Monitoring Component Interaction in Jini Federations

Peer Hasselmeyer, and Marco Voß*

IT Transfer Office, Computer Science Department, Darmstadt University of Technology
Wilhelminenstr. 7, 64283 Darmstadt, Germany

ABSTRACT

Jini is an infrastructure for spontaneous ad hoc service networks. It allows clients to find services without prior knowledge of their network surroundings. For service interaction proxy objects are used which are supplied by service providers. These proxy objects interact directly with the service provider. Compared to architectures that use a virtually central communications broker (like a CORBA ORB or an e-speak Core), this method offers a large amount of flexibility in the selection of an appropriate communication protocol. On the downside, debugging a distributed application using this approach is rather hard, as the interactions between clients and servers are not visible. This paper describes an approach using Java's dynamic proxies that allows component interaction in a Jini federation to be traced. By putting the functionality into the Jini lookup service, the approach is generic and transparent for both services and clients.

Keywords: Jini, component interaction, debugging distributed applications, dynamic proxies

1. INTRODUCTION

Distributed applications are becoming increasingly important. With the explosion of internet use, their deployment will become even more widespread. Testing and debugging "traditional" applications is already a hard task, but debugging distributed applications is even more challenging. One problem is that the applications are running on more than one machine and there is no central debugging station. Especially the communication between individual components of a distributed application is hidden from the engineer.

With the release of the Jini connection technology,[8,9] Sun Microsystems introduced an infrastructure for spontaneously networked components. Being designed for distributed systems, the problems mentioned above apply to Jini as well, although in a slightly different flavor. In a group of Jini-enabled components (a so-called *federation*) the relationships between individual components (clients and servers) are determined at runtime and can even change during the federation's lifetime. This makes debugging a Jini federation especially hard. In addition, interaction between entities of a Jini federation is handled via service proxies which are mobile objects. These objects can implement whichever protocol they see fit. This implies that there is no virtually central authority—like a CORBA ORB[3] or an e-speak Core[4]—that controls and mediates the whole communication. It is therefore not directly possible to trace communication and get an overview of the interactions between components.

The latest Java platform (Java 2 Platform, Standard Edition, Version 1.3, J2SE1.3[6]) introduces dynamic proxies. Dynamic proxies are objects that implement arbitrary Java interfaces. These interfaces do not have to be specified or even known at compile-time, instead, they are set at runtime. This technique offers a powerful tool for dynamically generating wrappers for arbitrary objects. In this paper we describe the use of dynamic proxies for wrapping Jini services with a thin layer that notifies a central logging station about function invocations and returns. It therefore allows tracing the interactions between components in Jini federations. Our approach makes use of a modified Jini lookup service. It is completely transparent to clients and servers, meaning that they do not have to be modified or recompiled.

Section 2 describes the aspects of Jini relevant to this paper. Section 3 introduces dynamic proxies. In Section 4 we describe our architecture that uses dynamic proxy objects for tracing component interaction. A few details about our implementation are given in Section 5. Section 6 discusses a few problems that our architecture faces. Related work is briefly discussed in Section 7. Section 8 concludes the paper and describes some topics for further study.

