

RMIX: Multiprotocol RMI Framework for Java

Dawid Kurzyniec, Tomasz Wrzosek,
and Vaidy Sunderam
Dept. of Math and Computer Science
Emory University
Atlanta, GA
{dawidk,yrd,vss}@mathcs.emory.edu

Aleksander Słomiński
Department of Computer Science
Indiana University
Bloomington, IN
aslom@cs.indiana.edu

ABSTRACT

Web Service technologies have recently attracted attention as promising vehicles for interoperability in e-commerce and enterprise collaboration. Attempts to leverage Web Services for high performance scientific and distributed computing, however, have encountered performance problems owing to the unsuitability of SOAP (the de facto Web Services wire protocol) for large volume data transfers. Alternative standards are likely to emerge, but currently, many different protocols remain in use. Supporting a scheme to dynamically select from multiple protocols as appropriate to the situation seems to be a sound solution. In the context of distributed computing with Java, RMI is a natural programming paradigm. However, the default RMI implementation is bound to a concrete wire protocol, JRMP, that is neither interoperable nor very efficient. In this paper, we propose a framework that permits the use of multiple wire protocols within the unified model of RMI. The wire protocols are defined in terms of pluggable transport protocol service providers that may be added to the system at any time. A suite of protocols with varying properties can then be used seamlessly and collectively. This approach facilitates applications that communicate with different classes of peers using various protocols, applications that are protocol-independent and migratable between different protocols, or distributed systems with dynamic peer-to-peer protocol negotiation. Additionally, enhancements to the RMI model towards multi-user environments are proposed. We describe the design and implementation of the framework, and present two transport service providers: the JRMPX provider using the standard RMI wire protocol (JRMP), and the XSOAP provider using SOAP. We include benchmark results demonstrating performance characteristics of these two providers in comparison to standard RMI.

1. INTRODUCTION

With increasing performance and reliability characteristics of home, institutional, corporate, and general purpose net-

works, and with the advent of Web Services and grid computing paradigms, distributed computing is rapidly gaining importance. Java-based frameworks in particular, are receiving special attention due to their portability and interoperability. In Java-based distributed systems, there are three commonly used communication protocols for interaction among cooperating entities: JRMP (default RMI), IIOP (RMI-IIOP) and SOAP (JAX-RPC). Standard Java RMI [13] is highly Java-specific and not very interoperable; RMI-IIOP [14] provides interoperability with CORBA, and JAX-RPC [12] provides interoperability with Web Services. In addition, there exist scientific implementations of RMI paradigm that focus on performance and provide support for fast networks like Myrinet. All those solutions are mutually exclusive and do not provide any facilities for mixed usage despite their common basis in the shared RMI paradigm and programming model – one that is very natural and familiar to Java developers. In this project, we attempt to address this situation by providing a flexible framework that permits the dynamic selection of transports as appropriate to a given distributed computing scenario.

Our work has two main goals. The first is to define a unified, pluggable RMI framework in which variety of transport protocols (those already existing as well as those yet-to-be-defined) may be exploited seamlessly via a unified API and with common remote method invocation semantics. Such a unification has a number of immediate advantages. First, it simplifies the development of applications wishing to accommodate different classes of remote peers. As an example, consider a scientific distributed computation running on a Myrinet cluster and using an efficient Myrinet-based protocol, but providing SOAP interfaces that permit the monitoring and control of the application from thin remote clients and handheld devices. As another example, consider a Web Service accessible via multiple endpoints over various protocols, some of them offering maximum interoperability (e.g. SOAP) while others are optimized for specific kinds of clients (e.g. Java Remote Method Protocol, JRMP). In fact, the suite of available protocols would not have to be known *a priori* and the application could discover them dynamically. Another benefit brought about by the unified RMI framework is improved reusability of software modules, since they are based on the remote method invocation paradigm but no longer depend on specific transport technologies. For instance, the migration from IIOP to SOAP would be much easier within the boundaries of such framework than otherwise possible; ideally, applications would use new transport

protocol seamlessly via the unified API without being modified.

