Transparent Dissemination of Adapters in Jini

Julien Vayssière
INRIA Sophia-Antipolis
2004, Route des Lucioles, BP 93
F-06902 Sophia-Antipolis Cedex, France
Julien.Vayssiere@sophia.inria.fr

Abstract

Jini is a Java-based technology for 'spontaneous' distributed computing which enables programs to dynamically discover nearby services by means of a type-based lookup mechanism. However, this mechanism requires that all the parties involved first agree on a set of common well-known interfaces for describing services which offer the same high-level functionalities. We believe that this mechanism will in some situations prove to be too rigid or complex and that "interface fragmentation" will inevitably happen. We propose a mechanism for automatically disseminating adapters, small downloadable components that convert between types that describe similar services but are yet incompatible. The implementation of the solution consists in the definition of a new Jini service, the Adapter Service, which is a repository for adapters that registers adapter-augmented proxy objects with the Jini Lookup Service when new services appear on the network. This solution does not require any modification to clients, services or to the Lookup Service.

1 Introduction

In this section we first introduce Jini, then explain shortly what are the shortcomings we identified with Jini's type-based lookup mechanism. We then briefly describe our solution and present the organisation of the paper.

1.1 Jini

Jini [1] is a Java-based technology which provides an infrastructure for networked Java services to spontaneously form *federations* and engage in interactions without any prior knowledge of each other.

Jini requires that all participants in a Jini federation are Java programs running inside a Java virtual machine, or at least Java programs that act as surrogates for non-Java devices or applications [16]. As a consequence, Jini can take full advantage of Java's robust type system and dynamic code loading mechanism for looking up and communicating with remote services.

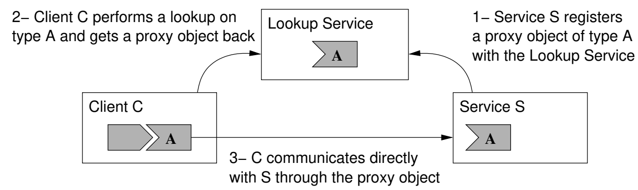

Figure 1. Simplified registration and lookup protocols

A Jini federation contains one or more instances of the *Lookup Service*. The Lookup Service is a normal Jini service which is responsible for keeping track of all the services that are currently available within a federation. The initial reference to the lookup service is obtained through a multicast-based protocol called the *Jini discovery protocol*.

One of the key features of Jini is its *join protocol*: Jini services register with the lookup service by passing to it a serialized proxy object along with a set of attributes (step 1 in figure 1). It is the type of this proxy object that we call the type of the service. Later on, when a client queries the lookup service for a service of a given type using the *lookup protocol* (step 2), the lookup service returns all the proxy objects it knows that match the type. Once deserialized on the client side, the proxy object is ready to handle the communication with the remote service (step 3). The proxy object can be as simple as an RMI stub or as complex as a full-blown client-side application, complete with GUI and client-side caching or fault-tolerance features, for example.

1.2 Limitations of the type-based lookup mechanism

Jini is arguably a step forward in the direction of distributed systems that adapt better to changing environmental conditions. One of the features of Jini that provides higher adaptability than standard distributed object systems such as RMI or CORBA is its type-based lookup protocol which allows us to dynamically discover previously unknown services. However, this mechanism relies on the implicit assumption that all the parties involved in designing service implementations for a given kind of service will manage to reach an agreement on a standard interface for describing this kind of service, for example a standard `Printer` interface for printing services or a standard `Storage` interface for data storage services. We believe that this will not always be the case, and that there will be situations where we have to make do with multiple incompatible interfaces for describing the same kind of service. This problem was also described in [8] where it is termed *interface fragmentation*.

1.3 Adapters and the Adapter Service

This is why we have designed and implemented a solution that aims at solving the following problem: bridging the gap between services that are semantically close (they provide the same functionalities), but nevertheless have incompatible interfaces. We present a Jini service, the *Adapter Service*, which acts as a repository for adapters, the small pieces of software that perform conversions between interfaces. The Adapter Service transparently populates the lookup service with adapter-augmented proxy objects so that a given service instance now appears under different types instead of only one. Our solution does not require any modification to Jini and remains perfectly transparent to the services, the clients and even to the lookup service.