* {hasselmeyer,mavoss}@ito.tu-darmstadt.de

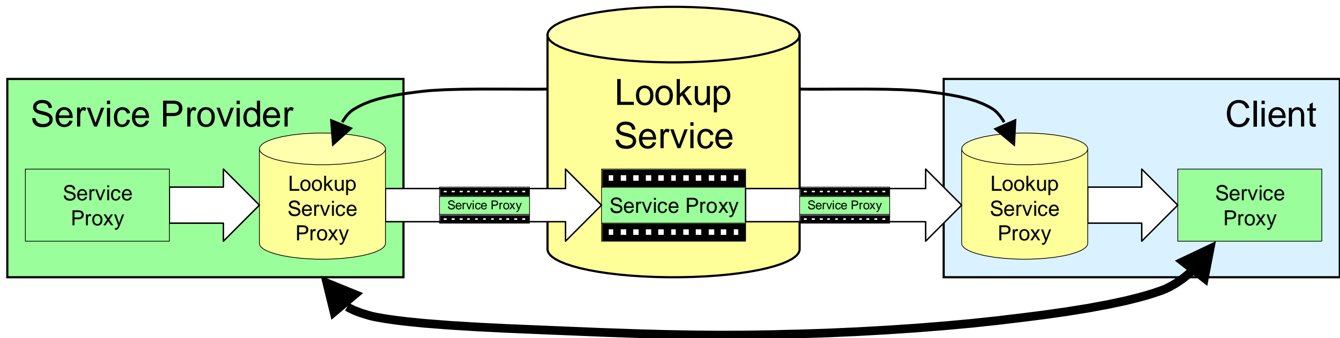

Figure 1. Jini architecture

2. JINI

Sun Microsystems' Jini is a Java technology which allows services and clients on a network find each other in an easy, automatic, and dynamic way. This section is not intended to present an overview of Jini, it rather focuses on the features which are needed to understand the architecture described in this paper: service proxies, object migration, and the role of the lookup service. Furthermore, the terminology used throughout this paper is explained.

A Jini *federation* is the collection of all Jini-enabled components that take part in a Jini system at a certain point in time. This includes entities in the role of servers as well as in the role of clients. The relationships between these entities are established at runtime and can be changed dynamically during the lifetime of the components. The dynamic behavior of a Jini federation is enabled by the use of so-called *lookup services*. Services that want to offer their functionality to a Jini federation contact one or more lookup services and register with them. Clients also contact lookup services and ask for desired services.

Figure 1 shows a more detailed view of the actual interaction. It focuses especially on the migration of objects in a Jini federation. It is important to note that all communication is mediated by *service proxies*. This is true for regular services as well as for the lookup service. Proxy objects are supplied by the *service provider* that they represent. A service is made up of the service provider and a service proxy. The service provider is the back-end and usually provides the actual service while a service proxy is a mobile object that is sent to a client to interface with the service provider. While a service provider only exists once (although it can delegate its work to other machines), the service proxy is copied to each client and it therefore usually exists on multiple clients at the same time.

As shown in Figure 1, service proxies migrate from the service provider to clients via the lookup service. How this migration is performed is not described in the Jini specifications. Only the interface to the lookup service is specified. In Figure 1 this interface is implemented by the lookup service proxy. When registering a service with a lookup service, the service provider supplies a service proxy to the lookup service proxy. The lookup service proxy transfers the service proxy to the lookup service, where it is stored for later retrieval. A client looking for a service asks the lookup service proxy for the desired kind of service. The lookup service proxy sends the request to the lookup service which responds by returning matching service proxies. The service proxies are then handed over to the client by the lookup service proxy.

Although the Jini specifications do not describe how object migration is performed, an obvious choice is the use of the features available in Java, namely *serialization*[7] and RMI classloaders. Serialization is the Java term for transforming the data state of an object into a stream of bytes. This stream can be sent over a communications link to migrate the object. As serialization puts all objects referenced by the initial object into the stream of bytes, usually a tree of objects is transmitted. When recreating the object tree from the byte stream, it is possible that the receiver is missing some Java classes. These classes can be retrieved at runtime via classloaders if serialized objects are annotated by URLs from which the missing classes can be downloaded. Figure 1 distinguishes between "live" and serialized objects (serialized objects are shown with "film strips" in the figure). Service proxies that are stored in the lookup service are shown to be serialized here. This is not a requirement but this is how "reggie"—the lookup service of Sun's reference implementation—works.

