PAAL, Stefan
KAMMÜLLER, Reiner
FREISLEBEN, Bernd

Java Remote Object Binding with Method Streaming

Publiziert auf netzspannung.org:
http://netzspannung.org/about/technology/
24.06.2004

Erstveröffentlichung**:** Proceedings of the 4th International Conference for Objects, Components, Architectures, Services and Applications for a Networked World (NODE 2003). Erfurt, Germany. 2003, S. 230-244.

Java Remote Object Binding with Method Streaming

Stefan Paal[1], Reiner Kammüller[2], Bernd Freisleben[3]

[1] Fraunhofer Institute for Media Communication
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany
stefan.paal@imk.fraunhofer.de
[2] Department of Electrical Engineering and Computer Science, University of Siegen
Hölderlinstr. 3, D-57068 Siegen, Germany
kammueller@pd.et-inf.uni-siegen.de
[3] Department of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Strasse, D-35032 Marburg, Germany
freisleb@informatik.uni-marburg.de

Abstract. The basic programming elements in object-oriented programming (OOP) are objects that interact with each other by virtually sending messages to and receiving messages from other objects. This principle is realized differently in various OOP languages, but the most commonly used approach is the definition of methods and classes which are subsequently used to instantiate objects. Typically, object references are used to bind and call methods on an object. In this paper, we present a new approach to remote object binding and method calling by introducing so called *method streaming*. Caller and callee objects are not directly bound using object references but by binding to a method stream. In contrast to existing approaches, method streams enable the transparent and customizable interconnection of legacy remote Java object implementations on top of conventional network middleware. Thus, method streams do not replace but supplement existing solutions like RMI and CORBA by providing an easy-to-use framework for Java-based distributed applications.

1 Introduction

The major paradigm of object-oriented programming (OOP) is the encapsulation of data and methods within an object and the definition of a well-designed interface to access it. For this purpose, objects send messages to other objects and in turn interpret messages received from other objects. In addition, classes are used to provide templates for creating new objects and to specify their data structure and behaviour. The programming language Java is essentially based on this paradigm. However, like in many other OOP languages, Java objects do not actually exchange messages among each other but they rather use object references to call methods which are defined in the interface of the corresponding class of the object. Thus, an object will never receive messages which it can not interpret. Instead, the Java compiler detects undefined method calls and denies their compilation. In this context, object references play an important role since the compiler evaluates them to determine which class has been used to describe the object, and hence which interfaces and methods are

available. Although the introduction of object references eases the compilation and running of Java applications, they put some contraints on object interaction, in particular with respect to distributed or remote objects. On the one hand, handling remote objects is supposed to be transparent compared to local objects, on the other hand, methods can not be called directly, since host boundaries have to be crossed. For this purpose, many network middleware approaches typically use stubs and skeletons [1] following the *Proxy* and *Adapter* [2] pattern to hide network communication, hence pretending a remote object to be a local object.

In this paper, we present a network middleware approach to Java remote object binding by introducing so called *method streams*. Similar to the widely used data stream concept in Java for transmitting data, we use method streams to transparently transmit method calls to the called object. The approach is not intended to replace existing solutions like RMI, CORBA or SOAP. Instead, it aims to introduce an easy-to-use framework for the implementation of Java-based distributed applications on top of conventional network middleware. The main benefit for the developer is a homogeneous programming interface which completely remains the same for local Java objects and remote Java objects in terms of interface design, implementation and underlying network middleware. In this context, we demonstrate the transparent use of our approach with legacy Java applications and its realization using *Java Dynamic Proxies* and *Java Reflection*. Furthermore, it will be illustrated how to easily add new features like binding aspects and to decouple caller and callee efficiently with only a small impact on performance. Finally, we demonstrate the application of method streaming in distributed environments and show how binding aspects can change dynamically and objects can migrate but related object references do not become invalid. The paper is organized as follows. Section 2 briefly outlines existing network related middleware approaches for binding remote objects. In section 3, the basic ideas of the method streaming approach, its realization and its use are presented. Applications of our approach are illustrated in section 4. Section 5 concludes the paper and highlights areas of future research.

2 Java Remote Object Binding

There are many network middleware approaches to bind remote objects and to call methods across network boundaries. Within this context, one of the most important objectives is the transparent handling of local and remote objects: developers should not be bothered with network related programming tasks, instead, they should be able to call methods transparently on local objects as well as on remote objects. In the following, we consider several approaches of Java remote object binding and how they meet this request.

