FORMI: An RMI Extension for Adaptive Applications

Rüdiger Kapitza
Dept. of Comp. Sciences
Informatik 4
University of Erlangen-Nürnberg
Germany
rrkapitz@cs.fau.de

Michael Kirstein
Holger Schmidt
Franz J. Hauck
Distributed Systems Laboratory
University of Ulm
Germany
formi@kirstein-michael.de
{holger.schmidt,franz.hauck}@
uni-ulm.de

ABSTRACT

RMI is a well-known middleware that smoothly integrates into Java. RMI uses classical RPC-based client-server interaction, precisely remote method calls. Although RMI has several extension points (i.e., for replacing transport protocols and call semantics), this is not enough for many applications as it can not cope with non-RPC-based communication, fault tolerance, scalability, and quality-of-service in general. We present FORMI, an RMI extension for supporting the very flexible fragmented-object model. This model allows to build distributed objects with arbitrary internal communication protocols and interaction patterns (e.g., internal peer-to-peer communication) and with a truly distributed internal structure (e.g., replicated servers, smart proxies, hierarchical servers). Both, internal communication and structure, remains hidden behind the RMI-object interface and is thus transparent to clients. We demonstrate our approach by an Internet radio example.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed applications*; D.2.12b [**Software**]: Software Engineering Interoperability[Distributed objects]; D.3.3.h [**Programming Languages**]: Language Constructs and Features—*Distributed objects, components, containers*

General Terms

Design

Keywords

Java RMI, Fragmented Objects, Adaptability

RM'05, November 28– December 2, 2005, Grenoble, France

1. INTRODUCTION

Distributed object-oriented applications are commonly implemented on top of popular middleware platforms like CORBA, .Net-Remoting or Java Remote Method Invocation (RMI). These platforms aim at simplifying the development and the execution of client-server–based applications. Whereas this is sufficient for most distributed applications based on the traditional client-server approach, there is a rising number of applications demanding for fault-tolerance, high availability, shorter response times, and many more so-called non-functional requirements. Fault tolerance and availability have been addressed by introducing the object group paradigm [1]: A server object is replicated among a group of objects that are kept consistent via a group communication protocol. Clients interact transparently with the replicated object using group proxies. Unlike CORBA, which provides a special extension called FT-CORBA [2], this is left open in the Java RMI specification. Instead, the RMI framework provides extension points to implement new call semantics and transport protocols. Various research projects [3, 4] have used these extension points to integrate the object group paradigm into RMI. Fault tolerance and availability are, however, only two of many extended demands of recent distributed applications. Furthermore, the proposed solutions usually introduce a restricted set of protocols and patterns of the possible internal structure. Existing extensions do not provide any way to generically adapt the interaction pattern and the internal structure of distributed objects to the application's needs.

A fragmented-object model as proposed by Shapiro [5] can meet the expected flexibility. It is far more generic and flexible than the traditional client-server approach or the object group paradigm. A fragmented object is a truly distributed object that can be arbitrarily partitioned. Parts of the object—named fragments—may exist on different nodes and provide the object's interface. Unlike RMI and most other middleware systems that use a stub-skeleton–based architecture in combination with an RPC-based protocol, accessing a fragmented object presumes the existence of a local fragment. This can act as a proxy supporting another fragment's functionality, but may also contain local functionality. This principle can increase the application's performance since in some cases no remote call is needed.

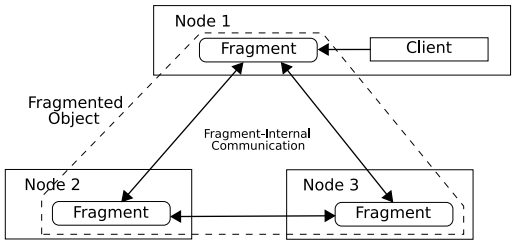

Figure 1: Fragmented object on three nodes

The internal communication among the fragments can use arbitrary patterns and protocols (e.g., real-time transport protocols). The internal structure of the fragmented object may be hierarchical or peer-to-peer or even something else. At the same time a client will just invoke local methods on a local representative as in any other middleware system (on a stub or fragment). Thus, the internal implementation of a fragmented object is truly transparent.

