing (LCR) which chooses the cheapest service provider based on a set of criteria, most notably the dialed number and the time of day. Traditionally, a least cost routing application has a single database which contains all the different charges of the selected providers (with the number of providers depending on the company providing the least cost routing software). As each telecommunications service provider has a different pricing model, the task of squeezing disparate cost models into one single database can be difficult and tedious. Furthermore, especially after market deregulation, the set of providers on the market is changing constantly and existing providers adjust their tariffs and pricing models a few times a year. This further adds to the complexity of the database and makes keeping it up-to-date close to impossible.

This paper is the result of a research project conducted by Tenovis GmbH and the Darmstadt University of Technology. It presents a novel approach for least cost routing using the Jini connection technology [7] which is shortly described in Section 2. The architecture, which is presented in Section 3, solves the problem of integrating disparate pricing models in one database by using an individual database for each service provider. Some implementation details are mentioned in Section 4. Section 5 demonstrates how a sample query is processed in our system. The results of some performance measurements are presented in Section 6. Section 7 gives a conclusion and describes further work.

2. JINI

Sun Microsystems' Jini is a Java technology, which allows services and clients on a network find each other in an easy and dynamic way. This section is not intended to present an overview of Jini, it rather focuses on the features which are used by the LCR architecture—separation of functionality and code migration.

Separation of functionality means that an application is cut into a number of pieces where each piece implements a single function. Each individual function can be modeled as an independent Jini service. Independence applies to both development and deployment. Each service can be developed without the need to care about the other surrounding services—only their interfaces need to be known in advance. This is equivalent to developing an application in an object-oriented fashion. But while applications written in an object-oriented language are statically linked

together before deployment, each Jini service can be run on its own. The relations between services and clients are established at run-time and can be changed dynamically during the lifetime of the components. If one entity (the client) needs the functionality of another entity (the server), it asks the Jini lookup service for a service implementing a specific interface (which describes the desired functionality). If such a service is found, a proxy object is transferred to the client that can now talk to the service via the proxy. Dynamic reconfiguration occurs when the server becomes unreachable (e.g., because of a broken network connection). The next time the client needs that particular functionality, it asks the lookup service for that interface again. Now, two cases are possible: there is no service available that implements the given interface or another service implements the interface. In the first case, the client can not have its request fulfilled and, in turn, can not perform its service (unless it knows about some other means to replace the missing service). In the second case, it simply uses the service found, resulting in a changed relation between components.

Using the dynamic reconfiguration feature, it is possible to easily implement service migration and improve fault tolerance. Service migration just requires starting the service at its new location and taking the old service down. All applications requiring this kind of service will automatically use the new service. Fault tolerance can be achieved in a similar way. The same service can be started on two (or more) separate machines. Every client will use one of them to fulfill its requests. As soon as one of the servers becomes unavailable, the client will find out by either observing the disappearance from the lookup service or by getting an exception while talking to the server. Both situations result in the client asking one of the remaining servers to fulfill its request.

Above, the transfer of proxy objects was mentioned. These proxy objects can be RMI stubs that simply relay each function invocation to the proxy's origin where the actual service is provided. However, as Jini does not prescribe any protocol to be used between the service and its proxy and as Jini not only transfers the state of an object but also its implementation (one or more Java classes), the proxy can also be a CORBA [4] stub or even a full-fledged Java object that provides the service local to its client. This can be an important feature for scalability. The server does not need to process a large number of requests as they are handled in the client's JVM. Another advantage of a local proxy is the reduction of latency. Calling a remote server's function requires the exchange of a number of packets over a (possibly slow) network connection. This data exchange (and its associated time penalty) is eliminated when calling a function of a local proxy object.

A proxy object does not have to follow the strict distinction of being either local or remote—any level in between is possible at the sole discretion of the service. It is important to note that the distribution of a service is not determined at interface design time but is an implementation matter and can be different for every service.