Our second goal is to enhance the semantics of exporting remote objects, in order to give service providers more control on how remote endpoints are created. In particular, we propose APIs enabling users to explicitly specify (1) a set of remote interfaces to be exported via the newly created endpoint, and (2) a server-side interceptor that would receive remote requests on that endpoint before they are dispatched to the target object. In the current RMI model, a remote object can have only one type of proxy (in terms of an appropriate RMI stub), and such a proxy will allow any user to invoke any remote method on the target object. With the proposed enhancements, it is possible to restrict access policy on a per-user basis, by creating customized endpoints per each remote user, and restricting the set of exported interfaces when necessary. Furthermore, interceptors may be used to enforce additional access control policies that may change during the lifetime of the remote object; policies are isolated from the actual remote service implementation. We believe that such enhancements make RMI a more robust and appropriate technology for the development of multi-user, cooperative distributed systems. We label the proposed framework “RMIX”, to imply “RMI eXtended” but also suggesting “RMI MIXture”, thus reflecting both of our motivations.

This work is founded on the basis of our experiences with the H2O project [15], that strives to provide a platform for resource sharing among independent, geographically distributed, heterogeneous peers and across administrative domains. The underlying assumption in H2O is that resources are represented by remote objects that provide services via remote method invocation semantics. However, the H2O model assumes independence and isolation between service providers and service clients, imposing strong interoperability and security requirements. The abilities to define customized endpoints and support multiple transport protocols are essential, but they can not be fulfilled by current Java RMI model. Experiences with H2O make us believe that the ideas presented in this paper solve real problems and bring general benefits.

The remainder of this paper is organized as follows. In Section 2, we relate our ideas to other projects on Java distributed computing technologies. In Section 3, we discuss the design goals of the RMIX framework. In Section 4, we demonstrate how these goals were met, describing framework APIs and the underlying implementation details. We stress the portability of the purely Java-based implementation of RMIX that does not interfere with the standard Java RMI implementation from Sun Microsystems, and on the simplicity of our proposed APIs that offer optional enhancements but that also support the basic, legacy RMI model. In Section 5, we describe two transport protocol service providers that are currently available as the part of our framework: the JRMPX service provider based on standard Java RMI and using Java Remote Method Protocol (JRMP), and the XSOAP service provider based on standalone XSOAP software [9] and using SOAP as the underlying protocol. Performance figures for simple data transfer over JRMPX and XSOAP service providers, in comparison

to the standard JRMP, are shown in Section 6. Finally, conclusions and future work ideas are outlined in Section 7. Throughout the paper, we refer to the standard Java RMI implementation as “RMI/JRMP”, and use the term “RMI” to denote the canonical remote method invocation model and semantics.

2. RELATED WORK

Our system is designed to embrace existing RMI system instead of replacing a standard SUN RMI. In this sense it is a meta-system and can use most of existing RMI-like frameworks. This way we hope to build on existing knowledge and experience contained in multiple available RMI packages and to provide user with the ability to use the best features found in each of that systems.

There were already proposals to build a multi-protocol RMI system such as extensions to SoapRMI that would use SOAP as a first contact protocol and then switch to more efficient shared protocol if possible [2]. But as far as we know there is no completely designed and implemented multi-protocol Java RMI system available to date. Instead, a typical approach in research community was to investigate RMI limitations and to propose new, improved RMI systems [8, 5, 6]. KaRMI [8] is one of such systems that focuses on performance and offers some interesting capabilities. KaRMI is a drop-in replacement for standard SUN RMI, written in pure Java and designed to be a faster RMI by taking advantage of efficient serialization. KaRMI supports non-TCP/IP communication networks, e.g. Myrinet, to provide a very efficient tool for cluster-wide RMI computations. However, since KaRMI replaces SUN RMI, it is not interoperable with ordinary RMI applications and services. KaRMI is a mature system that has now evolved into a distributed framework called JavaParty [3].

There are also other approaches possible such as Manta project [18] that sacrifices Java portability and uses native code to achieve best possible performance. Manta is a native Java compiler and it compiles Java source codes to Intel x86 executables. Manta focuses on achieving source level compatibility (instead of typical bytecode compatibility) with Java codes. Because of this, code using Manta is no longer easily portable and it requires additional work to execute in new environments. Our proposed system is written in pure Java, so although it sacrifices performance to some extent, it can take advantage of Java portability.

Web Services [19] represent a general trend to simplify integrating and accessing heterogeneous services on the Internet. One example of a commercial Web Service toolkit and hosting environment is Systinet WASP (Web Applications and Services Platform) [16]. WASP is a commercial product that supports all popular Web Service technologies including SOAP, WSDL and UDDI. Although WASP has a customizable protocol stack and can leverage multiple protocols, it assumes that these protocols are XML-based that precludes its usage in high performance applications.