2.1 Remote Method Invocation (RMI)

The first approach considered is Remote Method Invocation (RMI) [3] which is delivered with each JRE. While it does not provide support for other programming languages than Java (such as, e.g. CORBA [4]) or different networking protocols (like

SOAP [5] does), it is quite easy to use. Remote objects only have to declare a particular interface *java.rmi.Remote* and mark each remotely accessible method within the interface to eventually throw an exception *java.rmi.RemoteException*. As shown in fig. 1, a login service has to define an interface containing the methods to be published and which of them are declared to potentially throw a RemoteException. Accompanying tools like *rmic* will then automatically generate the corresponding stubs and skeletons required for handling the network communication.

```
public interface ILoginService extends java.rmi.Remote
{
   public void login(String szUser, String szPassword)
                     throws java.rmi.RemoteException;
}
```

Fig. 1. Particular declaration of a Java remote interface with RMI

Besides some additional effort required to establish the connection to the remote object (i.e. the binding and lookup of a remote object), the subsequent use of a remote object is mostly transparent compared to a local object. Based on RMI, there is ongoing work towards an extended, more transparent version of RMI called *Transparent Remote Method Invocation (TRMI)* [6]. It uses dynamic proxies and reflection to overcome some limitations of the original RMI. In fact, the approach uses a certain client stub and server skeleton implementation that wraps the underlying RMI code and does neither need to mark remote object implementations with RemoteInterface nor declare remote methods to throw RemoteException.

2.2 Common Object Request Broker Architecture (CORBA)

Another widely used approach is the *Common Object Request Broker Architecture (CORBA)* [4]. Its major goal is the interoperability of distributed objects, regardless of programming language, operating system or system architecture. CORBA was specified before Java was released and it was later extended to include Java applications as well. Therefore, it was not exclusively developed for Java and thus puts some constraints on using it with Java. First of all, CORBA remote interfaces may incorporate not only Java-like parameter types but for example so called *out* parameters resulting in "strange" parameter types like *StringHolder*. Another incompatibility is the casting of remote object references. While in RMI a remote object reference can be transparently casted using the native cast operator, the same has to be done in CORBA using the *narrow* approach. In turn, while CORBA interfaces are normally defined using the so called *Interface Description Language (IDL)*, meanwhile there are tools available which can automatically create them from native Java interfaces. Thus, the developer does not actually see that remote Java objects are called using CORBA. Further, CORBA can transparently intercept method calls using so called *interceptors* which can be used to add features like encryption. In addition, CORBA provides a so called *Dynamic Invocation Interface (DII)* which allows to dynamically call remote CORBA objects without actually having a related stub. However, the depicted limitations concerning Java (like the different casting

approach) remain and remote objects have to be particularly implemented or at least prepared to be used with CORBA. Thus, legacy Java code may have to be adapted manually or even rewritten completely.

2.3 Simple Object Access Protocol (SOAP)

The third regarded approach is SOAP which originally is an abbreviation for *Simple Object Access Protocol* [5]. However, even though the name implies that SOAP is another object-oriented middleware approach like RMI or CORBA, it rather represents a service-oriented middleware approach. Thus, SOAP is not really used to bind remote Java objects but it can be still used to interconnect distributed objects implementing a certain service interface. SOAP also uses stubs and skeletons to hide the network handling as well as a programming language independent approach to define a remote interface. In effect, the *Web Service Description Language (WSDL)* [5] is used similarly to CORBA IDL to create the Java stubs and skeletons automatically. In turn, there are also tools available which create an appropriate WSDL representation of native Java interfaces and allow to generate stubs and skeletons. In contrast to CORBA, SOAP is not bound to a certain networking protocol, though it typically uses the *Hyper Text Transfer Protocol (HTTP)* as its communication protocol and the *eXtensible Markup Language (XML)* as its data format. Thus, SOAP can be used over the Internet, crossing firewalls and bridging platforms without additional software other than a web server and an XML parser. Nevertheless, due to its openness to other programming languages, SOAP introduces comparable problems to CORBA concerning the transparent handling of connections to remote Java objects in comparison to native local Java objects, e.g. different mappings of data types and casting. Finally, compared to RMI and CORBA, it comes with a higher impact on runtime performance since every method call has to be particularly packaged and sent within an XML envelope.

2.4 Other Approaches

There are several other approaches which offer calling a method on a remote Java object. However, like RMI, SOAP or CORBA they can not be used in the same transparent manner as a method call to a local Java object. An example is the *eXtensible Markup Language - Remote Procedure Call (XML-RPC)* [7, 8]. It is also based on exchanging XML messages like SOAP but it has a simpler specification. The XML message is less complex and smaller than the comparable SOAP envelope. On the other hand, the client is much more complicated compared to SOAP and requires additional effort. Therefore, legacy Java code has to be extended with particular network handlers, and every method call has to be manually prepared and packaged in an XML envelope which leads to an impact on runtime performance comparable to SOAP. Certainly, there are further approaches not discussed in detail here, such as JAX-RPC [8, 9], JEDI [10], FlexiBind [11] and Remote Reflection [12]. In general, they differ mainly in the way which network protocol and data format they use to communicate over the network, the degree of hiding the corresponding network

logic from the business logic and the impact on runtime performance. As a result, they do not primarily support the transparent interconnection of remote Java objects in that the same object interfaces can be used as for native local Java objects. Instead, they rather focus on adding certain features like pluggable binding policies [11] or remote reflection [12] and accept the resulting limitations concerning their application to legacy Java code.