A fragmented-object model has been proposed in previous research projects to implement distributed applications with extended requirements. FOG [6] and Globe [7], however, provide this model within a proprietary middleware that cannot interoperate with popular middleware platforms like CORBA or RMI. Based on our former work [8] that uses a fragmented-object model within a CORBA environment this paper presents an approach to seamlessly integrate the fragmented-object model into the RMI framework. In contrast to our former work, this framework does not depend on an extended ORB implementation but provides fragmented objects in any Java 1.2-compliant run-time environment.

This paper is structured as follows: Sections 2 and 3 give a brief introduction to the fragmented-object model and present the RMI architecture. The integration of the fragmented-object model into RMI is the subject of Section 4. Section 5 presents an example application, a fragmented object for an Internet radio. Section 6 compares our work with related approaches. Finally Section 7 concludes the paper.

2. THE FRAGMENTED-OBJECT APPROACH

Fragmented-object models [5, 6] extend the traditional concept of stub-based distributed objects. Hence, an object, which has a unique object identity, can be distributed among different nodes. On these nodes a local fragment is created that belongs to the fragmented object. For binding to a fragmented object, a local fragment has also to be created, if not yet available, that acts as a (proxy-)fragment offering the whole object's functionality. Similar to a stub in classic RPC-based systems, the fragment acts as a gateway or proxy to the real object. Like stubs, fragments provide the same interface as the distributed object they belong to. In principle, an implementation can be designed in such a way that clients cannot distinguish between the access of a local object, a local stub or a local fragment. Thus distribution transparency can be maintained and it can also be transparent how distributed objects are accessed.

Figure 1 shows a fragmented object placed on three nodes. On Node 1 a client has bound to the object. The local fragment may have been created just for the binding purpose, whereas the other fragments may be placed at creation time of the object. The fragments can communicate with one another and there is no restriction about communication patterns and protocols. Fragments on Node 1 and 3 may act as stubs contacting the server fragment on Node 2. In another scenario, the fragment on Node 1 may act as a smart proxy, similar to the smart proxies in [9, 10]. Those can support caching mechanisms to reduce communication, or they may send method invocations to a group of replicas (e.g., on Nodes 2 and 3 there could be fragments replicating the state of the distributed object) in order to balance load or mask faults. As another alternative the fragments may communicate by peer-to-peer or real-time protocols (cf. Section 5).

The distribution of state and functionality over the fragments has to be done by the object developer by designing different fragment implementations. Thus, a fragmented object can provide fragments for replication and partitioning of state. Fragments may even be organized in a hierarchy to support scalability of large-scaled application objects. The internal structure and communication remains completely transparent to the client and allows even dynamic changes inside the fragmented object. The object may decide to introduce replicas or to migrate state and/or functionality whenever appropriate. Thus, a fragmented-object model supports static and dynamic adaptable applications. For supporting dynamic adaptability, a local fragment implementation should be replaceable by another version or variant without the clients noticing any difference.

A middleware system supporting the fragmented-object model has to implement ways to pass references to fragmented objects, and to create local fragments on the receiver side (e.g., when receiving a passed reference as a parameter). Unlike a classic RPC-based system that creates a local stub, this is more complex. The local fragment is object specific, i.e., depending on the object instance a specific fragment implementation has to be created that was configured by the object developer. For supporting dynamic adaptability the concrete fragment implementation may also depend on local properties (e.g., load and available resources) which makes the selection of such an implementation more complex.

3. JAVA RMI ARCHITECTURE

The Java Remote Method Invocation (RMI) is a Java Standard that allows users to call methods on objects located in another Java Virtual Machine (JVM). This JVM might even be located on a remote node. Java RMI aims at maintaining the semantics of the Java object model in a distributed environment including, e.g., distributed garbage collection. In this section we give a brief overview of the RMI architecture. Furthermore, we discuss the RMI architecture and extension points with respect to the seamless integration of a fragmented-object model.