3. ARCHITECTURE

The least cost routing architecture is based on two ideas: the separation of concerns and the notion of services. Each function of the system is modeled as an individual service. The complete architecture is depicted in Figure 1. It consists of three layers (or four, if you count the PBX as a separate layer) and not only contains the parts which are necessary for least cost routing but also features support for integrated administration which is not addressed in this paper.

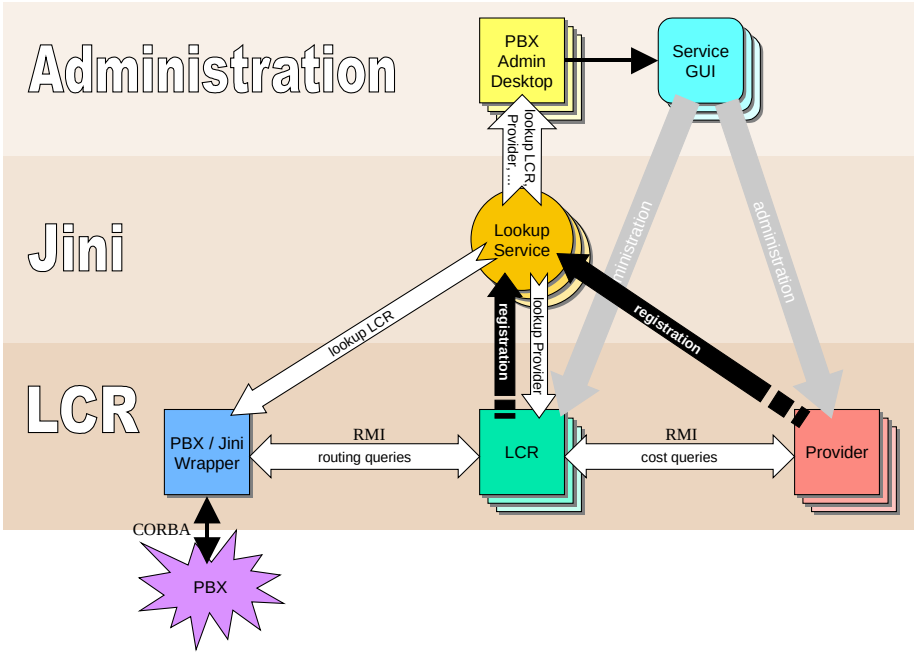

Figure 1 - LCR architecture

The only entity in the Jini layer is the lookup service. This is Jini's main component and is needed to bring clients and services together. Our system adds the three components shown in the LCR layer: a PBX/Jini wrapper, an LCR service, and a provider service. In the figure, arrows connecting components in the LCR layer with the lookup service indicate the direction of proxy transfer. When registering a service, a proxy of that service is uploaded to the lookup service. When looking up services, proxies of matching services are transferred to the requesting entity.

The PBX/Jini wrapper is responsible for translating LCR requests from a PBX to Jini function calls. The wrapper therefore shields the other components from the peculiarities of the communication interface and protocol of the PBX. On the Jini level the wrapper has to connect to available lookup services, get references to an LCR service, and forward requests to it. Queries contain only the called telephone number. As it is possible that one telecommunications provider is currently not able to connect a caller to its destination (e.g., due to congestion), the PBX can

repeatedly ask the LCR service for another (more expensive) provider until the destination is reached. The current state of an LCR query is encapsulated in an LCR session object.

The LCR service receives routing queries from PBXs and has to return data about the currently cheapest telecommunications provider. While clients (PBXs) do not care how an LCR service provides this functionality, in our architecture it distributes cost queries to all known providers, collects the answers, and returns a reference to the cheapest provider. Here, too, queries contain only the called telephone number, but a session concept is not needed. Instead of simply selecting the cheapest provider, further logic can be put into the LCR service. A company might have a preferred provider which should be chosen even if it is up to 20% more expensive than the cheapest provider. This can be easily accomplished by using an appropriate LCR service.

Each provider service models an individual telecommunications service provider. It offers a method for inquiring the current cost of a connection to a given destination. In addition, it can be asked to supply information on how to reach the destination via this provider. Usually, the long distance access code which has to be dialed to use the carrier is simply prefixed to the called number.