1.4 Organisation of the paper

This paper is organised as follows. Section 2 presents the problem in greater detail and illustrates it with an example. Section 3 explains what adapters are and how the *Adapter Service* can transparently enrich a Jini federation with adapters. Implementation issues are discussed in section 4. Some related work is presented in section 5 and section 6 presents concluding remarks and possible future work.

2 The problem

In this section we present in detail both the problem and the assumptions we made. We also provide an example of the problem that will be used throughout the remainder of the article.

2.1 The need for standard interfaces

As we briefly explained above, the discovery, join, and lookup protocols of Jini altogether provide a simple and elegant solution to the problem of connecting together networked Java entities that have never met before.

However, it is not exactly true to say that entities in a Jini federation need not have prior knowledge of each other in order to be able to interact. Client programs still need to know, at as early as compile-time, the interface under which services will register with the lookup service at runtime, i.e. the interface implemented by the proxy object that will be uploaded into the lookup service by the implementation of the service.

This is arguably one of the most common misconceptions about Jini. What Jini actually does is to simplify the deployment of services and their discovery by the clients: it is no longer necessary for each client to implement the client-side part of the communication protocol since the code for the communication protocol is contained within the proxy object downloaded by the client from the lookup service at runtime. But Jini still requires that the programmer who writes the client-side code knows the interface implemented by the proxy object, even if the actual implementation class of the proxy object is not known until runtime. Jini goes one step further than traditional distributed object systems, but not as far as to completely eliminate the need to agree beforehand on well-known interfaces for describing standard services.

The fact that clients need to know at compile-time the type of the proxy is not a problem in itself. It actually seems to be the only way to proceed, since it is difficult to imagine how a client program could guess the purpose of a service interface it discovers at runtime without any other information than the syntax of the interface. Standard introspection techniques, such as that used in the JavaBeans API[14], do not apply because what we are dealing with here is the extraction of the semantics of a service, i.e. its purpose and its behaviour, out of the bare syntax of its interface. There is a wealth of semantic information that is not captured in the syntax of interfaces. This information is usually contained in the comments of the code or in the documentation, and hence is only accessible to humans. These informations are found elsewhere, usually in code comments or documentation, and are meant to be read by human programmers, not processed by programs.

Jini's answer to the problem is to require that the client knows at compile-time the interface under which the service the client is interested in is registered. As Jini is meant to be used in very open environments, it is of paramount importance that all the services that provide the same high-level functionalities (for example, printing a document) agree on a single common interface for describing the service, as opposed to multiple proprietary interfaces.This is why there is a large ongoing effort within the Jini community to agree on standard interfaces to describe all kinds of useful services, such as printers or storage devices, for example. It also appears that this standardisation process is a long and tedious one, if not unfruitful.

Nevertheless, we believe that this standardisation process, although extremely useful for a large number of service types, cannot be generalised to all kinds of services. As Jini gets more momentum, situations will appear where multiple vendors will provide the same high-level service under different incompatible interfaces. This may be because none of them had the willingness to lead the standardisation effort, because vendors feel a common interface would hide the specific features of their product that gives them a competitive edge, or simply because some vendor deliberately attempts to establish their product and its proprietary interface as a *de facto* standard.

2.2 An example

Let us now illustrate by means of an example the problem of having services that are semantically close but syntactically incompatible.

Let us assume that we have two Jini-enabled temperature sensors that we bought from two different vendors. These vendors, for some reason, never managed to agree on a common interface for describing temperature sensors. As a result, we have to deal with two interfaces that are not type-compatible but which, in the end, describe two services that implement the same high-level functionality, namely measuring a temperature and making it available as a `float` number. Here is the code for the two interfaces `FooTempSensor` and `BarTempSensor`.

```
public interface FooTempSensor {
  public float getMeasurement ();
}

public interface BarTempSensor {
  public float getTemp ();
}
```

If the client software was written for communicating with sensors that implement the `FooTempSensor` interface, it will not be able to deal with sensors that implement the interface `BarTempSensor`. One solution would be to make the client software aware of both interfaces, but this has a major drawback: all the different interfaces need to be known at the time the client software is written. If a new vendor with another incompatible interface appears in-between the writing of the software and its deployment, it will not be possible to use it with this version of the client software. This is clearly a step backwards since Jini technology is very dynamic and runtime-oriented in essence.