2.5 Discussion

The reviewed approaches require some sort of code adaptation to mark remote Java interfaces and use related stubs and skeletons. CORBA, SOAP and XML-RPC are not limited to Java which is of course a major advantage when interconnecting distributed applications written in different programming languages. However, on the other hand Java developers who want their distributed applications to be platform-independent use often only Java for their implementations. They primarily need Java native support for heterogeneous networking environments and want to enable their legacy Java code to be remotely accessible. Although there are approaches like CORBA supporting different programming languages and network protocols, they do not transparently support native Java implementations. For example, in case there are tools to generate stubs and skeleton from native Java interfaces which ease the application of the related approach, legacy Java code has to be still customized, e.g. with extra methods to cast a remote object reference. Furthermore, the discussed approaches inherently try to treat remote object references like local object references. But in contrast to local Java objects, remote Java objects are not tightly bound to remote object references; they can be destroyed or moved to another host, so that remote object references would become invalid. Although, there are mechanisms to support object migration like *interoperable group references* in CORBA they require specific programming knowledge and are differently supported by each approach. In summary, there is no way to separately customize overall binding related aspects from the currently used network middleware or protocol, respectively. Instead, the utilization of certain features such as e.g. spontaneous object migration in peer-to-peer networks (P2P) [13] requires extra business logic code for each approach.

3 Method Streaming

In this section, we present the objectives, concepts and realization of our approach to tackle the problems discussed above.

3.1 Objectives

Instead of defining yet another approach to binding remote objects, we would like to propose a basic model which allows conventional binding approaches to be integrated and exchanged without modifying the business logic. The main aims of our approach are as follows.

Decoupling of Object Binding and Binding Aspects
As mentioned above, existing approaches do often put certain constraints on the programming model. The developer is forced to modify the source code whenever another approach should be used, such as switching from CORBA to RMI. Thus, a new approach should provide a common interface to bind objects independent of binding aspects. Consequently, neither extra implementation effort on behalf of the business logic nor recompilation of the source code would be required.

Transparent Selection and Switching of Binding Aspects
A further limitation of existing approaches is the fixed setting of certain binding aspects (like the networking protocol) after the remote object has been bound. This also leads to another problem when objects have to be rebound and are not longer reachable using the former networking protocol (such as e.g. migrated objects). Thus, it would be preferable to select and switch binding aspects on the fly.

Dynamic Rebinding of Spontaneously Migrating Objects
Conventional approaches tend to assume that remote objects do not migrate. But nowadays, with an increasing interest in peer-to-peer structures, remote objects are not longer bound to a specific platform but are appropriately migrated. Consequently, remote object references should be dynamically rebound to the new location of the object without affecting the actual business logic.

Customizable Interception of Method Calls
In contrast to local object references which are employed without particular impact on a certain method call, remote object calls are intercepted by the network logic. It takes care to encapsulate and transfer the method call from one host to another. Within this context, a particular issue is the dynamically customizable interception and delegation of method calls, especially dispatching calls to migrated objects.

Individual and Transparent Composition of Binding Aspects
Each interceptor should be composable with other interceptors. Thus, they can be used to add object independent functionalities dynamically to arbitrary objects, similar to the approach introduced by Aspect-Oriented Programming (AOP) [14]. This would separate the *streaming logic* defining how a method call is passed from the *business logic* defining where the method call is issued and processed.

3.2 Concept

In this section, we present the underlying concept of method streaming to satisfy the objectives listed above followed by its realization. The basic idea of our approach is to abstract a method call from the actual mechanism for calling the targeted method. Thus, we propose to transparently establish a so called method stream between the caller and callee as shown in fig. 2. Using a method stream, each method call is encapsulated similar to data in a data stream and can be transparently transmitted from the caller to the callee. As a side effect, method calls can be customizably and

transparently intercepted within the method stream, e.g. for bridging the network using a certain networking protocol. Thus, network communication is just a particular aspect of the method stream that can be dynamically customized.

Fig. 2. Methods calls are sent using method streams

Furthermore, a method stream is appropriately configured to connect caller and callee without modification of existing code. Hence, it is particularly suitable for legacy object implementations. As a result, a method stream basically represents a medium for transferring the method call from the caller to a certain callee while being able to transparently customize the method call. Expressed in a more implementation oriented way, it represents a *chain of intermediaries* [15] that can be adapted to the current networking environment and application scenario.