The largest benefit of modeling each telecommunications service provider as an individual Jini service is that each service can have its own implementation. This provides a large amount of freedom to implementing different pricing models. The encoding of the pricing model can be freely distributed among the code and a database. While currently a third party would supply the Jini services for all providers, we envision the provider companies themselves operating their own Jini Provider services.

As shown in the figure, there can be more than one instance of each component. While this is obvious and necessary for providers, there is no direct need for more than one LCR service. Multiple copies of the LCR service can help solve a number of problems, though, especially scalability and fault tolerance. If the LCR service gets overloaded by the number of requests, a second instance can be started on a different machine. The load is automatically distributed as PBX/Jini wrappers only query one LCR service and iterate among the available ones. Fault tolerance is achieved in the same way. If a client cannot reach the chosen LCR service, it can revert to querying the next in the list of available ones.

For reasons of simplicity, the figure does not show multiple PBX/Jini wrappers. This is possible, but only makes sense if there is more than one PBX. Of course, one wrapper can take care of more than one PBX.

4. IMPLEMENTATION

Implementing the components of our architecture was straightforward. The component-based architecture was very helpful in implementing the required services. Individual components were implemented autonomously by different people once the service interfaces had been designed. The interfaces between the components are simple Java interfaces describing the desired functionality.

Although Jini is not restricted to a particular protocol for component interaction, we chose Java Remote Method Invocation (RMI) [5] as it is the easiest to use in a Java environment. As mentioned before, the PBX/Jini wrapper abstracts the protocol used by the PBX. Although any protocol can be used here, we were lucky to have access to a PBX that was extended by a CORBA ORB and that issued LCR requests via this interface. This simplified our implementation, as we did not have to worry about writing communication functions for a proprietary protocol.

We implemented two different kinds of providers, which basically differ in the implementation of the services' proxies. One implementation uses plain RMI stubs as proxies. This means that every method call involves network communication. But, as described in Section 2, a proxy can be arbitrary Java code. The second implementation therefore supplied the whole service logic as the service's proxy. A cost query could therefore be answered locally to the client, eliminating the need for communication over the network. While this reduces query latency dramatically (see Section 6), it raises concerns about database consistency, as changes to the master database at the service provider have to be propagated to all proxies. While this could be solved using Jini distributed events [8], we chose an even simpler approach. Every change of the service's database results in the service reregistering with lookup services. Clients are notified about this event by the lookup service and download a fresh copy of the new proxy containing the new database.

Another method we used for speeding up LCR queries is service caching. This method is used by the PBX/Jini wrapper as well as by the LCR service. Usually, when a query arrives, the lookup service has to be contacted and asked for required services. This might slow down access to the service, especially when Java classes for the service's proxy have to be downloaded. We introduced a cache object that constantly monitors (via distributed events) lookup services and retrieves proxies of newly registered services. When a query comes in, the cache supplies the set of currently available services without the need to contact a remote lookup service. The concept of a service cache will be integrated in the forthcoming 1.1 release of Jini [9] under the name `LookupCache`.

5. EXAMPLE

Figure 2 demonstrates our architecture by detailing the message exchange of an example least cost routing query. The figure only shows the interaction between Jini-enabled components. How the connected PBX triggers an LCR request is not shown in this example and depends on the protocol used by the PBX. Furthermore, interaction between the components and the lookup service is left out as well because it is a standard Jini procedure. All services (LCR service, Provider service A and B) must be registered with a lookup service.