Until now, it was assumed that proxies are RMI stubs that simply relay function invocations to the service provider. 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, the proxy can very well be a CORBA 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 service provider does not need

to process a large number of requests as they are handled in the client's Java Virtual Machine (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 or congested) 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. DYNAMIC PROXIES

Dynamic proxies[5] were introduced with version 1.3 of the standard release of the Java 2 platform, J2SE1.3. They allow the creation of objects that implement arbitrary interfaces. In earlier Java version, interfaces implemented by objects could only be specified at compile-time. When creating a dynamic proxy (class `java.lang.reflect.Proxy`) the set of interfaces to be implemented is supplied at runtime and can therefore be varied under program control. This functionality allows the generation of proxies for arbitrary objects. With Java's introspection facilities information on the interfaces implemented by a given object can be retrieved. From this information a dynamic proxy object can be created that implements all given interfaces.

As dynamic proxies are created on-the-fly, function invocations are handled differently from invocations on regular objects. Usually, function invocations are dispatched by the Java runtime system and the developer of a class does not (need to) care about the dispatching mechanism. Dynamic proxies channel all function invocations through an invocation handler object that has to be supplied when creating the proxy. The invocation handler is supplied with enough information to know which function was called on which object and which parameters were supplied. If the dynamic proxy is a wrapper around a regular Java object, it can easily forward the function invocation to the wrapped object by using Java's reflection capabilities.

Dynamic proxies can be sent to remote virtual machines via serialization just as regular objects. Of course, dynamic proxies require object serialization to apply special logic to encode such proxies. As dynamic proxies have been introduced in version 1.3 of the Java 2 platform, earlier versions of Java do not incorporate the required logic and can therefore not serialize nor deserialize such proxies. The use of dynamic proxies is therefore restricted to applications running on J2SE1.3 and higher.

4. ARCHITECTURE

As mentioned before, Jini does not prescribe a protocol for communication between client and server. Instead, the protocol engine is sent as the service proxy to the client. Traditional approaches to monitoring the communication between clients and servers are therefore not applicable. In CORBA, for example, you can intercept all messages because they are mediated by a CORBA ORB. There is no such thing in Jini, as every service might use a different communication protocol. For monitoring service interaction in a Jini federation, a different approach therefore has to be taken.

Our approach for monitoring communication is based on the fact that services in a Jini federation are distributed to clients by the lookup service. The lookup service is therefore a central instance that all components in a Jini system have to talk to. Instead of intercepting all remote function invocations at the communication level (like in a CORBA system) our solution works at a higher level—at the service level. The idea is to not send the real service proxy to clients but to send a modified proxy instead. That proxy intercepts all function invocations and, in addition to calling the function of the real service proxy, sends information about the invocation to a logging service. We call this modified proxy a *monitoring proxy*.

As a monitoring proxy has to appear to clients just like the associated original service proxy, it has to implement all the interfaces the service proxy implements. As the set of service proxies is not known in advance and therefore the set of interfaces is not known either, monitoring proxies must be able to implement arbitrary interfaces which are given at runtime. Java's dynamic proxies offer this functionality. Our solution is therefore based on them.

Figure 2 depicts our architecture. The biggest difference to the description above is that the logging functionality has been split into two parts: the *log distributor* and its clients. The log distributor forwards all log records to registered clients. This has the main advantage of enabling the dynamic replacement of the logging component. In some circumstances it might be more desirable to have a visual representation of the current service interactions than having a log file. Often, more than one log listener is desirable. The architecture makes it easy to visualize current interactions and at the same time put logging data into a file for later analysis.