3.3 Realization

After having briefly depicted the basic concept of method streams we now illustrate its realization using *Java Dynamic Proxies* and *Java Reflection* [16]. When a callee object on the remote side is bound to a method stream, a so called *reflection dispatcher* is created that represents the remote end-point of the stream as shown in fig. 3. Its main purpose is to receive the method calls sent over the stream and to call the appropriate methods on the target object transparently. For this purpose, the dispatcher uses reflection to assemble a method call on the fly. In turn, if a caller wants to use the callee object, it retrieves the head of the method stream and creates a *Java Dynamic Proxy* that is dynamically customized to accept methods according to the interface of the callee. Subsequently, it injects the received method calls into the stream by using the method *invoke* which is provided by all links and the dispatcher while implementing the interface *InvocationHandler*. At this point, it should be stressed that the links within the stream do not use reflection to pass the method call to the next link. Instead, *invoke* is directly called avoiding extra runtime overhead. Furthermore, due to the common method *invoke* the links can be also easily arranged and used to inspect and modify the arguments of the method call or the call sequence itself. In this context, commonly required functionalities are enclosed in so called *intermediaries* which can be arbitrarily concatenated and reused among different callee objects. In this sense, the intermediaries represent a *chain-of-responsibility* [17] handling features like logging or access control which can be dynamically inserted and removed [18]. Moreover, since each link of the method stream is automatically associated with a unique *binding id* the caller can not only retrieve the head of the stream but also any link within the stream. Hence, it would even be possible that method calls of different callers are passed through different links to the same callee.

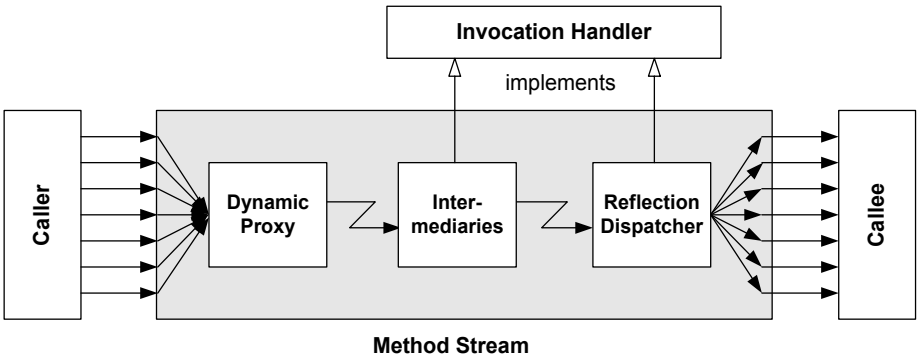

Fig. 3. Method streaming using Java dynamic proxies and reflection dispatcher

There still is a major problem that needs to be addressed: what about the localization and binding of remote objects when the caller has got a binding id of a remote object, e.g. returned by another method call? In the case of local objects, caller and callee are both using the same binding manager and there is no need for a special lookup or network binding. However, in the case of remote objects, the presented approach faces the same problems as conventional network middleware solutions like CORBA. But although our approach also requires some lookup service, the main difference to other middleware solutions is its strict decoupling of *object localization* and *object binding*. The mentioned binding id does not contain any binding information like host address or networking protocol; it is only a global unique identifier for the callee object. Hence, our approach leaves the localization completely up to the lookup service which has to determine the remote host somehow. In this way, we clearly separate the concerns and postpone the decision about which approach for a lookup service should be used in a certain environment as long as the binding id can be clearly resolved. Feasible approaches would be a central lookup service like the *rmi registry* or a distributed registry approach like in *JXTA*. After the host has been located, a particular connection service dynamically creates an appropriate stub to connect the callee. For that, the developer does not have to use or generate an extra network stub/skeleton pair for each new callee type nor to customize legacy Java code, e.g. using a particular dynamic invocation interface as in CORBA. In contrast, due to the common interface of the intermediaries there is always the same remote network interface regardless of the connected objects, hence only a single stub/skeleton implementation per network protocol is needed. Consequently, network handlers supporting a certain network protocol are implemented once as particular intermediaries and can then be dynamically applied in different method streams.