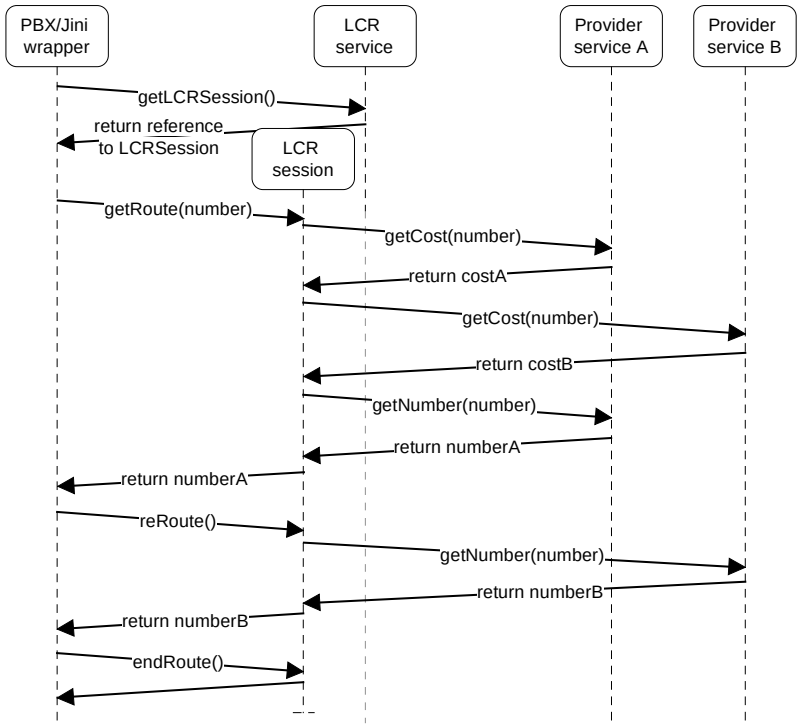

Figure 2 - Least cost routing example

To start an LCR query, the PBX/Jini wrapper asks the LCR service to create a new LCR session. This remotely accessible object handles all further communication. First, the called number is sent to the LCR session object which asks all available Provider services for their current charges to that number. After sorting the providers by cost, it selects the cheapest one (here: provider A) and asks it what number to dial to reach the destination via that provider. The LCR session object passes this number on to its client.

In the example, the destination is not reachable via provider A. The PBX/Jini wrapper therefore asks the session object for the next provider by calling the reRoute() function. As the object keeps a list of providers as well as the state of the LCR query session, it knows which provider is next. It asks that provider for the number to dial and returns that number to its client. The PBX was now able to reach the destination and ends the LCR query session.

6. PERFORMANCE

Performance is a major issue for a PBX. Telephone users do not tolerate long delays between dialing and hearing the ringing tone. It is therefore important that our architecture does not add a large amount of time to this delay. In Section 4 we already described two methods to improve the speed of least cost routing: service caching and local proxies. We did not quantify the performance gain from service caching, although we measured the time it takes to look up a set of services. This time varies widely depending on the number of selected services, the number of different service implementations and their implementation size. As this time can be up to a few seconds, it is always useful to shield time-sensitive applications from this delay by locally caching services.

We did quantify the usefulness of local proxies, though. The measurements are shown in Figure 3. It is obvious that local proxies cut the time needed to fulfill an LCR query dramatically. The setup we used to measure these times consisted of one machine running an LCR service as well as a modified PBX/Jini wrapper which issued and measured the requests. Another machine was running the provider services. Both machines were running Windows NT and JDK 1.3.0 and were connected via an uncongested 10Mbit/s Ethernet LAN.