As we will see in the next section, our solution based on adapters provides an easy and transparent way to convert, for example, between `FooTempSensor` and `BarTempSensor`.

2.3 What is exactly the type of a Jini service ?

Before we move on to describing what the Adapter Service is, we think it is important to make clear what we mean when we refer to the *type* of a Jini service.

What we call a type, in general, is simply a standard Java type as specified in the chapter 4 of the Java Language Specification [5]. More specifically, a type can be any of the following three things: a primitive type like `int` or `float`, or a class, for example `java.util.Vector`, or an interface like `java.io.Serializable`. At the language level, Java types are reflected as instances of the class `java.lang.Class` [15].

A Jini service does not have only one type but may have many types. This is because what we call the type of a Jini service is in fact the type of the proxy object that the service registers with the lookup service and that is later downloaded by the client. The proxy object is an instance of a class that extends a superclass and quite likely also implements one or more interfaces, one of which is the interface that describes the functionalities provided by the service. This is why there are more than one type compatible with a given proxy object.

Let us illustrate this property with a well-known Jini service, the Lookup Service. In the implementation provided by Sun, the proxy object for the Lookup Service is an instance of the class `RegistrarProxy`. This class directly extends `java.lang.Object` and implements three interfaces: `ServiceRegistrar`, which is the interface that describes the functionalities provided by the service; `Administrable`, which is an interface most Jini services implement to provide access to an administration interface; and `java.io.Serializable`, because the proxy object needs to be serializable. As a result, the Lookup Service is compatible with five different types: the two class types and the three interface types that we just mentioned.

For the remainder of the paper, it is understood that the type (singular) of a Jini service refers to the type of the interface that describes the functionalities provided by the service. It is not possible to programmatically determine which of the different types that are compatible with a proxy object is the actual type of the service, but this problem is alleviated because, by design, the input or output types returned by the introspection interface for adapters are types that describe functionalities. In the example above, the type of the service would be `ServiceRegistrar`.

3 Adapters and the Adapter Service

In this section we first present adapters, then we examine how the Adapter Service populates the lookup service with adapter-augmented proxy objects, which we illustrate by building on the example introduced in the previous section.

3.1 Adapters

Adapters [3] are small pieces of software that convert an interface into another interface that a client expects. It basically translates method calls it receives on its *input interface* into method calls it sends to another object that implements the *output interface*. Often, the two interfaces are not type-compatible (otherwise there would be no need for an adapter, an ordinary type cast would suffice) but they nevertheless describe services that are sufficiently similar to allow the adapter to perform its task transparently, as seen from the client side.

As both design patterns are very frequently used in distributed systems, we would like to stress the difference between the *Adapter* and the *Proxy* design patterns in order to avoid confusion. Both design patterns look similar because they are usually seen as a sort of 'tunnel' with one input and one output, but the Adapter pattern is in fact the opposite of the Proxy design pattern. An adapter converts between two different types but leaves the semantics unchanged, while a proxy object appears under the same type at both ends but modifies the semantics of the calls, for example by turning local calls into remote ones.

All the adapters we consider in this article are assumed to have been written by hand by human programmers, as opposed to adapters automatically generated by computers. What we describe in this article is a way to disseminate adapters transparently in a Jini federation, but not at all a way to create adapters.

In our implementation, all the adapters are required to implement the interface `Adapter` in order to provide support for introspection. This interface provides information about the input and output types for the adapter and also allows us to get and set the adapted object, which is essential for plugging the adapted proxy object into the adapter or chaining adapters together. As we will see later in section 4.2, this introspection feature is also very useful for detecting already-adapted services as their appear in the Lookup Service and thus avoiding infinite chains and cycles of adapters.