When compared to the RMI model, Web Services present to the user much lower level of abstraction on distributed computing. However, nothing prevents one from creating RMI layer on top of Web Services (as it was done in XSOAP [9]).

This is attractive to us as we can provide to RMI developers a simple and elegant RMI API to access heterogeneous resources over Web. We believe that in the current Internet environment, even for high performance computing, it is a necessity to access heterogeneous resources and that need can not be satisfied by any single RMI system.

3. DESIGN GOALS

This section describes in detail the design goals that were targeted by the RMIX framework.

3.1 Remote Method Invocation Semantics

Upon inspecting multiple distributed packages outlined in Section 2, it is apparent that most of them are incarnations of programming paradigms known for decades, namely: message passing, distributed shared memory, or remote procedure calls. In the case of Java, the latter usually takes the form of remote method invocations, which is arguably the most natural paradigm for Java distributed computing. Importantly, most remote method invocation solutions for Java follow the RMI remote interface definition rules and invocation semantics. RMIX builds on this common ground, inheriting the following set of rules from the RMI specification [13]:

- Remote access is possible only through objects implementing one or more *remote interfaces*, in terms of invoking their *remote methods*.
- Remote interface must extend `java.rmi.Remote` marker interface and may only contain remote methods.
- Remote method must have `java.rmi.RemoteException` (or its superclass) in its throw clause.
- Remote method parameters and return values are passed by value (as deep copies) except remote objects that are passed by reference (they are substituted with appropriate stubs).
- Remote objects passed by reference must be represented by their remote interfaces (rather than the actual implementation class) in the signatures of appropriate methods.
- Clients access remote objects via proxy objects (*stubs*) that implement remote interfaces of the target object and forward method invocations to the target object. In the original RMI, stubs always implement the full set of target object's remote interfaces. In RMIX, the set can be restricted on a per-endpoint (or effectively per-client) basis.
- If one of methods `hashCode`, `equals`, or `toString` is invoked on a remote stub, the call is handled within the client context (without contacting target object). Remote stubs of the same target object are always equal in terms of `equals` and `hashCode`, so they may be used as hash table keys.

Most of remote references used within a distributed application are obtained from remote method calls themselves. However, the bootstrap mechanism is needed in order for a

client to establish first contact with a remote server. The means of this mechanism varies among different technologies. The original RMI provides a simple naming service – an `rmiregistry` utility. Similarly, the notion of a naming service is used in CORBA environments. Web Services introduce a more sophisticated registration and lookup mechanisms in terms of WSDL service descriptors [1], WS-Inspection [4] documents grouping them along with additional metadata, UDDI registries [17] which can be used to store and locate information, and other techniques. In fact, even some out-of-band mechanisms (like e-mail, or a telephone) may be used to notify clients about service endpoints. We believe that due to this variety, the choice of an appropriate strategy should be left to the application. Therefore, RMIX does not mandate any particular type of a discovery mechanism. However, in order to provide some common ground and retain backward compatibility, it is guaranteed that every RMIX remote proxy can be stored in the `rmiregistry`.

3.2 Transport Protocol Service Providers

One of the key enhancements introduced by RMIX is dynamic support for multiple wire protocols. This is important, because various applications (or even various parts of the same application) may have different demands on the interoperability, performance characteristics, or semantical richness of a communication link, and there is no single, widely accepted protocol that would deliver all of them at the same time.

Specifically, there is no standard suite of protocols supported by RMIX. Instead, each protocol is independently served by the appropriate *transport protocol service provider* (SPI). Providers are completely independent modules, and they may be added to the system at any time. In order to add a provider, it is sufficient to place a JAR file containing its classes into the appropriate directory – no further configuration is necessary. Applications use dynamic discovery to find out about available providers, and the addition of a provider in the manner described above makes it immediately visible through this dynamic discovery mechanism. This model of dynamic upgradeability is especially beneficial for long-running server applications, as they may dynamically switch to new protocols without being restarted. The model also provides means to replace specific providers with newer versions at run time, although this requires the application to participate in the process by explicitly instructing RMIX runtime to unregister the currently loaded version of a provider.

