



IEEE DS ONLINE EXCLUSIVE CONTENT

Object Modeling**FORMI: Integrating Adaptive Fragmented Objects into Java RMI**

Rüdiger Kapitza • University of Erlangen-Nürnberg

Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Holger Schmidt • Ulm University

FORMI integrates fragmented objects into Java RMI without requiring internal modifications to the RMI runtime. FORMI objects can dynamically adapt themselves by replacing existing fragments.

Distributed object-oriented applications are commonly implemented atop middleware platforms such as CORBA, .NET Remoting, and Java remote method invocation (RMI). These platforms provide a simple mechanism to invoke methods of remote objects. Increasingly more applications are demanding nonfunctional properties such as fault tolerance, high availability, and adaptivity, which require extensions to distributed objects' basic interaction model.

A fragmented-object model, such as the one Marc Shapiro proposed,¹ can provide the required flexibility. It's far more generic and flexible than the traditional client-server approach. A fragmented object is a truly distributed object; it consists of multiple fragments located on multiple nodes. Such a model allows arbitrary partitioning of state and functionality on these fragments, and arbitrary internal interaction between fragments of a single object. We've investigated integrating a fragmented-object model into CORBA (AspectIX),² which requires internal modifications to the CORBA object request broker. Our approach for transparently integrating fault-tolerant objects into .NET Remoting is also useful for seamlessly integrating fragmented objects.³ Our FORMI architecture integrates fragmented objects into Java RMI without requiring internal modifications to the RMI runtime.⁴ (See the "Related Work" sidebar for a brief description of other fragmented-object systems and similar middleware concepts.)

The fragmented-object approach

A fragmented-object model extends the traditional concept of stub-based distributed objects.^{1,5} A fragmented object is an entity with a unique object identity. It consists of a set of fragments that an object developer can distribute among different nodes. A client that wants to access a fragmented object needs a local fragment of that object. If no such fragment exists at the client's node, the infrastructure instantiates an appropriate fragment—for example, a proxy to the fragmented object. Like stubs, fragments provide the same interface as the fragmented object to which they belong. In principle, it's possible to design an implementation such that clients can't distinguish between the access of a local object, a local stub, or a local fragment. Thus, the distribution of fragments and the remote interaction between fragments is fully transparent to clients.

Figure 1 shows a fragmented object placed at three nodes. At node 1, a client has bound the object (that is, mapped it to local code). The infrastructure might have created the local fragment just because the local client has bound the object, whereas the developer might have placed the other fragments when creating the fragmented object. The fragments can communicate with one another using arbitrary communication patterns and protocols. Fragments at nodes 1 and 3 might act as stubs contacting the server fragment at node 2. In another scenario, the fragment at node 1 might act as a smart proxy that supports caching to reduce communication. On the other hand, it might send method invocations to a group of replicas to balance load or mask faults. For example, nodes 2 and 3 might include fragments replicating the distributed object's state. Alternatively, the fragments could communicate through either peer-to-peer or real-time protocols.

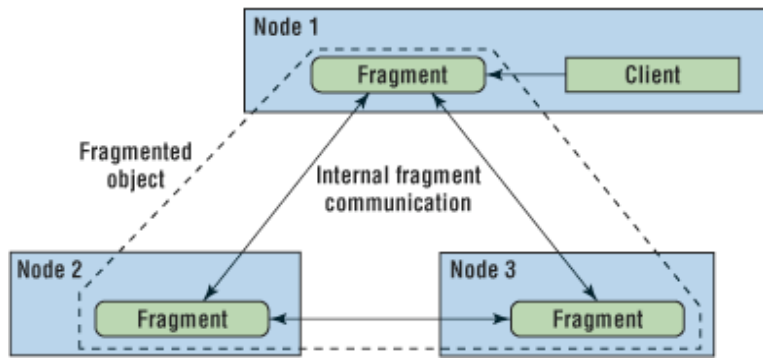


Figure 1. A client at node 1 accesses a fragmented object distributed over three nodes.

An object developer can distribute state and functionality over the fragments by designing different fragment implementations. Thus, a fragmented object can provide fragments for state replication and partitioning. It's even possible to organize fragments in a hierarchy to support scalability of large application objects. The internal structure and communication remain completely transparent to the client, allowing even dynamic changes inside the fragmented object. The object can introduce replicas, as appropriate. It can also migrate state or functionality. Thus, a fragmented-object model supports statically and dynamically adaptable applications. To support dynamic adaptability, a local fragment implementation should be replaceable by another version or variant without the clients noticing.