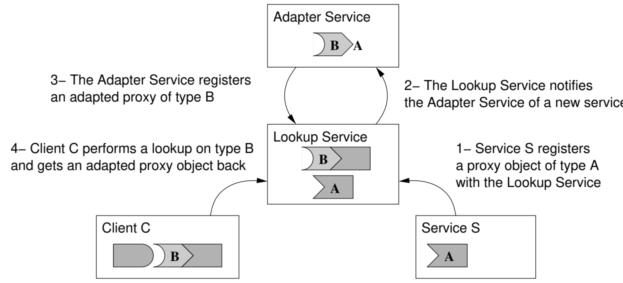

Figure 2. How the Adapter Service comes into play

```
public interface Adapter
{
  public Class getInputType ();
  public Class getOutputType ();
  public Object getAdaptee ();
  public void setAdaptee (Object adaptee);
}
```

3.2 The Adapter Service

The *Adapter Service* is a Jini service that acts as a repository and registering service for adapters. Whenever a new instance of a Jini service appears on the network (step 2 in figure 2), the Adapter Service searches its repository for all the adapters whose output type matches the type of the service that just appeared. For each such match, the Adapter Service first builds a new proxy object that is composed of the adapter and the original proxy to the service (the *adapted proxy)* and then registers this new proxy object with the Lookup Service (step 3). As a result, the service instance that just appeared on the network is now registered with the Lookup Service under more than one types, the original type of the proxy object and the type of all the compatible adapters. All the adapted proxies point to the same service instance, which we call an *adapted service* in this context. Figure 2 summarises how the Adapter Service may enhance the example presented earlier.

3.3 An example

We now come back to the example introduced in section 2.2 in order to show how adapters and the Adapter Service help solve the problem.

We have two interfaces, `FooTempSensor` and `BarTempSensor`, that are type-incompatible but describe services that are basically the same, namely a temperature sensor. As our client software only understands the former interface, we want to write an adapter that would allow the client to talk to sensors that implement the later interface without having to modify the client. This adapter features `FooTempSensor` as the input interface and `BarTempSensor` as the output interface[1]:

```
class TempAdapter implements FooTempSen-
sor, Adapter {
  protected BarTempSensor theAdaptee;

  TempAdapter (BarTempSensor s) {
    this.setAdaptee (s);
  }

  // This is the only method in inter-
face FooTempSensor
  public float getMeasurement () {
    return theAdaptee.getTemp ();
  }

  // These methods come from inter-
face Adapter
  public Class getInputType () {
    return Class.forName (``FooTempSensor'');
  }

  public Class getOutputType () {
    return Class.forName (``BarTempSensor'');
  }

  public Object getAdaptee () {
    return this.theAdaptee;
  }

  public void setAdaptee (Object adaptee) {
    this.theAdaptee = adaptee;
  }
}
```

The simplest adapters only perform pure syntactical conversions, for example when two methods appear with different names in the input and the output interface but do exactly the same thing. An adapter may also convert the data types that are passed as parameters of the methods it implements.For example, converting a call to a method that takes two `float` arguments for the real and imaginary parts of a complex number into a call to a method that takes a `ComplexNumber` object as an argument results in no loss of information.

4 Inside the Adapter Service

In this section we address the implementation of the Adapter Service. We first detail how the Adapter Service keeps track of all available adapters and services, then we study the impact of using the Adapter Service on the number of proxies registered with the Lookup Service and finally we address how the Adapter Service deals with some of Jini's idiosyncrasy like attributes, service IDs and leases.

4.1 How the graph of types evolves over time

The data structure at the heart of the implementation of the Adapter Service is a graph that describes all the types that are known to the Adapter Service and how these types are connected together with either adapters or inheritance relations. This graph is used for two main purposes: first, when a new service is found, the Adapter Service searches the graph to determine the matching adapters for the type of the newly-found service and, secondly, when a new adapter is found, the Adapter Service searches the graph for services that the adapter can be plugged into.

The nodes of the graph are the types that are known to the Adapter Service. The types known to the Adapter Service come from two different origins: they are either the types of services that exist inside the Lookup Service or the types of the adapters contained in the repository of the Adapter Service. In the earlier case, all the types compatible with the proxy object for the service are added to the graph. In the later case, only the input and output types obtained through introspection of the adapter are added to the graph. The arcs of the graph represent the adapters. An arc from type A to type B means that there exist an adapter that converts calls sent to A into calls to an object of type B. The graph is a directed graph because an adapter that adapts type A to type B does not necessarily adapt from type B to type A.