In RMI there are different semantics for passing an object to a remote method. A primitive value like, e.g., an integer is transferred using a *call-by-value* semantics. Java objects as well as not exported RMI-objects to a remote method are passed using a *call-by-copy* semantics. Java RMI uses Java Serialization during the marshalling process. Hence, objects being passed using call-by-copy semantics have to be serializable. It is possible to export and to un-export RMI objects using the `UnicastRemoteObject` class. Exported objects are remotely accessible and are passed using a *call-by-reference* semantics.

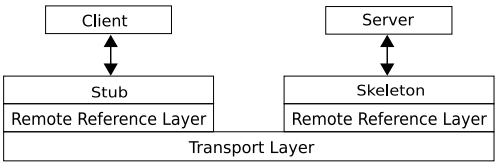

Figure 2: Java-RMI Architecture

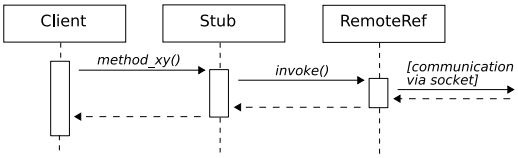

Figure 3: Java-RMI Invocation Path on Client Side

Figure 2 shows the Java-RMI architecture, which contains three layers. the first layer is the *stub and skeleton layer*. In RMI, the standard mechanism of RPC-based systems is used: Stubs and skeletons are generated as an interface for the client and the server to the middleware. Thus, these helper objects enable distribution-transparent invocations on remote objects. The stub and the skeleton implement a remote object reference, and take care about marshalling and de-marshalling. They are generated from the object's remote interface that has to be specified by the application programmer. In RMI, the stub on the client side has to be generated manually using the RMI compiler *rmic*[1] whereas the skeleton is always automatically created at run-time.

The common super type of all stub classes is `RemoteStub`. `RemoteStub` inherits `RemoteObject` and defines no further methods beside the constructor. For concrete distributed objects, *rmic* creates a stub, which also contains the remote object's methods. `RemoteObject` inherits `java.lang.Object`. Final methods of `java.lang.Object` cannot be overwritten. These include the objects synchronization methods (`wait()`, `notify()`, `notifyAll()`) as well as the `getClass()` method that returns the object's class. Thus, these methods relate to the stub object and not to the remote RMI object: The `getClass()` method returns the stub class instead of the class of the remote object. The synchronization methods use the local stub object for coordination. These final methods cause language-dependent limitations that reduce the access transparency of RMI objects, but this would apply to any other RPC-based interaction mechanisms that is integrated into Java.

Some of the non-final methods of `RemoteObject` are overwritten in a concrete stub: `equals()`, `hashCode()` and `toString()` are adapted to the semantics of a distributed object. The `RemoteObject` class contains a reference to a `RemoteRef` object, which represents a handle to the remote object.

The *remote-reference layer* specifies the call semantics of an RMI object. Therefore, a `RemoteRef` interface is defined in the RMI specification. Objects implementing remote call semantics have to implement this interface. The stub is using the `invoke()` method of the actual `RemoteRef` object in order to call remote methods. A reference to a `RemoteRef` object is stored in every stub.

The RMI implementation from Sun offers standard `RemoteRef` objects for several call semantics, e.g., for activatable remote objects. Nevertheless, it is possible to implement own call semantics by creating another `RemoteRef` instance. The client does not have to be changed for this purpose; the call semantics are transparent to the client.

The *transport layer* uses Java socket classes in order to handle the communication between the different participating JVMs. Due to the socket-factory concept introduced in Java 1.2, the RMI system is able to use any stream-based communication.

Since Java 1.3, RMI offers the possibility to communicate using the Internet Inter ORB Protocol (IIOP). This allows access to RMI objects from CORBA environments. A so-called Java-to-IDL mapping takes care of the conversion of a RMI interface into an IDL interface.

Fig. 3 shows the invocation path through all layers of RMI on the client side. The stub marshals the parameters and calls a generic `invoke` method at the remote-reference object, which will implement the call semantics using an RPC protocol on top of UDP or TCP sockets.