Ideally, the application should be able to use any available protocol transparently, and obtain exactly the same semantics. However, RPC protocols differ in their level of sophistication, and it would be unreasonable to expect from every protocol to be able to support full RMI/JRMP functionality. In particular, the protocols may have restrictions on the types of parameters and return values of remote methods, or on the number of remote interfaces that remote stub may implement. Also, they may or may not support distributed garbage collection. The current version of RMIX does not explicitly address that issue, requiring an application to have a certain knowledge of the protocols it wishes to use. In future versions, we will introduce mechanisms for

transport providers to reveal additional meta-information about themselves via the dynamic discovery process. This meta-information will most likely take a form of the level of conformance to RMI semantics. Having access to that meta-information, applications will be able to make informed protocol choices without explicit protocol dependencies. At the lowest, mandatory conformance level, all protocols must be able to transmit instances of the following types as arguments and return values:

- all Java primitive types;
- class `java.lang.String`;
- arrays of primitive types and strings;
- Java classes declared `final` that have only public fields (or conform to the Java Bean design pattern), types of that fields (or bean properties) being either primitive types or strings;
- RMIX remote stubs (and *endpoints*, as it will be described in Section 4) regardless of transport providers backing that stub (or endpoint).

The last requirement defines RMIX protocol interoperability semantics, guaranteeing that RMIX remote proxies are globally serializable across all possible transport protocols. For instance, it is possible to serialize JRMP proxies using SOAP, and to serialize SOAP proxies using JRMP. This requirement is critical in order to support dynamic protocol negotiation and run-time protocol switching. Facilities offered by RMIX to assist providers in fulfilling the requirement without introducing explicit inter-provider dependencies will be described in the Section 4.

There are situations, however, when an application wishes to exploit specific features of a transport protocol provider. For instance, it may wish to export remote objects via JRMP using custom socket factories [13]. The RMIX framework's API allows for such customization, as will be described in the Section 4.

3.3 Customizable Remote Endpoints

In order to be able to receive remote method invocations, a remote object must be first *exported*. During the process, an *endpoint* to the object is created. RMI runtime starts listening on the endpoint and handling incoming method invocation requests. An address of the endpoint must be somehow delivered to the client. It can be passed as a return value from another remote call, fetched from a naming service, or even received via e-mail. Knowing the endpoint address, the client instantiates a proxy object (i.e. *stub*) that *binds* to that address. The stub handles the client's method invocations and forwards them to the endpoint, where they are received and dispatched to the target object by the RMI runtime. This invocation stack is depicted in Figure 1.

This conceptual model is followed by most (if not all) existing Java RMI systems. However, they usually allow only a single endpoint for any target object to be created. Since endpoints are strictly transport-specific, responsibility for creating them (thus, exporting objects) in RMIX is assumed

by transport protocol service providers. It has been thus a natural design choice to support multiple endpoints per target object, as they may be backed by various providers. Common use of this feature would be to create a set of multiprotocol endpoints to an object that provides some sort of a remote service, and publish them collectively in the form of a WSDL document. Both original and proposed models are contrasted in the Figure 2.

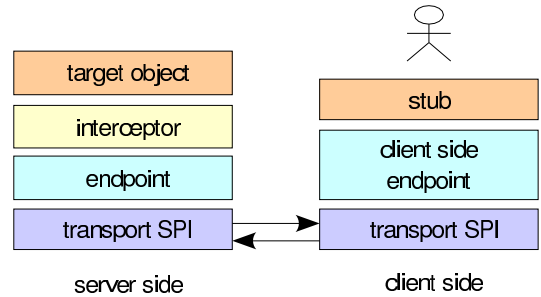


Figure 1: RMIX invocation stack

However, this enhancement has significant semantical consequences. In the default RMI, a remote object passed to or returned from a remote method is either substituted by the appropriate stub corresponding to the endpoint on which the object was exported, or (since Java 1.3) passed by value in case it has not been exported yet. However, in RMIX the number of endpoints may be anything from zero to infinity, so there is no default stub the object should be substituted with. For that reason, we decided to introduce different semantics which we feel are more appropriate under the aforementioned circumstances:

- If a remote object (i.e. object that implements the `java.rmi.Remote` interface) is returned from a remote call, it is substituted with a stub created by the same transport protocol service provider as the endpoint through which the call was dispatched, inheriting that endpoint's export parameters. For instance, remote stubs returned from an encrypted JRMP connection will continue to use encrypted JRMP channels to communicate with their own endpoints.
- If a remote object is passed as an argument to a remote call, it is substituted with a stub created by the same transport protocol service provider as the stub on which the call is invoked, but with default export parameters. For instance, callback objects passed to remote object over encrypted JRMP connection will (by default) use plain JRMP to communicate with their endpoints.
- In both cases, if an appropriate endpoint exists already, transport provider may choose to reuse it; otherwise, the object is exported and new endpoint is created.