3.4 Use

In the following we briefly illustrate the use of our approach with a simple login service as shown in fig. 4. We want to point out that there is no necessity for a remote object to implement a particular interface or to be derived from a certain class.

```
public interface ILoginService
{
   public void login(String szUser, String szPassword);
}
```

Fig. 4. Native declaration of the LoginService

Before an object can be connected through a message stream, it has to be bound by the binding manager as shown in fig. 5. This creates a new stream that is associated with a unique binding id, identifying the current head of the message stream. In the following this binding id can be used to connect the bound object, in the example the object referenced by *loginSvc*. In addition to the automatically created binding id the method stream can also be named with a customer-defined string like *loginservice*.

```
IBindingManager bindMgr = CBindingManager.getManager();
IService loginSvc = new CLoginService();
CBindingId bindId= bindMgr.bind("loginservice",loginSvc);
```

Fig. 5. Binding and registering a method stream on the server side

In order to connect the method stream, the client has to know the related binding id or has to call the lookup service to retrieve named method streams as in the example in fig. 6. Assuming that the caller has obtained the binding id from the message stream, it can be used to connect the caller to the method stream. After that, the caller is able to cast the retrieved method stream to any appropriate interface, in the example the interface *ILoginService*.

```
CBindingId bindId = bindMgr.lookup("loginservice");
ILoginService loginSvc =
                 (ILoginService)bindMgr.connect(bindId);
loginSvc.login(szUser, szPassword);
```

Fig. 6. Locating and connecting a method stream on the client side

Apart from automatically creating a proxy object which implements all interfaces of the remote object, the proxy can also be individually instantiated and customized to offer only certain interfaces. In contrast to RMI where the client requires access to the appropriate stub code providing all interfaces of the remote object, the method stream can be still used even after it has been extended with new methods in the meantime. In this sense, the presented approach decouples caller and callee and enables the interconnection of arbitrary objects, given that the callee understands the messages of the caller.

3.5 Performance

Since the illustrated realization in Java is based on Java Dynamic Proxies and Java Reflection, it of course introduces some runtime overhead in contrast to native Java method calls. However, the overhead depends heavily on the used reflection mechanisms and parameters passed to the method. For example, primitive Java

parameter types like *long* or *double* have to be encapsulated within objects before they can be used in Java Dynamic Proxies, but object references can be passed directly. Consequently, we have evaluated our approach in two different method calling scenarios. In the first one, a method *login* is called using object references only, and in the second one a method *sqrt* is defined with long as parameter type and double as return type. The evaluation has been performed by calling each of the methods 100000 times, at first on a local object using a regular method call, then on a remote object using RMI and finally using SOAP. Subsequently, a comparable evaluation has been performed using the presented method streaming approach. The results are shown in fig. 7 with a logarithmically scaled y-axis, grouped by local calls and remote method calls using RMI and SOAP. Each left bar in the group stands for the first method and the right bar symbolizes the second. In addition, the row in front represents the measurements of the native approach and the row in the back the measurements using method streaming.

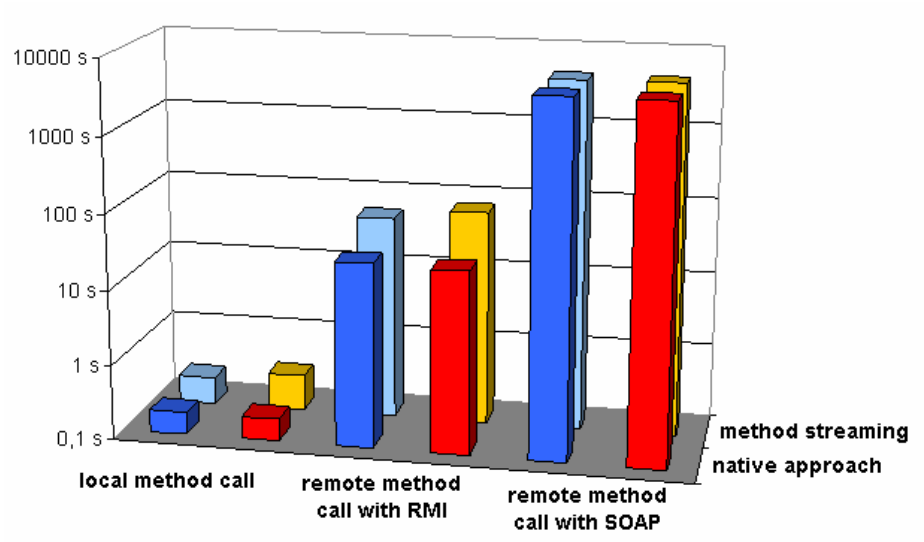

Fig. 7. Performance evaluation of regular local method calls, RMI and SOAP (first row) using object references (left bar per group) and primitive data types (right bar per group) compared to their application with method streaming (second row)

In comparison to the native approaches there is a certain overhead introduced by method streaming, shown as the difference between the first and second row in fig. 7. But compared to a local method call, the additional overhead for a remote method call caused by method streaming is relatively small compared to the original overhead as shown in the diagram between the bars in the first and second row of remote method calls and compared further to the bars of the corresponding local method calls. Surprisingly, due to the binary encapsulation of the method call, method streams using SOAP turn to be even slightly faster than the native SOAP approach.