On startup of an instance of the Adapter Service, the graph is empty and then new nodes are added or removed on the occurrence of the following events:

A new service is discovered. First, the set of all the types that are compatible with the proxy object of the new service is computed. For each element of the set, the following operations are performed: if the graph does not already contain a node for the type considered, a new node is created and added to the graph. The node starts with no arcs connected to it since there is no adapter with this type, either as input or output. In the opposite case, if the graph already contains a node for the type, the Adapter Service determines all the possible chains of adapters that lead to this type. For each such path in the graph, a serialized object is constructed that contains all the adapters in the chain and the original proxy object for the service as the last element in the chain. It is this composite serialized object that is the proxy object passed to the Lookup Service for registering the adapted service.

A service is discarded. The Adapter Service cancels the

[1]The code excerpts shown here have been edited for clarity and length.

registration of all the composite proxy objects that were generated as the consequence of the discovery of the service that just got discarded.

A new adapter is discovered. New adapters may be discovered at runtime because instances of the Adapter Service are designed to share adapters between them. When a previously unknown adapter is discovered, new nodes are created if necessary for representing the input and output types for the adapter, and an arc is added to the graph to link the input type to the output type. The Adapter Service then computes the set of the types that are reachable from the input type of the adapter. For all the elements of the set that correspond to an actual service, the service is registered with the lookup service as a proxy that contains all the adapters on the path from the newly-found adapter to the proxy for the service.

An adapter is discarded. Since the Adapter Service maintains a persistent repository of adapters, we only include this case here for the sake of completeness.

The Adapter Service may be queried directly for available adapters, just like any other Jini service. However, the preferred mode of operation is to let the Adapter Service work transparently in the background and make the adapters available through the Lookup Service, so that client Jini entities can use adapters without having to know about it. It works in the following manner (see fig 2):

A service S starts up and registers a proxy object of type A with the Lookup Service. Since, by design, the Adapter Service is an observer for the Lookup Service[2], it receives a notification that a new service S of type A has just appeared. The Adapter Service then performs a search of its repository for an adapter with an output of type A. If such an adapter is found, the Adapter Service registers service S by passing the Lookup Service a proxy object that consists of the adapter that converts type B into type A and the regular proxy object to S. As a result, a new service appears in the Lookup Service. This service is of type B and its implementation is S. Service S is therefore known under two different types, A and B.

4.2 Avoiding infinite chains and cycles

When an instance of the Adapter Service registers an adapted proxy with the Lookup Service, the Lookup Service notifies all interested listeners of the availability of the new service. As the instances of the Adapter Service are listeners for this type of event, they receive such notifications, which may lead to generating chains of adapters of infinite length if these events are not handled correctly.

Imagine a situation where the graph of the types inside the Adapter Service contains an adapter from type A to type B and another adapter from type B to type A. If a service of type B registers with the Lookup Service, the Adapter Service will therefore register an adapted version of this service that consists of an adapter from A to B and the original proxy for the service. As a consequence, the Adapter Service will receive a notification from the Lookup Service that a new service with type A has just registered, which will in turn trigger the registration of an adapted proxy of type B for this service (using the adapter from B to A), and we end up with an infinite loop. The Adapter Service will keep on registering adapted proxy objects of type A and type B, alternatively, with each proxy object containing one more adapter than the previous one.

This problem could be avoided by having the Adapter Service keep track of the already-registered services (by using the unique service ID for example) and avoiding registering services when a cycle in the graph is detected. Nevertheless, this only solves the problem in the case with a single instance of the Adapter Service on the network, but not with multiple instances of the Adapter Service.

Let us say that we have two instances of the Adapter Service in the same federation. The first instance contains an adapter from A to B and the second instance contains an adapter from B to A. None of the instances contains a graph that shows cycles, yet an infinite number of adapters will be registered because each instance is notified when the other instance registers an adapted proxy with the Lookup Service. Every time the first instance registers an adapted proxy with type A, the second instance of the Adapter Service registers an adapted proxy of type B, and we are back to the same problem we had with a single instance of the Adapter Service.