To maintain distribution transparency, references to fragmented objects should be the same as references to the middleware's traditional objects. Clients should deal only with interfaces. A middleware system offers transparent support for the fragmented-object model only if it implements a way to pass references to fragmented objects and to create local fragments when receiving such a reference as a remote invocation parameter. This process is more complex than the instantiation of a simple stub in a classic Remote Procedure Call (RPC) system. A local fragment is object specific. For each object instance, the object developer can decide which specific fragment implementation the infrastructure must instantiate. To support dynamic adaptability, the concrete fragment implementation might also depend on local properties, such as load and available resources, making selecting such an implementation even more complex.

Fragmented objects can support multiple fragment implementations that serve different purposes—for example, ordinary stub fragments or fragments that serve as caching proxies. The choice of which fragment implementation to create at binding time could depend on arbitrary parameters. For example, the infrastructure will install, depending on the developer's configuration, the caching proxy only if the client side has enough memory or the right credentials. The middleware should provide mechanisms to locally help choose the appropriate fragment implementation. Because it isn't possible to determine at compile time which fragmented objects will be bound and which fragment implementation these objects will choose, the necessary fragment code might not be locally available. Thus, the fragmented-object approach can benefit from mechanisms that dynamically load code.

Java RMI architecture

Java RMI lets applications call methods on objects located at another Java Virtual Machine (JVM). Java RMI maintains the Java object model's semantics in a distributed environment that includes, for instance, distributed garbage collection. Java RMI supports several semantics for passing parameters to a remote method. To transfer a primitive value, such as an integer, Java RMI uses *call-by-value* semantics. Object developers can mark an object as *remote* and export them at the Java runtime so that they'll be remotely accessible. Java RMI passes such exported remote objects using *call-by-reference* semantics. To pass remote objects that aren't exported, as well as all other Java objects, Java RMI employs *call-by-copy* semantics, which uses Java serialization to marshal the object.

Figure 2 shows the layered RMI architecture. On top, the *stub and skeleton layer* contains generated code that enables distribution-transparent invocations on remote objects. The stub and the skeleton implement a remote object reference, and they manage marshalling and unmarshalling. The RMI compiler generates them from the object's remote interface, which the application programmer must specify.

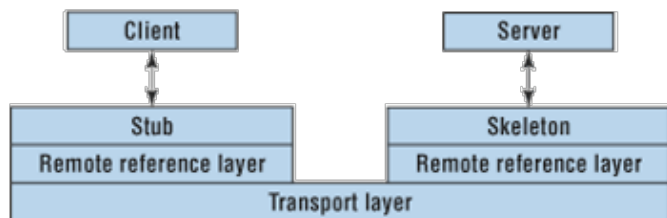


Figure 2. The different layers of the Java remote method invocation (RMI) architecture.

The *remote-reference layer* defines a `RemoteRef` interface. An implementation of this interface represents the link to the remote object and specifies the remote interaction's *call* semantics. Sun's standard `RemoteRef` implementation supports a unicast connection to a previously instantiated and exported remote object. But you can also employ custom mechanisms as Sun did with activatable objects—for example, multicast communication—by creating your own `RemoteRef` implementation. The stub stores a reference to a `RemoteRef` implementation and uses that object's `invoke()` method to call remote methods. Changing the call semantics by using a nonstandard `RemoteRef` implementation is fully transparent to the client.

The *transport layer* uses Java socket classes to handle communication between separate JVMs. Thanks to the socket-factory concept introduced in Java 1.2, the RMI system can use any stream-based communication.

Fragmented objects in Java RMI

Our objective is to seamlessly integrate a fragmented-object model into Java RMI. Clients shouldn't see any difference between a standard RMI object and a fragmented object, which we call a *FORMI object*. Thus, even existing applications will be able to access FORMI objects and benefit from a fragmented-object model. (For example, they could access a fault-tolerant service without knowing that parts of the fault tolerance mechanism are locally implemented in a special proxy fragment.) This means references to FORMI objects should look like references to standard RMI objects, and standard RMI marshalling operations should serialize these references.