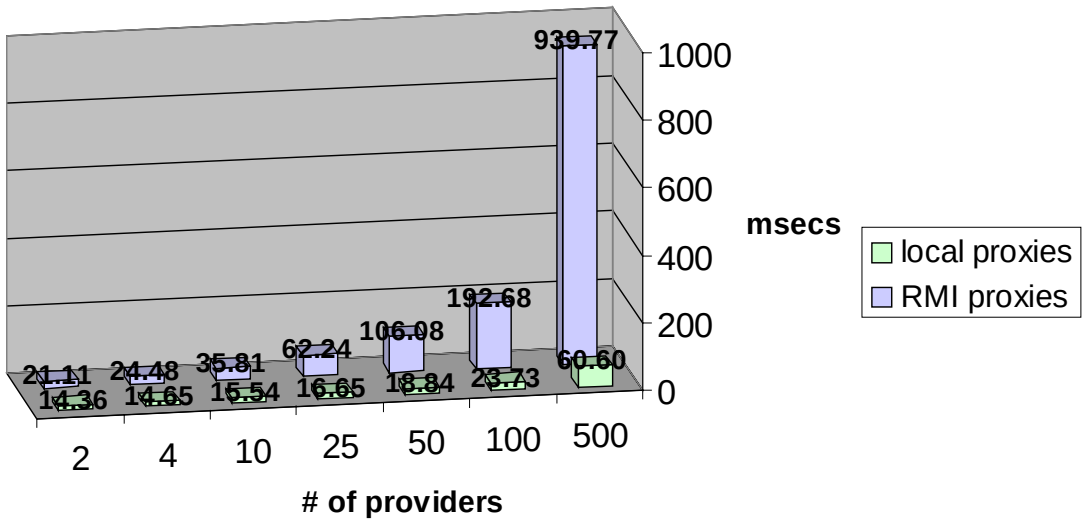

Figure 3 - Local vs. remote proxies

Even when using RMI proxies from one hundred providers, fulfilling an LCR query takes less then ¼ second, which is fast enough for deployment in the real world. We did not test the behavior in presence of a large number of simultaneous requests, though. But as mentioned in Section 3, faster throughput can be achieved by adding further LCR service instances.

7. CONCLUSION AND FUTURE WORK

In this paper we described an architecture that makes legacy PBXs part of a dynamic service community. We showed how this architecture solves the problem of supporting disparate LCR charging models by making each telecommunications service provider a Jini service. The component-based architecture allowed for implementing all components in parallel, speeding up the development process. In addition, the architecture allows for LCR database updates without disturbing the running PBX. We also showed that our architecture is fast enough to be deployed in the time-sensitive environment of a PBX.

An open issue is the deployment of the described architecture. Currently, all components including LCR databases would be supplied by a single third party. As said before, we envision every telecommunications service provider to operate its own Jini service that answers cost queries. We doubt the global acceptance of such a scheme, though.

Supposing that telecommunications service providers accept this scheme, the next issue to be solved is security. LCR services cannot necessarily trust Provider services. A malicious Provider service can always supply zero as its current cost and is therefore always chosen (unless somebody else supplies zero as well). In the "real world", some kind of trust must be established between components in the architecture. This can either be done by mutual authentication or, if only basic trust is required, by an architecture like the one described in [1].

ACKNOWLEDGMENT

I am grateful to Mark Andrew (Tenovis Competence Center) and Dr. Gerd Meister (Tenovis Application Development) for their cooperation and valuable help in this research project.

REFERENCES

[1] Hasselmeyer, P., Kehr, R., and Voß, M. Trade-offs in a Secure Jini Service Architecture. *Trends towards a Universal Service Market (USM 2000)*. Munich, Germany, September 2000.

[2] Microsoft, Inc. *Telephony Application Programming Interface Overview*. http://msdn.microsoft.com/library/sdkdoc/tapi3/tapiover_00c7.htm

[3] Novell, Inc. *TSAPI Documentation*. http://developer.novell.com/engsup/sample/tids/doctsapi/doctsapi.htm

[4] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.3*. OMG document 98-12-01. ftp://ftp.omg.org/pub/docs/formal/98-12-01.pdf.gz.

[5] Sun Microsystems, Inc. *Java Remote Method Invocation (RMI)*. http://java.sun.com/products/jdk/rmi/

[6] Sun Microsystems, Inc. *Java Telephony API*. http://java.sun.com/products/jtapi/

[7] Sun Microsystems, Inc. *Jini Architecture Specification—Revision 1.0.1*. November 1999.

[8] Sun Microsystems, Inc. *Jini Distributed Event Specification—Revision 1.0.1*. November 1999.

[9] Sun Microsystems, Inc. *A Collection of Jini Technology Helper Utilities and Services Specifications—Version 1.1beta*. May 2000.