3.6 Discussion

The concept of method streams aims to separate the *business logic* of a caller from the *binding logic* needed to connect a callee. The presented Java realization therefore uses Java runtime reflection to bind a callee to the method stream and let the caller issue method calls on the callee by connecting it to the related method stream using a Java Dynamic Proxy. Both actions are usually transparently performed and hidden from the developer except for the initial creation of a method stream by binding a callee and connecting a caller, respectively, as illustrated above in section 3.4. Thus, the proposed approach does not require any extra particular implementation effort in the business logic. Subsequent remote object references are automatically bound on the callee side to newly created method streams and dynamically connected on the caller side. Along with that, the approach offers the generic encapsulation of binding aspects within so called *intermediaries*. In this sense, it provides a binding framework which decouples the caller and callee from the involved binding aspects. Hence, binding aspects of existing method streams can be changed on the fly, e.g. switching the underlying networking protocol from RMI to CORBA. Compared to programming language independent solutions like SOAP and CORBA, the presented approach is of course not applicable for interconnecting non-Java objects. On the contrary, it focuses on supporting Java developers to transparently use remote Java objects similar to RMI. However, it requires no adaptation of the object implementation at all and offers the capability to individually adjust binding aspects. Another notable feature of the presented approach is its separation of object localization and object binding, since it introduces an abstract level of object referencing with the mentioned binding id. This way, method streaming decouples the object binding from the currently used lookup service but nevertheless enables the application of widely used services like LDAP [19] or CORBA NamingService [20] by using a localization extension in form of a suitable plugin. As a result, the binding approach is completely independent of the chosen localization model. Finally, according to the performance measurements it is not useful to bind every Java object using the proposed solution. Rather, method streaming should be applied where its major features of transparent remote object binding are needed and where the extra overhead is quite small compared to the already existing binding and network transmission efforts.

4 Application of the Approach

The presented approach has been developed as part of the middleware platform *ODIN* [21], which is currently being used in several ongoing research projects like *netzspannung.org* [22, 23] and *Awake* [24]. The Java implementation of ODIN uses the presented approach and its application is illustrated in the following.

4.1 Customizable Binding Aspects

Distributed applications typically consist of components and objects which are hosted on different platforms. As mentioned above, they interact across network boundaries

using middleware solutions like CORBA or SOAP, and the business logic uses stubs/skeletons which enable the transparent calling of a remote method. Within this context, the presented approach allows to add additional features and binding aspects seamlessly as shown in fig. 8.

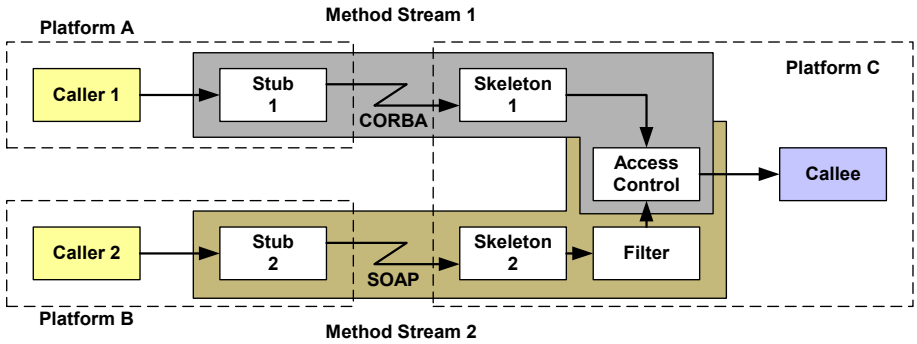

Fig. 8. Customizable binding aspects

The caller 1 on platform A is able to call methods on the callee using the method stream 1. The involved binding aspects to bridge the network using CORBA and to control access on the callee are virtually hidden and do not affect the caller 1 nor the callee. A second caller 2 on platform B is also able to call methods on the callee. But in contrast, it uses SOAP to bridge the network and there is an additional binding aspect filter in its method stream in front of the access control. It is able to suppress some method calls at all, e.g. an external configuration file could specify to reject calls with an exception or discard others without feedback similar to a network packet filter. Thus, method streams open a new field for transparent runtime customization since the communication with a callee can be inspected and modified by the system or configured by the user.

Finally, it should be pointed out again that the presented approach in Java works with dynamic proxies and reflection, hence it relies on very basic language elements and does not need any particular implementation effort. Furthermore, no additional parsing/conversion is required like in XML intermediaries [25], rather each binding aspect works directly with Java objects. They are chained with each other on the Java interface level and can be transparently exchanged, allowing e.g. rebinding the callee with another networking protocol.