These default substitution rules are only applied if remote objects are passed directly to or from remote calls. Users may choose to bypass these rules by performing their own

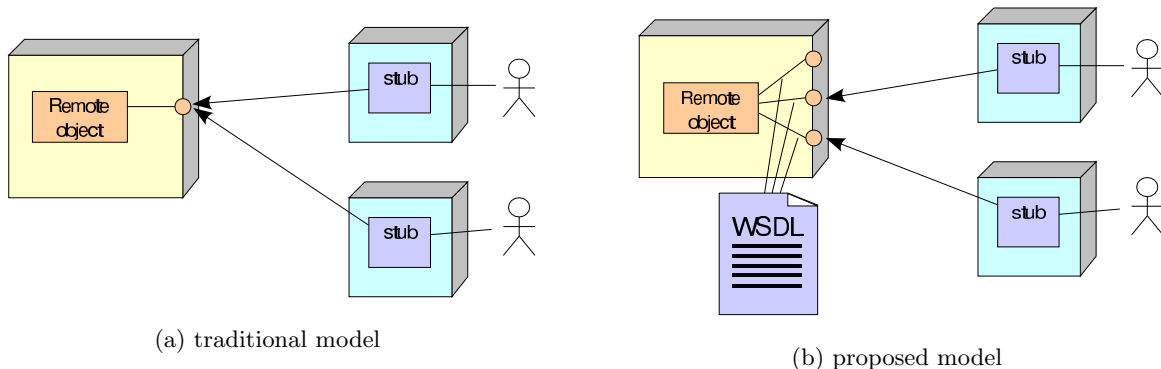


Figure 2: Remote communication in RMI

substitution (i.e. use stubs instead of actual objects) e.g. to perform protocol switching. These rules effectively define RMI dynamic export semantics: remote objects passed to or returned from remote methods are automatically exported as necessary. It is no longer required to explicitly export remote objects when they are created; RMI considers the very fact that the object was marked with `java.rmi.Remote` interface as sufficient to decide that remote semantics should apply.

As a consequence of dynamic export, RMI never sends remote objects by value. However, we believe that it is not a good programming style in any case to depend on pass-by-value semantics for unexported remote objects, so we do not consider this semantical incompatibility to be significant.

RMI allows for further customization of remote object endpoints, in terms of (1) restricting the set of exported remote interfaces and (2) specifying an invocation interceptor that will catch remote calls before they are dispatched to the target object, as shown in the Figure 1. Both features may be used to implement fine-grained access control semantics. For instance, endpoints may be created on a per-client basis over secure connections. Every client may be given its private endpoint with tailored set of remote interfaces and/or guarded by specific interceptor. Restricting interface set effectively limits a set of methods that the client may invoke; in fact, if some interfaces were hidden by the endpoint, client will be unaware that they were implemented by the target object. Interceptors may enforce more dynamic access control policy, i.e. they may block (or redirect) specific method invocations according to some criteria that may change during program execution. These enhancements simplify development of Java-based multi-user distributed applications, allowing to keep access control functionality apart from the actual application classes. These features have found immediate application in the H2O system [15], and they have significantly contributed to its simplicity.

Stubs used in the default RMI (and in many other implementations) are instances of stub classes that must be generated in compile time using appropriate tools (e.g. `rmic` [13]). It places significant burden on development and usage scenarios of distributed applications, since stub class bytecodes must be somehow delivered to clients. Two solutions are possible. One is, stub classes may be bundled

with client application codes. However, it has several disadvantages: it pins client down to a particular transport protocol (since stubs are usually protocol-dependent), and it affects maintainability of client-server applications, since changes made at the server side need to be synchronized with clients. The other solution involves publishing stub classes on the network and letting clients download them as needed using Java dynamic class loaders. However, it requires using additional file or HTTP servers and increase load on the client's network connection. RMI avoids all that problems whatsoever by leveraging dynamic proxy functionality introduced in J2SE 1.3 [11]. This feature allows RMI stubs to be generated dynamically within the client context and according to the endpoint characteristics.