Structure of fragments

Each FORMI object fragment internally consists of multiple components, as figure 3 shows. A *fragment interface* is a local object that the client uses as a reference to the local fragment. This interface forwards all methods of a fragmented object to a *fragment implementation*, which provides the fragment functionality. The fragment implementation must implement the remote interface like any RMI object. The client can't directly access the fragment implementation, so the fragmented object can decide to dynamically and transparently replace the fragment implementation at runtime by updating the reference in the fragment interface.

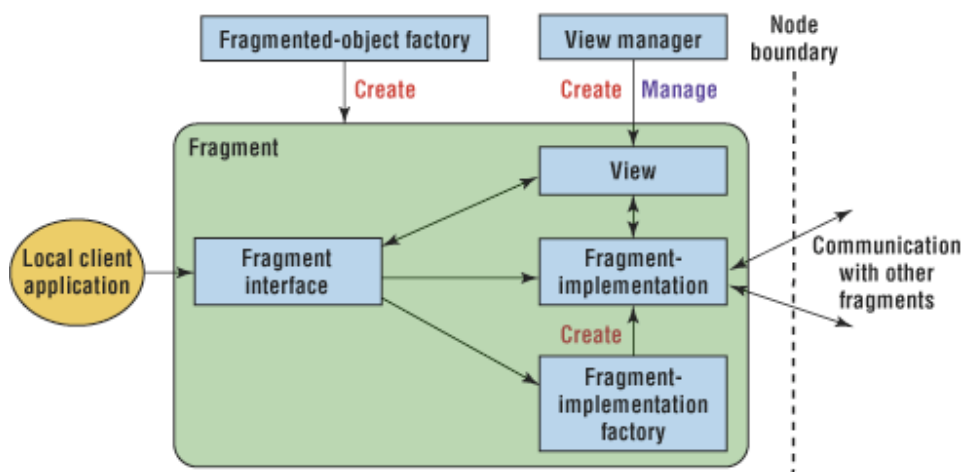


Figure 3. Internal architecture of a local fragment.

The generic component that we call *view* stores internal data such as the object ID and coordinates the references between fragment interfaces and the fragment implementation. This component also provides an interface for quality-of-service requirements based on *policy configurations*; such configurations are used for dynamic reconfiguration. The view, in

conjunction with a *view manager*, can help detect a locally existing fragment implementation for reuse when a client binds a remote object reference. The view manager is a singleton object that FORMI automatically creates the first time it needs it.

Finally, the *fragment-implementation factory* is responsible for instantiating an initial fragment implementation. It encapsulates the strategy for selecting the specific fragment implementation, and it can contain object-specific contact addresses of other fragments.

For the integration into Java RMI, FORMI changes the RMI architecture's stub-skeleton and reference layers.⁴ The fragment interface inherits the `RemoteStub` class, which also serves as a base class for ordinary RMI stubs. Thus, the client sees an interface equivalent to that of a stub.

So that clients can access the RMI reference layer directly even for fragmented objects, the fragment interface also implements the `RemoteRef` interface. If a client directly calls the reference layer's `invoke()` method, the `RemoteRef` implementation unmarshalls the call in the fragment interface and forwards it to the local fragment implementation like any direct method invocation. To avoid unnecessary marshalling and unmarshalling operations, FORMI doesn't use the reference layer for direct invocations.

The fragment architecture introduces no additional level of indirection compared to plain Java RMI. Given a FORMI object, the client invokes a method at the fragment interface, which passes this method to the fragment implementation. In plain RMI, the client calls a method at the stub, which in turn calls a generic invoke method at the remote reference layer (see figure 2).

To automatically create code for the fragment interface, we provide a code generator similar to Sun's *rmic*. We could also have used a generic reflection-based fragment interface or dynamically generated our fragment interface at runtime, as in standard RMI mechanisms in recent Java versions. However, we didn't implement these approaches in our prototype because a reflection-based approach causes significant runtime costs per invocation and dynamic code generation at runtime heavily increases the cost of binding operations.

For client applications, the difference between a FORMI fragment interface and an RMI stub is fully transparent. In both cases, the system loads the required class code from the RMI code base.⁷

Creating objects and fragments

The FORMI infrastructure provides a generic *fragmented-object factory* to create new FORMI objects. An application passes the object type and an instance-specific fragment-implementation factory to the fragmented-object factory (see figure 3). The fragmented-object factory creates a fragment interface and the view. Then the factory calls the fragment-implementation factory to instantiate the initial fragment implementation, and thus builds a new fragment. As a result, the application gets an RMI reference to the new object in the form of a Java object reference to the fragment interface object. Unlike standard RMI, which uses an exported remote-object implementation, all FORMI fragments look like RMI stubs.