4.2 Transparent Object Migration

In addition to be able to bind a single callee instance using various binding aspects concurrently, the presented approach can be also used in different object migration scenarios, e.g. in decentralized object localization environments such as P2P networks or in centrally managed object migration situations as shown in fig. 9.

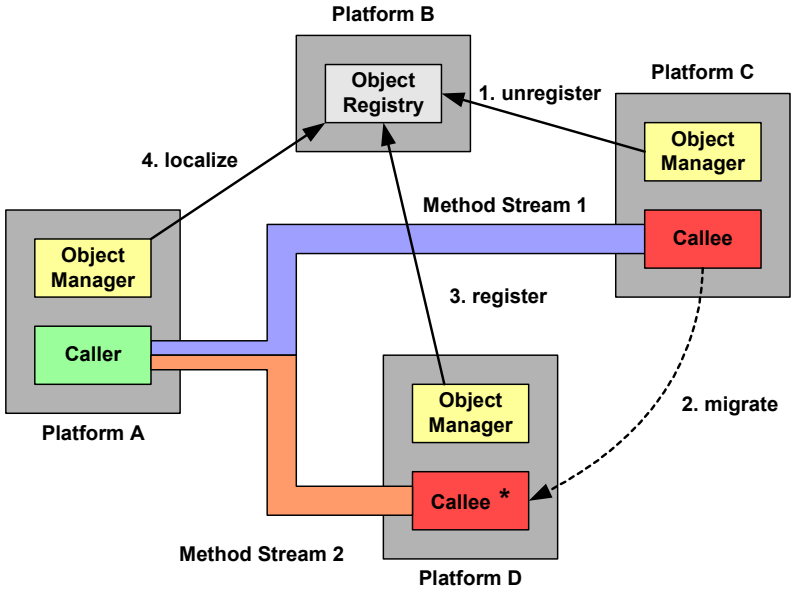

Fig. 9. Transparent object migration

The callee is initially located on platform C and bound by the caller on platform A using method stream 1. After some time, the callee is supposed to migrate from platform C to platform D. First, it prepares itself to migrate and issues an unregister to the central object registry on platform B. Then, it migrates from C to D and registers itself with its binding id to the binding registry. Later, the caller is supposed to issue a method call on the callee but the method stream determines that the callee has been migrated. It asks the binding registry for the new location using the same binding id and rebinds itself to the callee*, establishing method stream 2. This happens transparently to the caller as well as to the callee. It would even be possible that method stream 2 uses another networking protocol than method stream 1. But both method streams still start with the same binding aspect such that the caller is not affected by the migration. The binding id, used to identify a method stream, does not contain any address information, hence it is not limited to a certain network protocol nor it becomes invalid when the related object migrates.

5 Conclusions

In this paper, we have presented a Java-based network middleware approach to binding remote objects based on so called method streams. The major benefits are an easy-to-use Java framework, the seamless integration of legacy Java code and different network middleware approaches, the transparent encapsulation of binding aspects and the separation of binding and localization without the overhead of

conventional intermediary approaches [15]. We have shown that method streams support the decoupling of caller and callee, enabling seamless object migration with invariant object identifications. The realization of method streams in Java has been described using Java Dynamic Proxies and Java reflection. Possible applications of the approach were also discussed for two selected scenarios, highlighting the benefits of method streaming. Finally, it should be mentioned that the presented approach does not want to replace but supplement existing network middleware solutions like CORBA or RMI by providing an overall framework which eases the development of distributed Java applications and the configuration of commonly needed features.

There are several issues for future work. Certainly, the presented Java implementation is much faster than conventional intermediary approaches like XML intermediaries where the passed method call has to be parsed, evaluated and converted again into XML for the next intermediary. However, the usage of Java reflection and Java Dynamic Proxies introduces some overhead which can be reduced using appropriate adapters. The same is valid for the Java Dynamic Proxy when injecting the method call into the stream. Although both could be automatically created using *meta-programming* [26], we currently investigate how this could be managed when objects are dynamically migrating since the specific adapter object would have to migrate as well. Another issue is the dynamic insertion of binding aspects during runtime in contrast to conventional *Aspect-Oriented Programming (AOP)* [14] that is performed during compile time. This opens the interesting question if and how method streams can be used to complement aspect-oriented programming with aspects customized at runtime. Finally, the basic concept of the presented approach is not limited to Java. A comparable realization would be possible with *Microsoft's C#* [27] and *.NET Remoting* [28] where runtime reflection and dynamic proxies, so called *Transparent Proxies* are available even across different programming languages and are already used in a similar way to some extent.

6 Acknowledgements

The presented approach has been used and evaluated in the implementation of the Internet platform *netzspannung.org*. The related project CAT [29] is funded by the German Federal Ministry for Education and Research and is conducted by the research group MARS of the Fraunhofer Institute for Media Communication, Sankt Augustin in cooperation with the University of Siegen and the University of Marburg, Germany.