3.4 Localization Aspects

Usually, clients invoke remote methods from remote machines. However, there are situations (especially in distributed systems with load balancing and component migration [3]) that client is local to the target object's JVM. In such case, it is appealing to exploit the opportunity and dispatch method locally. The straightforward and the most efficient solution would be to substitute remote stub with a direct target object reference. However, this solution is unsatisfactory for the number of reasons:

- It does not retain remote method invocation semantics, backing to default Java semantics of passing objects by reference. If client code relies on the fact that remote method arguments are passed by value, it will fail.
- It bypasses RMI access control mechanisms defined in terms of restricted interface set and server-side interceptor.
- It bypasses the symbolic resolution of remote interface names, that may lead to class cast exceptions if client and server used different class loaders to load remote interface classes.
- Since client refers to the target with a strong reference, it is impossible to forcibly disconnect it.

To overcome these issues, the remote stub indirection layer must be retained along with marshaling and unmarshaling

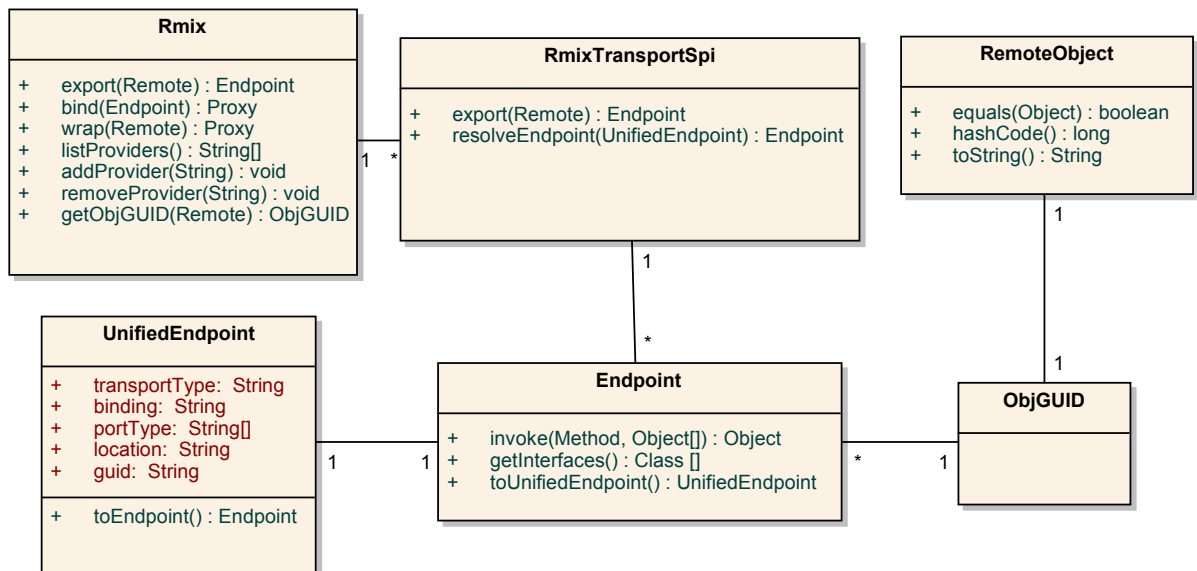


Figure 3: RMIX API

codes [8]. Short-cutting may start as late as at the marshaling level; primitive values, immutable objects, and remote stubs may be passed without change but for other objects, pass-by-value semantics must be emulated by performing marshaling and unmarshaling as in the remote dispatch [8]. This kind of operation is transport provider-specific, thus the optimization is left to providers. However, we consider introducing general extensions (in terms of export-time hints) that would make it possible to instruct RMIX to compromise semantics for better performance. Two levels of optimization are possible: allowing clients to obtain a direct target reference, or preserving the remote stub indirection layer and giving up only on (the most costly) pass-by-value semantics.