Hopefully, the introspection mechanism described above (see 3.1) provides a solution to the problem, whatever the number of instances of the Adapter Service is. Just like before, whenever an instance of the adapter service is notified of a new service, it searches its graph of types for all the adapters that can be plugged into this type. The difference is that now the Adapter Service avoids registering an adapted service if the adapter in question already appears in the proxy object of the newly-discovered service. This is possible because the newly-discovered proxy for the service is first instrospected in order to determine if it is composed of adapters, and, if so, what these adapters are.

As a conclusion, the proxy object for a service can contain an arbitrary number of adapters in front of the actual proxy for the service, but we now have the guarantee that each adapter is different from the others, which is sufficient to prove that infinite chains and cycles of adapters cannot

[2]Any object that is capable of receiving remote events can subscribe to the Lookup Service in order to receive notification whenever a service instance is added or removed from the Lookup Service.

accidentally appear.

4.3 Controlling the number of registered services

For each instance of a service that appears on the network, the Adapter Service may register a potentially large number of adapted proxies for the service, in the case where the Adapter Service contains a large number of adapters that have the type of the newly-found service as their output type. This is why we want to study how the number p of proxy objects registered with the lookup service evolves for a Jini federation that contains n instances of services. In the standard case with no instances of the Adapter Service, p is equal to n.

If we consider that adapters are only written by hand by human programmers, we can safely assume that new adapters appear far less often than new service instances. As a consequence, the number of adapters (or chains of adapters) that have a given type as their output type can be considered as a constant. Let us call M the maximum over all the possible types of the number of paths that lead to a node in the graph. In other terms, it is the maximum number of adapters of chains of adapters that can be found for a type.

As a consequence, we can state the following gross approximation for the total number p of proxy objects registered with the Lookup Service:

$$n \leq p \leq M \cdot n \text{ and hence } p = O(n)$$

with the above hypothesis about the number of possible adapters for a given type being a constant in mind.

4.4 Controlling the number of adapters

As each adapter performs a conversion from one type to another type, we may have a problem in a situation with a large number of incompatible types that all describe the same high-level functionality. Indeed, if we have n such types, the number of adapters needed to convert between all these different types may be as big as $O(n^2)$. This corresponds to the case where there exist a different adapter for converting each type into any other type. In this case, all the adapted proxies registered with the Lookup Service consist of only one adapter and the original proxy for the service.

But this is a worst-case scenario. If there actually exist a path in the graph that goes through all of the n different types, the number of adapters is reduced to $O(n)$, at the expense of the size of the adapted proxy objects which will be composed of a possibly long chain of adapters and the original proxy object.

This is clearly a problem that requires further investigation. More specifically, we would like, for a given set of adapters, to determine the optimal point in the trade-off between the number of adapters and the size of the adapted proxies registered with the lookup service.

4.5 Dealing with attributes and Service IDs

Looking up Jini services by the type of the proxy object they register with the Lookup Service may in some situations be not precise enough, so we may also want to differentiate further between all the different instances of the same service that are available in a federation. For example, we would like to be able to attach a location attribute to every Jini-enabled printer so that a user can choose the printer nearest to his or her office. This is made possible in Jini by passing a set of attributes at the moment a service instance first registers with the Lookup Service. On the client side, attributes can be specified as part of a standard lookup request, and only those service instances that match the attributes are returned to the client.

In the context of the Adapter Service, the question is as follows: what are the attributes that an adapted proxy should register with ? As we want our solution to be as transparent as possible, it makes sense to decide that the adapted proxy for a service registers with the exact same set of attributes as the original service, since in the end it is the same service instance.

Another issue that we came across has to do with service IDs. When a service instance registers for the first time with the Lookup Service, it is assigned a unique service ID number. This unique ID number is required for handling crash recovery and registration of a service instance with multiple instances of the Lookup Service. Unfortunately, we cannot register the adapted proxy object for a service with the same service ID as the original proxy object for the service because the specification for the Lookup Service allows us only one entry for each service ID. If we were allowed to use the same ID for both the service and the adapter service, it would be impossible to distinguish between registering an adapted proxy and the original service instance registering on startup after a crash.