4. FRAGMENTED OBJECTS IN RMI

Our goal is to integrate a fragmented-object model into RMI such that clients cannot see any difference to standard RMI objects. This way, even already existing applications could have access to fragmented RMI objects and thus benefit from a fragmented-object model (e.g., have access to a fault-tolerant service without knowing that parts of the fault-tolerance mechanism will be locally implemented in a special proxy fragment). This means that references to fragmented RMI objects should look like references to standard RMI objects, and should be serializable by standard RMI marshalling operations.

In this section we first discuss several implementation alternatives based on the extension points of RMI. Then we present the design of local fragments within our approach.

4.1 Implementation Alternatives

As already described in Section 3, the RMI architecture is divided into three layers: the stub-skeleton layer, the reference layer and the transport layer. Whereas the first two layers are part of the specification the third layer is from an architectural point of view completely implementation specific. For this reason and for the fact that the remote and all internal communication is object specific and therefore implementation dependent we will inspect how the fragmented-object model can be integrated or even completely replace the stub-skeleton layer or the remote reference layer.

4.1.1 Extending the Remote Reference Layer

The remote reference layer can be extended as designated by Sun to provide new remote-invocation semantics. This has already been done to support on-demand activation of objects by Sun itself and to provide replicated objects in the JGroup system [3] and by Cazzola et al. [4]. Fragmented objects could be easily integrated the same way: The rmic-generated stubs remain the same and only the `RemoteRef` object has to be extended. It needs to refer to the local fragment, which has to implement the remote interface of

[1]With Java 1.5, stubs can also be generated automatically at run-time.

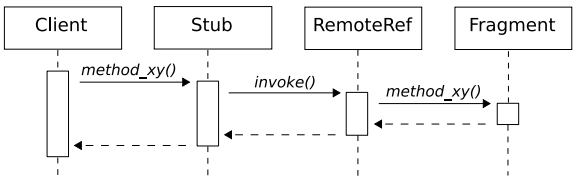

Figure 4: Invocation path with extended reference layer

our FORMI object. The `RemoteRef` object forwards incoming invocations to the local fragment.

As marshalling of the reference to a FORMI object (i.e., the reference to the stub) will serialize the stub and its adjunct objects, it will also marshal the fragment. This is not appropriate as every client may get its own individual fragment, perhaps even with entirely different implementation. Therefore, the fragment reference needs to be transient and a fragment-implementation factory has to be introduced that is marshalled and takes care of creating the fragment locally depending on whatever decisions. The fragment-implementation factory may also take care of reusing already existing local fragments.

Although this approach is very lightweight and easy to implement it has one major weakness. Each call of the local fragment instance is treated like a remote method invocation. The parameters are already marshalled in the stub, and the `RemoteRef` object is invoked (see Fig. 4 and cf. Fig. 3). This has to de-marshal the parameters in order to invoke the corresponding method on a local fragment. Marshalling partially involves Java serialization and reflection. Both are expensive operations and thus inefficient.

4.1.2 Extending the Stub-Skeleton Layer

For the elimination of inefficient serialization and reflection operations we could replace the stub and skeleton for a fragmented object. In the standard implementation of RMI, the `RemoteRef` class is responsible for setting up the communication based on its encapsulated contact information, for forwarding the incoming call to the remote object, and for implementing the call semantics. In context of fragmented objects this is actually not necessary since a fragment instance itself is responsible for setting up and managing the object's internal communication. From a conceptual point of view, a `RemoteRef` instance is not essentially needed.

Following this idea, custom stubs can be generated that take the role of the fragment. Standard stubs inherit from class `RemoteStub` which in turn inherits `RemoteObject`. As `RemoteStub` provides some static methods (e.g., `toStub()`) it is preferable that custom stubs inherit those classes as well to maintain compatibility.