Creating a new fragmented object requires a fragment-implementation factory (figure 3), which selects the local fragment implementation's specific class. The selection process can use local system properties or consider the fragmented object's state. A fragmented-object developer can provide arbitrary selection strategies in the factory implementation. A simple factory might statically select a single implementation class. Such a strategy is appropriate for fragmented objects with peer-to-peer interaction between fragments.

In an application with a client-server structure, the factory can also distinguish between client and server fragments. Upon creation of a new FORMI object, the local fragment-implementation factory doesn't find any other fragment. In this case, the factory creates a server fragment and internally stores its contact address, which other fragments can later use to access the server. You can make the FORMI object public by registering it in a naming service. If another node binds the same FORMI object, it receives an RMI reference to that FORMI object, which contains a copy of the fragment-implementation factory. Thus, this node will use a factory instance that knows the server address and will instantiate a client-side fragment implementation initialized with the server address.

More complex factories can choose between different fragment implementations according to local environment conditions, such as resource availability, domain trust level, application, and users. The fragment-implementation factory can also use an external location service to retrieve other fragments' communication addresses and to examine how many other

fragments are available for a certain fragmented object. For example, this factory can deploy another replica fragment, if fault tolerance is at risk, owing to an insufficient number of replicas.

Passing references to fragmented objects

Because the fragment interface looks like an ordinary RMI stub, the RMI runtime will recognize it as such. Passing a reference to a fragmented object through an RMI will lead to exactly the same handling as for standard RMI stubs. The RMI runtime system serializes the RMI stub at the source and deserializes it as a complete copy at the receiver side.

RMI marshalls and transmits the fragment interface and the fragment-implementation factory. But we internally marked the view and the fragment implementation as `transient` because these two components of a local fragment represent local-only parts. After demarshalling, only the fragment interface and the fragment-implementation factory remain. FORMI doesn't create the missing view or the fragment implementation before the fragment interface receives the first call to the fragment. This approach differs from standard RMI and separates addressing and accessing a remote object, similar to the approach in Jonathan.⁸ For example, the RMI registry must reference a remote object, but it doesn't need to instantiate a local fragment implementation. Hence, our approach optimizes performance when a fragment implementation isn't necessary.

Upon a method's first local invocation, the fragment interface contacts the view manager to create a view object for the fragmented object. The view manager maintains a mapping between object IDs and locally available view objects. The view manager uses the object ID, which resides in the deserialized fragment-implementation factory, to detect if another fragment of the same object already exists. If so, the view manager reuses that fragment's view and its fragment implementation. Otherwise, the view manager creates a new view object and requests that the fragment-implementation factory create a local fragment implementation.

Runtime reconfiguration

The FORMI infrastructure supports dynamic reconfiguration and adaptation of fragmented objects by providing a mechanism for replacing the fragment implementation at runtime. First, it's possible to replace a local fragment with another implementation. This allows adaptations to the client-side behavior—for example, establishing caching if many invocations occur. Second, it's possible to change the fragmented object's internal structure by replacing fragments at different locations—for example, by partitioning state over multiple fragments to achieve scalability. This procedure needs object-internal communication for a coordinated transition. There could be an entity outside the fragmented object that triggers fragment replacement—for example, the client application or a resource-management system. In any case, the fragment itself ultimately initiates replacement.

Internal reconfigurations. The fragment implementation developer provides code to trigger internal reconfigurations. For example, the developer can insert instrumentation code in a stub fragment that measures the communication cost of invocations and that triggers the instantiation of a caching proxy or replica if the cost exceeds a certain threshold. The developer code calls a method provided by the view to execute the replacement. The view synchronizes the replacement with concurrent invocations at the interface and updates the references in the fragment interfaces.

Client-driven reconfigurations. A client can pass explicit policies to the view through the fragment interface for reconfigurations. The fragmented object must implement a `FORMIInterface`, which provides access to the view. Policies are name-value pairs (Java hashmaps) that describe user-defined properties. Policies can express client requirements to the object. The view notifies the fragment implementation about policy changes; the fragment implementation can then trigger a fragment replacement. In FORMI's current configuration, object developers must manually weave the necessary reconfiguration code into the application logic. We're working on automating this operation.