4. RMIX API AND IMPLEMENTATION

This section describes in detail the RMIX public APIs and explains how the design goals outlined in the Section 3 were met by our implementation. The RMIX API is localized in the Java package `edu.emory.mathcs.rmix`, and consists of a small number of classes and interfaces, as shown in Figure 3. The central class, `Rmix`, provides a set of static methods through which most RMIX functionality is exposed. They provide facilities to export remote objects, bind to remote endpoints, manage transport protocol service providers, and handle object identity. The `Endpoint` class represents a serializable object endpoint, and `UnifiedEndpoint` is its unified, protocol-independent counterpart. `RmixTransportSpi` is a base class for transport protocol service providers. `ObjGUID` class represents globally unique object identifier, as used by RMIX to compare remote stubs. `RemoteObject` is a convenience class for remote objects to inherit from, providing them with remote comparison semantics.

4.1 Export and remote bind operations

The `Rmix` class provides several overloaded variants of an `export` method, that is used on the server side to create

endpoint to a remote object. The simplest version of the method accepts a single parameter being a remote object to export. It is roughly equivalent to the `UnicastRemoteObject.exportObject` method from the `java.rmi.server` package as it uses a default protocol (as determined by value of an appropriate property), and it exports all remote interfaces implemented by the target object. Overloaded variants of that method allow the specification of the particular transport provider to use (through a fully qualified class name), set of remote interfaces to export, server-side interceptor, and the provider-specific set of parameters.

As a result of `export`, an instance of the `Endpoint` class is returned. It is then delivered to the client that binds to the endpoint and creates dynamic stub with the `bind` method. The base version of the `bind` method accepts a single parameter that is the endpoint to bind to. By default, attempt is made to resolve all remote interfaces exported through the endpoint (as to be implemented by dynamic stub) with current context class loader. Optionally, the class loader may be explicitly given along with a boolean flag indicating if it is necessary that all interfaces are resolved. With this parameter set to false, client is able to make use of a remote object even if it has access to bytecodes of only selected interfaces. It allows for extending server functionality (via new interfaces) without affecting existing clients.

As a convenience, a set of `wrap` methods is provided that combine the effects of `export` and `bind` and result in a dynamic stub generated at the server side. When the stub is sent to the client, the outcome is as if an `Endpoint` was sent and `bind` was invoked by the client. Sometimes it is necessary to generate stubs at the server side – common scenario is to explicitly return customized stubs from remote methods instead of depending on default substitution mechanisms. In the original RMI, stubs are always instantiated at the server side. RMIX introduces the two-phase export/bind process

since it is more flexible (as the additional bind options may be given by the client) and imposes less overhead on the server (as dynamic stub generation may be costly).

Note that simple applications will need to use this API only to a very limited extent, i.e. to create endpoints for bootstrap objects (e.g. by invoking `wrap` method and storing the resulting stub in the `rmiregistry`). After that, default substitution rules and dynamic export will guarantee proper semantics for remote reference serialization. More frequent use of the API is required only in non-trivial usage scenarios, e.g. involving protocol switching or endpoint customization.

4.2 Pluggable transport service providers

Since semantics of endpoint creation are protocol-specific, `Rmix` delegates export requests to appropriate transport service providers. If the provider with the requested name has not been used before, `Rmix` attempts to locate it using the process of dynamic discovery. To be dynamically discoverable, a service provider must be bundled within the JAR file with defined global attribute “`Rmix-Provider-Class`” pointing to the main class of the provider. Dynamic discovery involves scanning all JAR files in the *provider search path*, checking if they define that attribute, and if so, comparing its value to what is searched for. The provider search path consists of the directory `$JRE/rmix`, followed by the default JRE extension path, followed by a path defined by the property `rmix.spi.path`. Thus, providers may be added either globally or per user/application. If the appropriate JAR file has been located, the provider main class (that must extend `RmixTransportSpi`) is loaded and instantiated, and that singleton instance is kept by the RMIX to handle further requests. Thus, costly dynamic discovery is performed only on the first use of a particular protocol.

Applications may explicitly perform dynamic discovery to find out about available transport providers, using the `findProvider` method. The resulting list is never cached by RMIX, so that an up to date set is always returned. Applications may also explicitly install providers (that would not necessarily be discoverable otherwise; e.g. those not bundled into JAR files) using `addProvider` method. Similarly, the `removeProvider` method can be used to unload a specific provider. If that provider has not been loaded, this method has no effect. Otherwise, it causes RMIX to detach from the provider object so it becomes eligible for garbage collection as soon as endpoints backed by it are closed. If a new export request for the same provider is issued, and if the class is dynamically discoverable, RMIX will reinstantiate the provider from the possibly updated, rediscovered JAR file.