References

1. Vinoski, S. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. IEEE Communications Magazine. Vol. 35, Nr. 2. IEEE 1997. pp. 46-55.
2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.
3. Grosso, W. Java RMI. O'Reilly & Associates. 2001.

4. Orfali, R., Harkey, D. Client/Server Programming with Java and Corba. John Wiley & Sons, Inc. 1998.
5. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S. Unraveling the Web Services Web. An introduction to SOAP, WSDL, and UDDI. Internet Computing. Vol. 6, Nr. 2. IEEE 2002. pp. 86-93.
6. Guy-Ari, G. Empower RMI with TRMI. JavaWorld. Nr. 9. IDG 2002. http://www.javaworld.com/javaworld/jw-08-2002/jw-0809-trmi_p.html
7. Winer, D., XML-RPC Specification, http://www.xmlrpc.com/spec, UserLand, Burlingame California, October 1999.
8. Deitel, H. M., Deitel, P. J., Gadzik, J. P., Lomeli, K., Santry, S. E., Zhang, S. Java Web Services for Experienced Programmers. Prentice Hall. 2003.
9. Java API for XML-Based RPC (JAX-RPC). http://java.sun.com/xml/jarpc/
10. Aldrich, J., Dooley, J., Mandelsohn, S., Rifkin, A. Providing Easier Access to Remote Objects in Client-Server Systems, Proc. of the Thirty-first Hawaii Int. Conference on System Sciences. Hawaii. IEEE 1998. pp. 366-375.
11. Hanssen, O., Eliassen, F. A Framework for Policy Bindings. Proc. of Intl. Conference on Distributed Objects and Applications (DOA 1999). IEEE 1999. pp. 2-11.
12. Richmond, M., Noble, J. Reflections on Remote Reflection. Proc. of the 24th Australasian Computer Science Conference (ACSC 2001). IEEE 2001. pp. 163-170.
13. Flammia, G. Peer-to-Peer is not for everyone. IEEE Intelligent Systems. Vol. 16, Nr. 3. IEEE 2001. pp. 78-79.
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. Proc. of the European Conference on Object-Oriented Programming (ECOOP). LNCS 1241. Springer-Verlag 1997. pp. 220-247.
15. R. Barrett, P. P. Maglio, Intermediaries: An Approach to Manipulating Information Streams, IBM Systems Journal. Nr. 38. IBM 1999. pp. 629-641.
16. Eckel, B. Thinking in Java. Prentice Hall. 2000.
17. Vinoski, S. Chain of Responsibility. IEEE Internet Computing. Vol. 6, Nr. 6. IEEE 2002. pp. 80-83.
18. Kenens, P., Michiels, S., Matthijs, F., Robben, B., Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P. An AOP Case with Static and Dynamic Aspects. ECOOP'98 Workshop Reader. LNCS 1543. Springer-Verlag 1998. pp. 428-429.
19. W. Yoeng, T. Howes, and S. Kille. Lightweight Directory Access Protocol. RFC 1777, Standards Track, March 1995.
20. Landis, S., Shapiro, W. CORBA Naming-Service Evaluation. IEEE Concurrency. Vol. 7, Nr. 4. IEEE 1999. pp. 44-51.
21. Open Distributed Network Environment. http://odin.informatik.uni-siegen.de
22. netzspannung.org, Communication Platform for Digital Art and Media Culture. http://netzspannung.org
23. Fleischmann, M., Strauss, W., Novak, J., Paal, S., Müller, B., Blome, G., Peranovic, P., Seibert, C. netzspannung.org - An Internet Media Lab for Knowledge Discovery in Mixed Realities. In Proc. of 1st Conference on Artistic, Cultural and Scientific Aspects of Experimental Media Spaces (CAST01). St. Augustin, Germany. 2001. pp. 121-129.
24. AWAKE - Networked Awareness for Knowledge Discovery. Fraunhofer Institute for Media Communication. St. Augustin, Germany. 2002. http://awake.imk.fraunhofer.de
25. Slominski, A. et al. Design of an XML based Interoperable RMI System: SoapRMI C++/Java 1.1. Proc. of the 2001 Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA). Las Vegas, USA. CSREA 2001. pp. 40-46.
26. Baker, J., Hsieh, W. Runtime Aspect Weaving through Metaprogramming. Proc. of the 1st Intl. Conf. on Aspect-Oriented Software Development (AOSD). ACM 2002. pp. 86-95.
27. Farley, J. Microsoft .NET vs. J2EE: How do they stack up. O'Reilly 2001.
28. McLean, S., Williams, K., Naftel, J. Microsoft .NET Remoting. Microsoft Press. 2002.