This is why the adapted proxies register with a different service ID as that of the original proxy for the service. Based on the experience we gained in deploying the Adapter Service, this is not a problem, since client programs perform lookups based on type and attributes and are not supposed to use the service ID directly.

4.6 Handling leases

When a service registers with the Lookup Service, it is granted a lease for a duration set by the Lookup Service. It is the responsibility of the service implementation to renew the lease before it expires. Thanks to these time-bound leases, Jini federations exhibit a *self-healing* property: dangling references are automatically discarded after a finite amount of time if a service instance is no longer available, which contrasts with distributed garbage-collection mechanisms which are known for not handling failures very well.

As adapted proxies register with the Lookup Service just like any other Jini service, they too are granted leases. It is the responsibility of the Adapter Service to renew the leases for adapted services and, just as importantly, cancel leases for adapted services if the lease for the original service expires. As a consequence, one can be sure that there will not be adapters for services that are no longer available.

4.7 Implementation details

Our implementation of the Adapter Service is based on Jini 1.1. It is composed of 12 classes and has a total of about 1000 lines of code, comments included. The implementation is fully functional and was tested on a set of dummy services and adapters, in lack of a better testbed. We hope that, as Jini gets larger acceptance, we will be able to test our system in a real-life setting.

5 Related Work

Jini is not the only player in the field of dynamic service discovery on a network. Other technologies such as Bluetooth, Salutation, Universal Plug and Play or SLP [10] present alternative solutions, but Jini is the only one that features both dynamic code-loading and a single type system for all participating devices. Similar mechanism appeared recently in the realm of so-called *Web Services*, such as, for example, UDDI (Universal Description, Discovery and Integration), which is pushed forward by, among others, Microsoft and IBM.

Nevertheless, a significant amount of work has already been performed on extending Jini's service registration and lookup mechanism. For example, solutions for introducing traditional public-key authentication and encryption into Jini service proxies were recently proposed [2, 4]. A conclusion we can draw from this work as well as from the work we present in this paper is the following: extending Jini hardly requires modifying the Jini specifications, most of the extensions to Jini can be implemented transparently through downloadable proxies or alternative implementations of the Lookup Service specification.

The concept of adapter can hardly be thought of as a new one in the field of object-oriented programming. It has its roots in the old controversy between inheritance-based and prototype-based object-oriented languages [9]. Although inheritance-based languages have been extremely successful in the last fifteen years, we believe prototype-based solutions exhibit runtime-adaptability properties that may prove very useful with dynamic and open systems like Jini [7].

A solution somehow similar to ours was described in 1994 by Trevor and al. within the context of computer-supported collaborative work (CSCW) [18]. Their adapters do not only perform transparent conversions between types but also provide additional semantics to the method calls that they forward, like support for sharing and locking or access control. In terms of patterns, these adapters implement both the *Adapter* and the *Proxy* pattern. Their implementation features an *Adapter Manager* that acts as a central point of coordination and reference for adapters, which is not so different from our own Adapter Service.

Adapters are also routinely used for reusing software components [11, 6]. They are usually written by hand by programmers and, apart from very obvious cases, the automation of the generation of adapters has not produced convincing results yet. This issue was nevertheless addressed by Thatté in [17]. The solution presented builds on results from type theory in order to *largely automate* the process of writing adapters, which means that it is the programmer who still has to take the critical decisions in the end. A solution that also includes the human programmer in the loop was proposed by Schmidt and Reussner in [12]. It models adaptable components by means of finite state machines and then, using programmer input for resolving conflicts and incompatibilities, generates adapter classes.

Finally, we discovered during the writing of this paper that another author designed and implemented an idea similar to ours in the context of Jini [8]. Although his design seems to be similar to ours in essence, the paper does not give enough technical details to determine if most of the problems we investigate in this paper (infinite chains and cycles of adapters, attributes, service IDs, leases, etc.) were addressed or not. Nevertheless, we were quite happy to find out that another author also came to the conclusion that using Jini in large-scale and open environments will inevitably raise *interface fragmentation* problems.