Our architecture is designed to work as follows. Service proxies are sent to the lookup service as usual. The lookup service wraps the service proxy in a monitoring layer. This monitoring proxy is sent to clients. To clients, the monitoring proxy appears just like the regular service proxy. The client invokes functions of the proxy in the regular way. Before transfering control to the real service proxy (which was sent to the client as part of the monitoring proxy), the monitoring proxy sends information

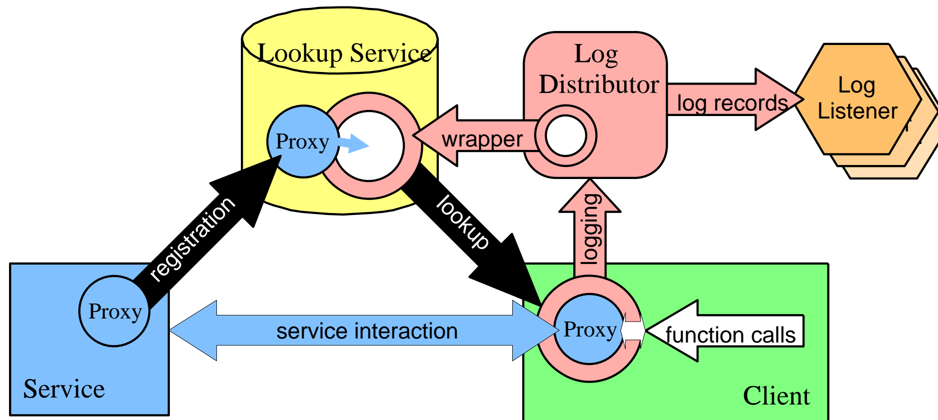

Figure 2. Monitoring architecture

about the function invocation to the log distributor. When the function returns control to the monitoring proxy, information on the return as well as the function result is sent to the log distributor.

5. IMPLEMENTATION

To demonstrate the viability of our architecture we modified "reggie", the lookup service that comes with Sun Microsystem's reference implementation of Jini. A few modifications had to be made to let it support monitoring proxies. In a few areas we deviated from the architecture described in the previous chapter. From the logical point of view, we implemented the architecture as is, but we sometimes shifted functionality to other places to ease implementation. What we changed and why it was done is described in this chapter.

First, some more details on the implementation of reggie are needed. Reggie stores all service proxies in a special internal format. It does not keep "live" proxy objects but rather stores them in their serialized form. This prevents the lookup service from downloading all classes that are needed to recreate the original object. Not downloading the class files results in less memory and bandwidth usage as well as improved performance. Our architecture tries to maintain these advantages. We therefore deserialize the object only at its destination—the client that asked for a particular service. This decision also affects the design of where to create the monitoring proxy.

Before describing the design alternatives, we have to take a closer look at the anatomy of a monitoring proxy. In the previous chapter it was said that it consists of two parts: the service proxy and a wrapper. As the wrapper's job is to send function invocation data to the log distributor, it needs to be able to transfer that data to an associated, but remote log distributor. It therefore needs a proxy of the distributor that knows how to transfer data. We call this proxy the *log proxy*. A monitoring proxy therefore consists of the three parts service proxy, log proxy, and wrapper.

There are (at least) three different solutions for introducing monitoring proxies into the Jini architecture. These are depicted in Figure 3. The first and most obvious solution (Figure 3a) is to create the monitoring proxy at the lookup service and ship it to the client. As the service proxy is stored in serialized format (again shown in the figure by the "film strips") at the lookup service, the monitoring proxy will also contain a serialized copy of that proxy. The original service proxy will be recreated upon the first function invocation. While this solution is the most obvious and does not require any changes to the lookup service proxy, it has the drawback of having to check before every function invocation if deserialization already occurred.

The second solution (Figure 3b) is to create an "empty" monitoring proxy (one that is not yet associated with a service proxy) at the lookup service and send it to the lookup service proxy together with the serialized service proxy. The lookup service proxy then deserializes the service proxy and hands it over to the monitoring proxy before returning the monitoring proxy to the client.

The second solution can even be enhanced because there is no real need to create the monitoring proxy at the lookup service (Figure 3c). It can equally well be created locally to the client by the lookup service proxy. This increases the lookup service's performance, as less data has to be serialized and transmitted. This latter approach was therefore chosen for our implementation. One further improvement is to send the log proxy together with the lookup service proxy. It therefore has to be sent only once.