A severe problem of this approach is that the reference layer is not existent. When trying to marshal such a stub for transferring an RMI reference a `MarshalException` will occure because the RMI serialization mechanism cannot handle a `null` value for a `RemoteRef` reference. Solutions for this issue might be the usage of a non-empty dummy object representing the `RemoteRef` or the usage of `writeReplace()` and `readReplace()` methods for supporting the marshalling process. We propose a solution that solves all the mentioned problems in a more elegant way in the next sub-section.

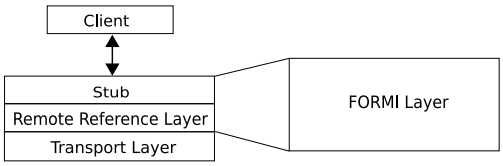

Figure 5: FORMI-stub replacing the Stub and the Remote Reference

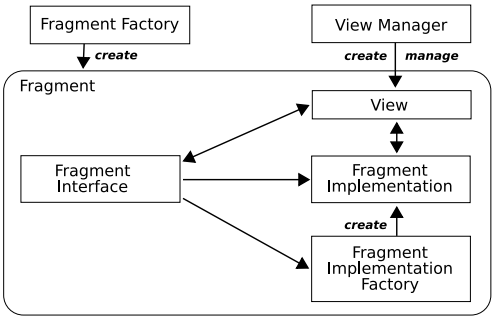

Figure 6: Architecture of a Fragment including the involved helper objects

4.1.3 Combined Approach

Regarding the previously presented solutions we propose a combined approach. A special stub—named *FORMI-stub*—where the remote reference layer and the stub-skeleton layer are merged (see Fig. 5). More precisely, this means that our stub extends `RemoteStub` and implements the `RemoteRef` interface at the same time.

This approach has some advantages: First of all, marshalling the stub is possible without any efforts since the reference to a `RemoteRef` object can be a self reference and thus is not null. Second, calls are processed faster because there is no need for using reflection and serialization (see Section 4.1.1).

4.2 The Structure of Local Fragments

As seen from the previous sub-section, we can implement our fragments by inheriting from `RemoteStub`. This has the drawback that the fragment implementation cannot be exchanged at run-time as clients may have references to that fragment that cannot be redirected. Thus, we introduce one level of indirection to the fragment structure as shown in Fig. 6. A so-called *fragment interface* is used as a reference to the local fragment. The fragment interface inherits `RemoteStub` as described before and offers the complete set of methods of the fragmented object (cf. Section 3). It forwards calls to a *fragment implementation* which car-

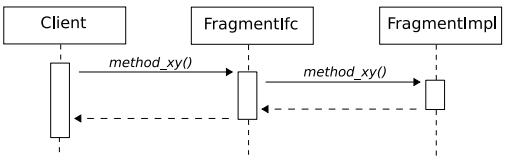

Figure 7: FORMI invocation path on client side

ries the actual fragment functionality. The fragment implementation has to implement the remote interface as any RMI object. This mechanism enables dynamic, transparent exchange of the current fragment implementation at runtime. Another fragment implementation has to be created and the corresponding reference in the fragment interface has to be updated. The last internal component, the *view*, manages the fragment interfaces, the fragment implementation and its exchange, and stores internal data like, e.g., the object id. Furthermore, it provides an interface for quality-of-service requirements related to the local fragment based on so-called aspect configurations (cf. [11]). The view is, in conjunction with a *view manager*, also used to detect locally existing fragments for reuse when receiving remote object references. The invocation path in FORMI is shown by Fig. 7 (cf. Fig. 3).

For supporting the application programmer we provide a stub generator equivalent to Sun's rmic[2]. Therefore, creating fragment interfaces in our environment is more or less the same for an application programmer using RMI. For client programmers this will be transparent as they will dynamically load the class code via the RMI codebase.

4.3 Creating Objects and Fragments

For the creation of a new fragmented object a fragment-implementation factory has to be created. This is responsible to select the class of the local fragment implementation. The seletion process may even depend on local properties or on the distributed state of the fragmented object. An instance of a new FORMI object is created by calling the FORMI fragment factory and passing a reference to the object-specific fragment-implementation factory (cf. Fig. 6). FORMI will build a new fragmented object and create an initial fragment by calling the fragment factory. As a result the application will get an RMI reference to the new object in form of a Java-object reference to the fragment-interface object.