6 Conclusion and Future Work

The problem we addressed in this paper was that of filling the gap between Jini services that are syntactically different (the interfaces under which they are published are not type-compatible) but are yet semantically close (they provide the same high-level functionalities), a problem which is sometimes referred to as *interface fragmentation*.

The solution we designed and implemented makes use of adapters, which are small pieces of software that perform the conversion between a type and another type that is not type-compatible with the former one. All the adapters that we consider here are assumed to have been written by human programmers. Our system is only concerned with transparently and automatically disseminating adapters in a Jini-enabled network, and not at all with the automatic or semi-automatic generation of adapters, which is an orthogonal problem to the one we address in this paper, and a far more difficult one, too.

At the heart of our solution is the *Adapter Service*, a Jini service that registers adapter-augmented proxy objects with the Lookup Service. Such a compound proxy object is composed of an adapter, or a chain of adapters, and the original proxy for the service. As a consequence, it becomes possible to use the type of the first adapter in the chain in order to access the service represented by the original proxy object that is the last element of the chain.

An important property of our design is that it is perfectly transparent to the services, to the clients and also to the Lookup Service.

There is a number of possible research directions that we would like to explore. Class versioning is an important one, since in a distributed and open environment, many different versions of a Java class may co-exist and possibly collide. This problem was extensively addressed in the context of object-oriented databases [13], but received very little attention so far from the distributed programming community. Also, the security issues raised by disseminating adapters cannot be over-emphasised, since a malicious person may want to use the solution we describe for easily spreading viruses. On an almost orthogonal research direction, we hope the problem of interface fragmentation we raise in this paper will provide a new field of application to research in automatic generation of adapters.

References

[1] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini Specification*. Addison-Wesley, Reading, MA, USA, 1999.

[2] P. Eronen, J. Lehtinen, J. Zitting, and P. Nikander. Extending Jini with decentralized trust management. In *Proceedings of OpenArch'2000*, Tel Aviv, Israel, 2000.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1996.

[4] C. Gehrmann and P. Nikander. Securing ad hoc services, a Jini view. In *Proceeding of the First Annual Workshop on Mobile and Ad Hoc Networking and Computing. MobiHOC*, Boston, MA, USA, August 2000.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, USA, 1997.

[6] G. Heineman and H. Ohlenbusch. An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute, February 1999.

[7] G. Kniesel. Type-safe delegation for run-time component adaptation. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 351–366, Lisbon, Portugal, June 1999. Springer-Verlag.

[8] J. Lawrence. Ubiquitous annoyance. *Communicate*, 5(2):54–59, December 2000. http://www.broadcom.ie/communicate/.

[9] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, USA*, September 1986.

[10] G. Richard. Service advertisement and discovery: enabling universal device cooperation. *IEEE Internet Computing*, 4(5):18–26, Sept–Oct 2000.

[11] D. Rine, N. Nada, and K. Jaber. Using adapters to reduce interaction complexity in reusable component-based software development. In *Procedings of the Fifth Symposium on Software Reusability (SSR-99)*, pages 37–43, New York, May 21–23 1999. ACM Press.

[12] H. Schmidt and R. Reussner. Automatic component adaptation by concurrent state machine retrofitting. Technical Report 25/2000, Universität Karlsruhe, Department of Informatics, 2000.

[13] A. H. Skarra and S. B. Zdonik. The management of changing types in an object-oriented database. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 483–495, New York, NY, 1986. ACM Press.

[14] Sun Microsystems. The JavaBeans API Specification, July 1997.

[15] Sun Microsystems. The Java Core Reflection API, 1998.

[16] Sun Microsystems. Jini Technology Surrogate Architecture Specification, 2000.

[17] S. Thatte. Automated synthesis of interface adapters for reusable classes. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 174–187, New York, NY, Jan. 1994. ACM.

[18] J. Trevor, T. Rodden, and J. Mariani. The use of adapters to support cooperative sharing. In *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*, Technologies for Sharing I, pages 219–230, 1994.