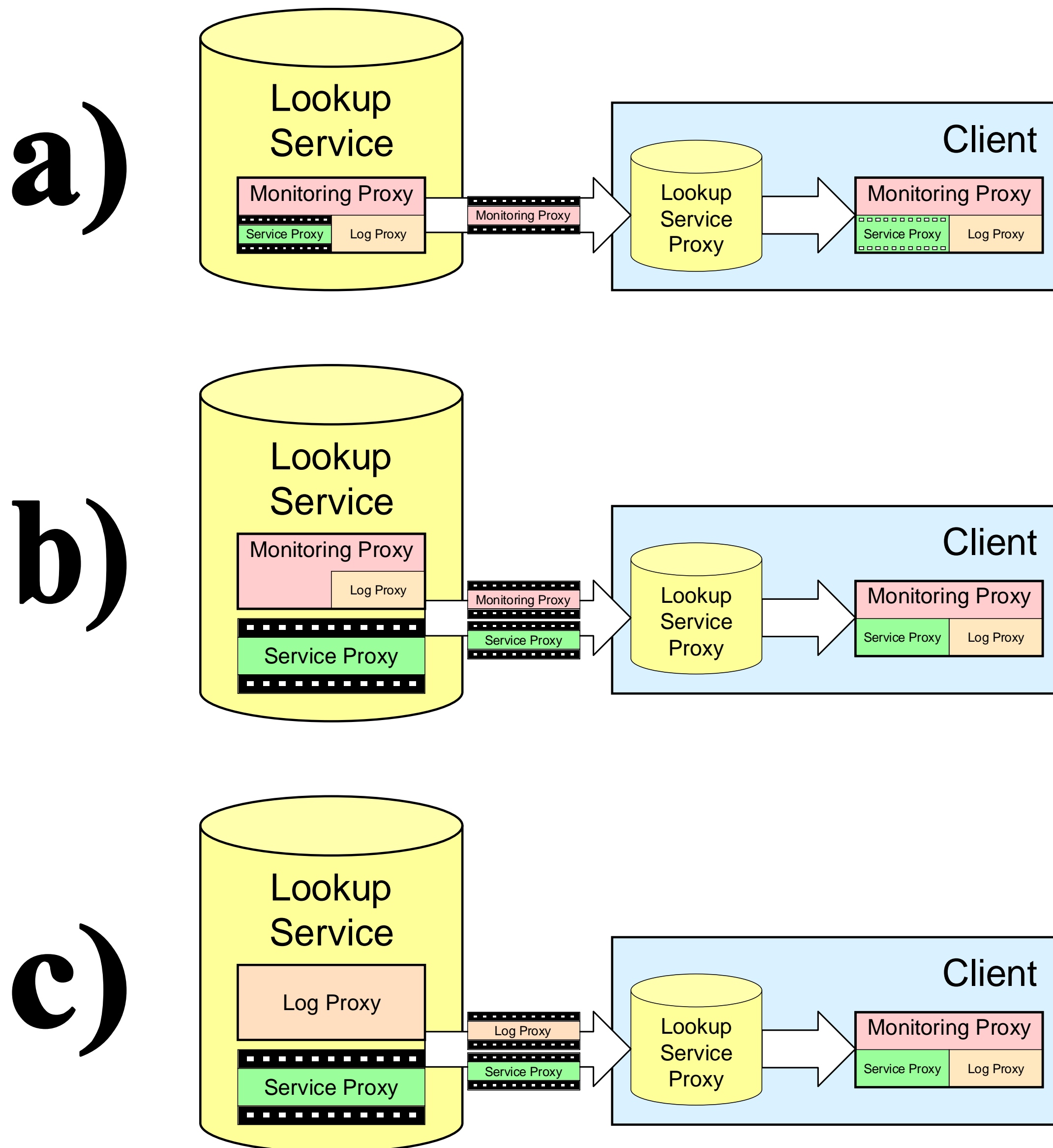

Figure 3. Implementation alternatives

Although our architecture allows for arbitrary distribution of lookup service and logging functionality, we added the log distributor to the lookup service for ease of implementation. As it is a simple RMI server, the log proxy is a regular RMI stub. Another advantage of putting the distributor in the lookup service is easier class downloading. As neither lookup service proxy classes nor monitoring proxy classes are available at clients, they have to be downloaded at runtime. The location to download classes from (the so-called *codebase*) is transferred together with the serialized objects. If the log distributor is put into the lookup service, the classes required for the monitoring proxy can be put in the same place as the lookup service proxy's classes. The codebase does not need to be changed. If both sets of classes are kept separate, the codebase might have to be changed in nontrivial ways.

To intercept all function invocations, all possible ways to acquire a service proxy have to be modified to only supply monitoring proxies. While it was reasonably easy to replace service proxies that are returned by the two `lookup()` functions of the lookup service, there is another way of getting access to services: via notifications. Lookup services allow clients to register listeners which are notified of changes of the set of registered services. There are three possible changes: addition of services, removal of services, and changes to the attributes of services. Besides the kind of change that triggered a certain

event, the event object also contains the proxy of the affected service (unless the service has been removed, in which case the corresponding field is set to `null`). Sure enough, that proxy has to be wrapped by a monitoring proxy as well. This poses a problem as events are not passed through the lookup service proxy. They are rather sent directly to the listeners. The solution therefore was to intercept the registration of listeners as these do go through the lookup service proxy. Instead of registering the real listener a proxy will be registered. This *listener proxy* replaces the original service proxy with a monitoring proxy before passing the event notification on to the real listener.