When passing a reference referring to a FORMI object to another RMI object, the fragment interface and the fragment-implementation factory are marshalled and transmitted. The view and the fragment implementation have to be marked `transient` because these two components of a local fragment represent the local only parts. After de-marshalling just the fragment interface and a reference to the fragment-implementation factory are existent. The view and the appropriate fragment implementation are not created until the first call to the fragment is received at the fragment interface. This approach optimizes performance, e.g., for the case of registering the object with an RMI registry. The registry will not use the object and thus does not need a fragment implementation.

5. EXAMPLE APPLICATION

In order to demonstrate the usability of our approach we implemented a fragmented Internet radio. An Internet radio is a service that broadcasts audio streams to clients. The service should be implemented using RMI. That means, the Internet radio should be registered in a naming service (e.g., RMI registry) and accessible by RMI clients like any other RMI object. The client accesses the radio by a lean interface (`RadioInterface`):

- `getAudioStream()` (retrieves a stream delivering the audio data of the radio service)
- `getFormat()` (returns audio format)
- `send()` (sends file to clients)

With standard RMI a client would only be able to interact with the service by RPC-based method invocations. This would not be appropriate for delivering audio data. With the fragmented-object model the radio service is implemented by different fragments, a server fragment that broadcasts audio data to a IP multicast group, and a client fragment that receives those data (see Fig. 8).

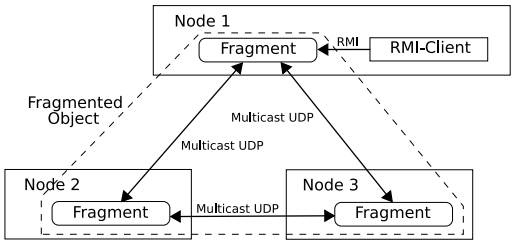

Figure 8: Fragmented Audio Service using IP multicast for internal communication

Based on this interface, the fragmented audio service was created. First, the fragmented object is created by creating an initial fragment. FORMI returns a reference to the first fragment interface. This supports the `RadioInterface`. It is registered at an RMI registry using `bind()`. After this process the `send()` method is called in order to start sending the audio data. The communication address for inter-fragment communication is stored in the fragment-implementation factory which is referenced by each fragment interface and transferred to each node of the distributed system. The factory can pass this information to every new fragment implementation.

A client retrieves the reference to the radio service from the RMI-registry using the `lookup()` method. As usual, the returned reference (to the fragment interface) is casted to the `RadioInterface`. With the first invocation the fragment implementation is automatically created. The implementation will act here like a smart proxy: it will open the multicast socket and receive audio data, which will be available to clients as an audio stream.

With the local fragment implementation, there is implicitly appropriate code available for interacting with the radio server. This code is always specific to the particular distributed object. The service can use non-RPC-based protocols and even maintain quality-of-service properties (e.g., resource reservation on the network[3]).

6. RELATED WORK

Shapiro [5] introduced the concept of fragmented objects used in the FOG project [6]. The concept was considered

[2]Similar to Java 1.5, our stubs, aka fragment interfaces, could be generated automatically at run-time. This, however, would introduce additional run-time costs due to the calls to the reflection API that are used within these stubs. We plan to introduce this as an optional feature in the future, similar to Java 1.5.

[3]Reservation is not part of our prototype implementation.

especially useful for designing distributed applications. The FOG project focused on creating tools for supporting users in creating fragmented objects. The concept of fragmented objects was also subject of the Globe project [7] for supporting scalability by caching and replication. Unlike Globe, we support implicit binding as it is used in most object-based systems: A local fragment is automatically created when a fragment reference is passed through the marshalling process.

The concept of smart proxies [9, 10] has benefits similar to our approach (e.g., regarding caching and support of replicate groups). Nevertheless, smart proxies represent the traditional client-server concept whereas fragmented objects offer a more powerful and flexible approach; special fragments can even act like a smart proxy. As the implementation effort of using fragmented objects is comparable to the usage of smart proxies, we preferred the fragmented-object approach.