Support for garbage collection

Java RMI supports distributed garbage collection to automatically delete remote objects that are no longer referenced by any client. Java RMI implements garbage collection using a distributed reference-counting algorithm. At the client side, the RMI system keeps track of all references to a remote object within the JVM. When the first reference enters it, the JVM sends a *referenced* message to the server object. When a Java application has discarded all local references, the JVM sends an *unreferenced* message. The service-side RMI runtime uses weak references to refer to RMI objects that are no longer referenced remotely. A weak reference lets the local garbage collector remove the object if there are no other local references.

Distributed garbage collection within the FORMI architecture is more complex compared to standard RMI. Fragmented objects can have a complex life of their own that the middleware doesn't prescribe at all. Thus, a fragmented object's fragments must manage garbage collection themselves. Nevertheless, the middleware should support the developer in this respect and tell a fragment implementation whether or not this implementation is locally referenced. The fragments can exchange this information so that the fragmented object can determine which fragments to destroy; the fragments can even trigger the removal of the entire object.

The local view object detects whether a local fragment is referenced. The view maintains a Java weak reference for each fragment interface. The garbage collector can thus destroy any fragment interface that the local client is no longer using. The view notices such a removal and, if the last fragment interface is gone, notifies the fragment implementation through a management interface. If the local client again obtains a reference to the fragmented object, FORMI creates a new local fragment interface, and the view notifies the fragment implementation about this change.

Another situation to consider is the behavior of the local fragment implementation during termination of the JVM. There will be no final garbage collection because it would terminate the local fragment without notifying the other parts of the fragmented object. Thus, FORMI notifies fragment implementations about the JVM termination using a *shutdown hook* in the JVM, which permits code execution before shutdown. This notification lets the fragment implementation cleanly leave the fragmented object. However, the fragment developer must implement the actual realization.

For distributed garbage collection, a fragmented object must know when all fragments are no longer locally referenced so that the object can delete itself by deleting its fragments. This usually requires a specific protocol that fits the object's needs. Nevertheless, the middleware could support the developer by offering a framework for internal communication. Our prototype only provides notifications to the fragment implementation.

Example applications

Two example applications demonstrate the use of self-adaptation mechanisms and custom communication protocols with FORMI. Beside these cases, FORMI permits the implementation of even more progressive fragment implementations—for example, those that interact in a truly peer-to-peer fashion.

Dynamic smart proxy

FORMI's simple dictionary service demonstrates its self-adaptation features. A service provides a remotely accessible dictionary with a `get()` and a `set()` method at some node, implemented in a server fragment. Clients use an instrumented stub fragment as the default fragment implementation; this stub fragment provides basic remote invocation with measurement of invocation frequency. If this frequency exceeds a specific limit, the stub fragment replaces itself by a smart proxy that caches all successful retrievals. Here, we use standard RMI for internal communication. Therefore, the servant fragment additionally exports itself at RMI runtime, and client-side fragments internally receive a standard RMI reference to the exported object as part of the fragment-implementation factory. The dictionary service developer fully controls the implementation of all fragment types.

A simple experiment compares the invocation times of a traditional RMI implementation of the dictionary service with invocations to local FORMI fragments. Table 1 shows the invocation times observed by a client when using a standard RMI stub, a FORMI stub, and a FORMI cache fragment. The transition from FORMI stub to FORMI cache is fully transparent to the client. We conducted these measurements both in a distributed two-node network (AMD Athlon 2 GHz PCs running Linux and Java JDK 1.5, connected by a local 100 Mbps network) and on a single node with two independent JVMs. The instrumented FORMI stub introduces no significant overhead compared to standard RMI, and the automatic instantiation of the client-side cache results in a huge speedup.

Table 1. RMI invocation times with standard RMI and FORMI fragments.

	Standard RMI (μ s)	FORMI stub fragment (μ s)	FORMI cache fragment (μ s)
Distributed (two hosts)	188.0 (\pm 20.0)	192.0 (\pm 24.0)	0.1 (\pm 0.0)
Single host (two JVMs)	53.5 (\pm 2.3)	53.7 (\pm 2.5)	0.1 (\pm 0.0)

Dynamic audio streaming

We also implemented a fragmented Internet radio, a service that broadcasts audio streams to clients. We want to implement the service using RMI and register the corresponding radio object in the RMI registry. Thus, users can access this Internet radio with an RMI client, just like they can access any other RMI object.