A big obstacle was to identify individual clients. While it is easy to identify services by their unique service id, nothing comparable exists for clients. To be able to map function invocations to the correct caller, some identification must exist, though. One distinguishing feature usable for identification is the client's IP address. Sure enough, this is not enough as more than one client might run on the same host. Usually, each client and each service runs in its own virtual machine. A feature uniquely identifying a certain JVM is therefore needed. Unfortunately, this is not readily available. As an identification we chose to use the hashcode of the class `java.lang.Object`. This value seems to be unique for each given program. The limitation is obvious: two instances of the same client code run on the same host can not be distinguished. Furthermore, this method depends on the implementation of the JVM and has only been tested with Sun's J2SDK1.3 running on Linux.

Another related problem is that some services are clients of other services (like the storage service in Figure 4 which acts in a service role to the shown client, but in a client role to the accounting and non-repudiation services). It is important to map requests to and from that service to one single instance. We therefore refrained from using the service id as a distinguishing identifier and solely relied on the above mentioned scheme. Obviously, the scheme does not result in easily understandable identifiers of components. We therefore introduced a mechanism to attach names to the components. Services are automatically named according to the attributes they registered with. The default name can be overriden by setting the property `de.tud.ito.jini.monitoring.Name` to the desired name upon startup of a component. The property will be read by the monitoring proxy and used for all log records. The drawback of this approach is that—although it does not require any changes to the code—it requires the startup script to be modified.