For supporting fault tolerance the Jgroup project [3] integrated the group communication paradigm supporting one-to-many semantics into Java RMI. A replicated object's method is successively invoked on each replicate until a successful invocation returns the result. The group communication was integrated by providing a special `RemoteRef` implementation (see Section 3). Cazzola et al. offered a similar group communication framework in Java RMI by enhancing java to support object groups [4]. Besides offering a special `RemoteRef`-implementation (one-to-many semantics) they also changed the stub layer (see Section 3). When invoking an operation on a replicated object the client receives a result array containing the invocation-result of every replicate. In contrast to Jgroup and Cazzola et al. our approach enables any part of the distributed objects being locally available. Furthermore an arbitrary communication mechanism is possible and the distribution might even change dynamically at run-time in our solution.

7. CONCLUSIONS

In this paper we presented a novel approach of integrating the concept of an fragmented-object model into Java RMI. Our FORMI architecture supports a more powerful and more flexible infrastructure for distributed applications: local fragments can act as smart proxies, can have dynamic distribution of state and functionality, and can be implemented with arbitrary internal communication and structure. We still preserve compatibility with standard RMI-clients; any RMI-capable client is able to use FORMI objects without even noticing.

According to our former work of integrating fragmented objects into CORBA within the AspectIX project, the integration of fragmented objects into Java RMI is even simpler for the application programmer. For this purpose we provide a Java archive (JAR) containing the needed functionality for easy integration of FORMI in foreign projects.

8. REFERENCES

[1] K. P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, 1993.

[2] Object Management Group (OMG). *Common Object Request Broker Architecture*. Object Management Group (OMG), 3.0.3 edition, March 2004.

[3] A. Montresor. The Jgroup Reliable Distributed Object Model. In *Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Systems (DAIS '99)*, Helsinki, June 1999.

[4] W. Cazzola, M. Ancona, F. Canepa, M. Mancini, and Vanja Siccardi. Enhancing Java to Support Object Groups. In *Proceedings of Recent Object-Oriented Trends (ROOTS'02)*, Bergen Norway, April 2002.

[5] M. Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the 6th Int. Conf. on Distributed Systems (ICDCS)*, pages 198–204, Cambridge MA (USA), May 1986.

[6] M. Makpangou, Y. Gourhant, J.-P. Narzul, and M. Shapiro. *Fragmented objects for distributed abstractions*, pages 170–186. IEEE Computer Society Press, 1994.

[7] P. Homburg, L. van Doorn, M. van Steen, A.S. Tanenbaum, and W. de Jonge. An Object Model for Flexible Distributed Systems. In *Proceedings of the 1st Annual ASCI Conference*, pages 69–78, Heijen, Netherlands, May 1995.

[8] H. P. Reiser, F. J. Hauck, R. Kapitza, and A. I. Schmied. Integrating fragmented objects into a CORBA environment. In *Proceedings of the Net.ObjectDays (Erfurt, Germany, Sep. 22-24, 2003)*, 2003.

[9] R. Koster and T. Kramp. Structuring QoS-supporting services with smart proxies. In *Proceedings of the IFIP/ACM Middleware Conference (Middleware)*, volume 1795, Berlin, Heidelberg, New York, Tokyo, 2000. Springer-Verlag.

[10] N. Santos, P. Marques, and L. Silva. A Framework for Smart Proxies and Interceptors in RMI. In *ISCA 15th International Conference on Parallel and Distributed Computing Systems*, Louisville, Kentucky, USA, September 2002.

[11] F. J. Hauck, E. Meier, U. Becker, M. Geier, U. Rastofer, and M. Steckermeier. A middleware architecture for scalable, QoS-aware and self-organizing global services. In *Proc. of the 3rd IFIP/GI Int. Conf. on Trends towards a Universal Service Market - USM (Munich, Sep. 12-14,2000)*, number LNCS 1890, pages 214–229. Springer, 2000.