With standard RMI, a client could interact with the service only through RPC-based method invocations. This doesn't work well for delivering audio data. Using the fragmented-object model, a developer can implement the radio service using different fragments: a server fragment that streams audio data through the User Datagram Protocol (UDP) and a client fragment that receives this data (see figure 4). In FORMI, the client can access the radio using a lean interface (`RadioInterface`):

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

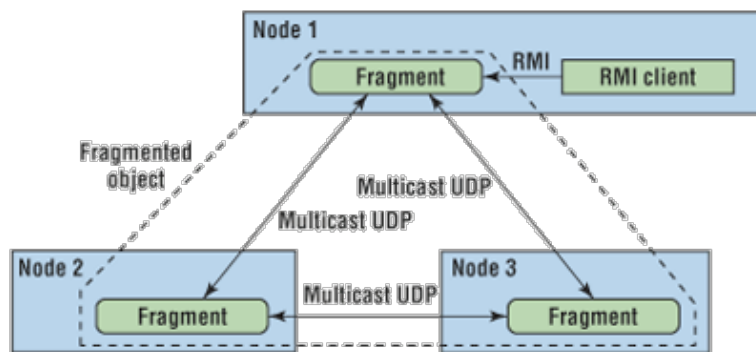


Figure 4. Fragmented audio service using User Datagram Protocol (UDP) multicast for internal communication.

You can create a radio service by instantiating an initial server fragment. A local call to the `send()` method will send audio data to all known clients. The initial fragment stores the communication address for interfragment communication in the fragment-implementation factory. Then you register the radio object at an RMI registry, and a client retrieves the reference to the radio service from the registry. The first invocation at the client triggers the instantiation of a local fragment implementation based on the information stored in the fragment-implementation factory. The client-side fragment implementation now acts like a smart proxy: it opens the socket and receives audio data, which the initial fragment makes available to the client as an audio stream.

This approach is convenient for small user groups. To increase scalability, we can provide an extended client-side fragment for stream sharing within a local network. Upon creation of a client-side fragment, that fragment sends a local multicast discovery message requesting the audio stream. If there is no response, the fragment connects to the remote streaming fragment (as described earlier) and listens for local discovery requests. If another client binds the reference at a different host in the same local network, the newly instantiated fragment again broadcasts a stream discovery request. An already connected fragment answers this request and begins to locally broadcast the stream via UDP multicast. Thus, multiple clients in the same subnet can share one dedicated connection to the serving fragment.

If the local broadcasting client fragment stops, either voluntarily or because of a node crash, another client fragment must connect the server fragment and act as a local stream provider. All client fragments will independently detect the dropped connection and again send local-broadcast stream discovery requests, which include a unique client ID. If, after a specified timeout, a node hasn't received any requests with a higher ID, it directly connects to the remote server. If a node has received a request from another node with a lower ID, it starts multicasting the incoming stream and answering new discovery requests.

This extended client-side fragment implementation goes beyond a smart proxy's traditional usage patterns, such as caching or custom data transport. In this implementation, fragments dynamically turn from simple, service-consuming client stubs to entities with server functionality.

FORMI is already a useful middleware extension to RMI. However, we plan to extend the developer support, especially for fault-tolerant objects. On the one hand, we'd like to integrate mechanisms for active replication—for example, group communication, deterministic scheduling, state transfer, and fault detection. On the other hand, we plan to enhance our FORMI compiler so that it can generate glue code for fault tolerance.

Acknowledgments

We thank Michael Kirstein for his contributions to the very first prototype, and the anonymous reviewers for their valuable comments. FORMI is available under the LGPL license as open source. You can access our prototype implementation at www.aspectix.org/formi.