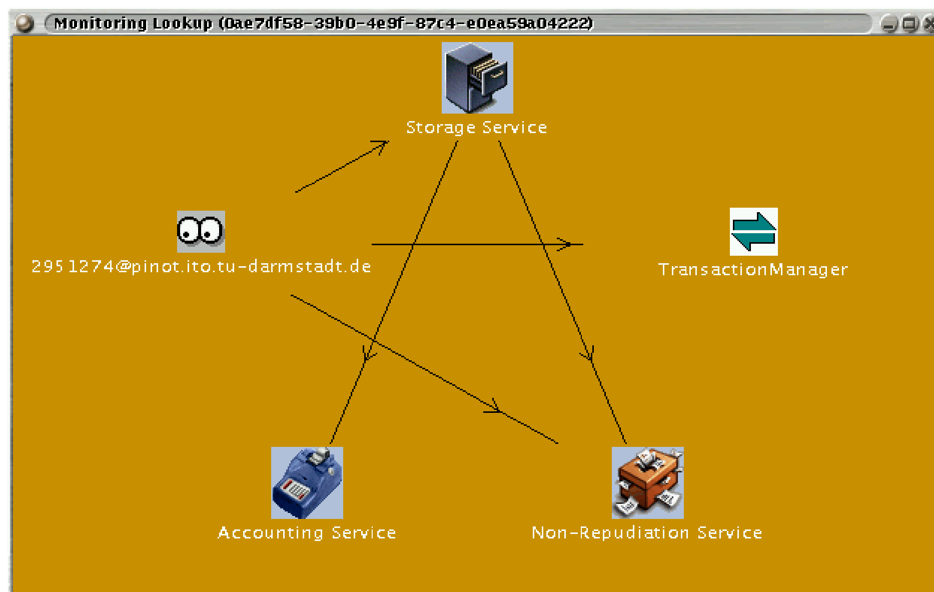

Figure 4. Snapshot of relationship visualization tool

To actually perform logging, we implemented two different listeners for the log distributor. One simply prints all received log records to the screen. The other one is much more sophisticated. It visualizes the current relationships between components. A snapshot is shown in Figure 4.

6. PROBLEMS AND DRAWBACKS

There is one serious problem with our approach. While it works perfectly with simple, "regular" service proxies, it does not monitor function invocations made on other objects that were handed to a client by the service proxy. Two important areas are

affected by this problem: leases and factories. If a service grants a lease to a client, it does so by giving the client a lease object which usually contains a reference to a remote object. Cancellations of the lease can not be observed by our system as leases access remote objects which are different from the service proxy. Only service proxies are monitored, but no other objects, including the lease objects. The same applies to service proxies that act as factories for other objects. The objects created by the factory are not monitored, as they are not wrapped by monitoring code.

The problem here is not that function invocations on other objects can not be intercepted. The problem is to decide which objects need to be monitored. It is usually not usefull to monitor access to all objects which would include for example `Strings`. On the other hand, objects of interest are not necessarily identifiable from their outer appearance, i.e., their classes, and the interfaces they implement. Nevertheless, we think that an appropriate marker for interesting objects is the interface `java.rmi.Remote` because remotely accessible objects usually implement this interface. This is not always the case, though. Interfaces might be defined in CORBA IDL and therefore do not extend `java.rmi.Remote`. On the other hand, objects implementing `java.rmi.Remote` do not have to be remote objects—they can be local objects just as well. But even if they are local objects, there might still be the need for monitoring them. As the question of which objects to monitor appears to be application-specific we did not implement such a functionality.

Another problem is that the end of an interaction between two components is not known. This is not a problem for simple function invocation logging, but for our graphical output client. As soon as an interaction between two components occured, they will be connected by an arrow. But, unless one of the components disappears, the arrow will stay forever, as there is no dedicated function to mark the end of an interaction. The removal of arrows after a certain period of idle time seems possible but was not implemented and it is not appropriate if relationships are to be identified over a long period of time.

An important point to note about our approach is that it does not monitor communication. It monitors access to service proxies, but the proxies do not necessarily communicate with their back-ends—the service providers. Proxies might be able to perform the service locally in their client's Java Virtual Machine. We actually see this functionality as an advantage of our approach, as the first step when debugging a distributed application is to find out where the interaction failed (and this can be achieved with our approach). Only the second step is to find out why the interaction failed which might include monitoring the communication.