References

1. M. Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle," *Proc. 6th Int'l Conf. Distributed Computing Systems (ICDCS 86)*, IEEE CS Press, 1986, pp. 198–204.
2. H.P. Reiser et al., "Integrating Fragmented Objects into a CORBA Environment," *Proc. Net.ObjectDays*, Transit GmbH; www-vs.informatik.uni-ulm.de/dept/staff/reiser/pubs/doc/reiser03integrating.pdf.
3. H.P. Reiser, M.J. Danel, and F.J. Hauck, "A Flexible Replication Framework for Scalable and Reliable .NET Services," *Proc. IADIS Int'l Conf. Applied Computing*, IADIS, 2005, pp. 161–169.
4. R. Kapitza et al., "FORMI: An RMI Extension for Adaptive Applications," *Proc. 4th Workshop Reflective and Adaptive Middleware Systems*, ACM Press, 2005, vol. 116, art. 2.
5. M. Makpangou et al., "Fragmented Objects for Distributed Abstractions," *Readings in Distributed Systems*, T. Casavant and M. Singhal, eds., IEEE CS Press, 1994, pp. 170–186.
6. F.J. Hauck et al., "A Middleware Architecture for Scalable, QoS-Aware, and Self-Organizing Global Services," *Proc. 3rd IFIP/GI Working Conf. Trends in Distributed Systems: Towards a Universal Service Market*, LNCS 1890, Springer, 2000, pp. 214–229.
7. "Java Remote Method Invocation Specification," tech. doc., revision 1.10, Sun Microsystems, 2004, <http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf>.
8. B. Dumant et al., "Jonathan: An Open Distributed Processing Environment in Java," *Distributed Systems Eng.*, vol. 6, no. 1, 1999, pp. 3–12.



Rüdiger Kapitza is a PhD student in the Distributed Systems Group at the University of Erlangen-Nürnberg, Germany. His research interests include adaptive middleware and service platforms. He received his Dipl.-Inf. in computer science from the University of Erlangen-Nürnberg. Contact him at Informatik 4, Univ. of Erlangen-Nürnberg, Martensstrasse 8, D-91058 Erlangen, Germany; rrkapitz@cs.fau.de.



Jörg Domaschka is a PhD student in the Distributed Systems Lab at Ulm University, Germany. His research interests include adaptive middleware, peer-to-peer systems, and fault-tolerant replication. He received his Dipl.-Inf. in computer science from the University of Erlangen-Nürnberg. Contact him at Distributed Systems Lab, Ulm Univ., James-Franck-Ring, O27, D-89069 Ulm, Germany; joerg.domaschka@uni-ulm.de.



Franz J. Hauck is a professor in the Distributed Systems Lab at Ulm University. His research interests include middleware systems, especially their combination with fault tolerance; other nonfunctional requirements; and middleware concepts for multimedia streaming. He received his habilitation in computer science from the University of Erlangen-Nürnberg. He's a member of the IEEE Computer Society, ACM, and German Computer Society GI. Contact him at Distributed Systems Lab, Ulm Univ., James-Franck-Ring, O27, D-89069 Ulm, Germany; franz.hauck@uni-ulm.de.



Hans P. Reiser is a PhD student in the Distributed Systems Lab at Ulm University. His research interests include middleware architectures, fault tolerance, and distributed algorithms. He received his Dipl.-Inf. in computer science from the University of Erlangen-Nürnberg. Contact him at Distributed Systems Lab, Ulm Univ., James-Franck-Ring, O27, D-89069 Ulm, Germany; reiserh@acm.org.



Holger Schmidt is a PhD student in the Distributed Systems Lab at Ulm University. His research interests include mobile object systems, adaptive middleware systems, and multimedia communication. He received his Dipl.-Inf. in computer science from the University of Erlangen-Nürnberg. Contact him at Distributed Systems Lab, Ulm Univ., James-Franck-Ring, O27, D-89069 Ulm, Germany; holger.schmidt@uni-ulm.de.

Related Links

-  [DS Online's Middleware Community](#)
-  ["Stable, Time-Bound Object References in Context of Dynamically Changing Environments," Proc. 3rd Int'l Workshop Mobile Distributed Computing](#)
-  ["Methodology for Java Distributed and Parallel Programming Using Distributed Collections," Int'l Parallel and Distributed Processing Symp.](#)
-  ["Mobile RMI: Supporting Remote Access to Java Server Objects on Mobile Hosts," 3rd Int'l Symp. Distributed Objects and Applications](#)

Cite this article:

Rüdiger Kapitza, Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Holger Schmidt, "FORMI: Integrating Adaptive Fragmented Objects into Java RMI," *IEEE Distributed Systems Online*, vol. 7, no. 10, 2006, art. no. 0610-ox001.



dsonline.computer.org

 PRINT

 CLOSE