7. RELATED WORK

Carp@[2] is an architecture that addresses the same problem that this paper describes. It is also dealing with monitoring Jini federations. In contrast to our work, Carp@ is not transparent to service and client developers. Programmers have to instrument their code by adding so-called "carpat beans" to their programs and by calling dedicated functions that log program communication. Automated byte code instrumentation is proposed, but was not implemented in their prototype. Furthermore, our solution based on dynamic proxies appears to be a lot easier to implement.

Diakov et al.[1] and Wegdam et al.[10] describe solutions to monitor component interaction in CORBA-based distributed systems. They use interceptors to gain access to messages exchanged between components. As said before, intercepting messages at the communication layer is not a viable approach in a Jini system as different protocols can be used by different components. We therefore rely on intercepting calls at a higher level. This is made possible by the exclusive use of Java. Interception at the call level is not generically possible in a CORBA system as components might be written in different programming languages employing different methods of invoking functions. Another advantage of our solution is that no recompiling is necessary. The logging functionality is introduced by using Java's code migration facilities. The approach used in the metioned papers requires recompilation and code changes.

8. CONCLUSION AND FUTURE WORK

In this paper we described a framework for generically allowing the communication flow between components of a Jini service community to be traced. This eases the problem of debugging. Although the system does have some limitations, e.g. sometimes not being able to distinguish two instances of the same code running on the same host, it is a good starting point for debugging Jini federations and helped us in seeing the relationships between components. Despite the problems mentioned in Section 6, the system worked remarkably well.

Our approach is not limited to Jini federations. It can be applied to all architectures that use some kind of proxy for accessing other objects. The same scheme could be used in an implementation of an RMI registry.

Future work might focus on more powerful tools for debugging. Right now it is only possible to monitor function invocations. In a future version of our prototype we want to be able to interactively modify function arguments. By manually fixing

obvious errors in a function call's parameters debugging does not have to stop for every single mistake. We can therefore save the turn-around time between fixing a single mistake, compiling the component, and restarting it.

The current architecture and implementation are usable for debugging Jini systems, but they are also a starting point for managing Jini federations. Management requires more functionality, though. In addition to observing component relationships, management requires the ability to change these relationships. Unfortunately, changing the relationships does not seem to be easily done in a way transparent to component developers. It seems that a component's relationships have to explicitly be made externally visible and changeable by the component developer. Further research is currently conducted on this topic.

REFERENCES

1. Nikolay K. Diakov, Harold J. Batteram, Hans Zandbelt, and Marten J. van Sinderen. Monitoring of Distributed Component Interaction. In *Workshop on Reflective Middleware*, 2000.
2. Michael Fahrmair, Chris Salzmann, and Maurice Schoenmakers. CARP@ – A Reflection Based Tool for Observing Jini Services. In *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 209–227. Springer Verlag, 2000.
3. Object Management Group. *CORBA 2.3 Specification*. `http://cgi.omg.org/cgi-bin/doc?formal/98-12-01.pdf.gz`, June 1999.
4. Hewlett-Packard Inc. *E-Speak Architectural Specification – Release A.03.11*. `http://www.e-speak.hp.com/media/a0/architecturea0.pdf`, January 2001.
5. Sun Microsystems Inc. Dynamic Proxy Classes. `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`.
6. Sun Microsystems Inc. Java 2 Platform, Standard Edition, Version 1.3. `http://java.sun.com/j2se/1.3/`.
7. Sun Microsystems Inc. Object Serialization. `http://java.sun.com/j2se/1.3/docs/guide/serialization/`.
8. Sun Microsystems Inc. *Jini Architecture Specification – Version 1.1*. `http://www.sun.com/jini/specs/jini1_1.pdf`, October 2000.
9. Sun Microsystems Inc. *Jini Technology Core Platform Specification – Version 1.1*. `http://www.sun.com/jini/specs/core1_1.pdf`, October 2000.
10. Maarten Wegdam, Dirk-Jaap Plas, Aart van Halteren, and Bart Nieuwenhuis. ORB Instrumentation for Management of CORBA. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June 2000.