4.3 Protocol interoperability

As mentioned in Section 3, RMIX assures global serializability of remote stubs and endpoints across all transport protocols. This translates into the requirement that any transport SPI is able to serialize and deserialize endpoints created by any other SPI (note that serialization of stubs reduces to serialization of endpoints coupled with the `bind` operation at the receiver side). This is supported by the `UnifiedEndpoint` class that provides inter-protocol, unified endpoint format. This class has the following fields:

- *transportType*: class name of a transport SPI backing this endpoint.
- *binding*: provider-specific binding description. It may contain version information and/or other protocol properties.
- *portTypes*: set of remote interfaces exported via this endpoint, encoded in a provider-specific way.
- *location*: provider-specific string that represents network address of the endpoint.
- *guid*: globally unique identifier of a target object.

All protocol providers must implement two methods that convert their endpoints to and from the unified format, appropriately: `toUnifiedEndpoint` in the `Endpoint` class, and `resolveEndpoint` in the main provider class. Other providers can thus replace alien endpoints with their easily serializable, unified counterparts on the wire, and restore the originals at the receiver side. This is hidden behind common API, so that the two providers never have to talk to each other directly.

4.4 Remote objects identity

This is a very useful RMI feature that two remote stubs pointing to the same remote object are equal in terms of `equals` and `hashCode` methods. In standard RMI, this is achieved by comparing appropriate object endpoints that stubs refer to. However, a similar approach is not possible in RMIX since endpoints are protocol-dependent and thus not comparable. To solve this issue, RMIX associates globally unique object identifiers (GUID) with every exported object. Remote stubs contain GUIDs of their target objects, and those GUIDs are used to compare stubs. In effect, RMIX manages to extend RMI stub identity semantics for the case of multiple independent protocols.

RMIX bases its GUID generation scheme upon standard RMI solutions: the `ObjID` that represents an identifier unique within a JVM, and the `VMID` that is a globally unique JVM identifier. In RMIX, a pair of these represent a GUID that is unique under the following conditions (which are inherited from the `ObjID` and `VMID` classes):

- It is possible to determine unique and constant address of a local host;
- System clock is never set backward;
- Restart of the system requires at least 1 millisecond.

Even if those conditions are not met, the probability of a clash (false positive) never exceeds 2^{-32} for objects from the same JVM, and 2^{-64} for objects from different JVMs.

Similarly to standard RMI, RMIX provides a `RemoteObject` class that may be used as a convenience superclass for user-defined remote objects. It provides implementations of methods `equals`, `hashCode` and `toString` that make an object comparable to its remote stubs. This is achieved simply by keeping appropriate GUID in the object and using it for comparisons.

5. JRMPX AND XSOAP PROVIDERS

5.1 JRMPX

The JRMPX transport service provider implements RMI functionality on top of the standard RMI implementation that uses JRMP as a wire protocol. JRMPX retains complete semantics of standard RMI, including full compliance with the Java serialization specification, as well as support for class annotation and dynamic remote class loading. However, applications using JRMPX can take advantage of additional features supplied by the framework:

- Dynamic stub generation. Use of `rmic` compiler is not necessary, and the issues of handling stub classes are avoided whatsoever.
- Multiple, customized endpoints. Remote objects may be exported on many endpoints at the same time. Endpoints may have different characteristics, they may export different set of interfaces, and they may be guarded by invocation interceptors.

JRMPX is neither a modified version of standard RMI, nor it is written from scratch. Instead, it is build *upon* standard RMI and uses it to tunnel remote calls. This makes JRMPX lightweight, simple, and easily maintainable, as it consists of less than twenty simple classes. JRMPX does not depend on any non-public APIs, so it will remain compatible with upcoming releases of the Java platform; moreover, it will immediately benefit from any future RMI performance improvements. However, that approach itself introduces an overhead that will be discussed in Section 6.

5.2 XSOAP

XSOAP toolkit (formerly SoapRMI [9]) is an RMI system implemented in Java and C++ that is using SOAP as wire protocol. This toolkit, besides standard SOAP 1.1 functionality, provides convenient remote method invocation abstraction that works both in Java and C++. This allows to create SOAP based Web Services as easily as using the `UnicastRemoteObject` to export remote objects in Java RMI. Additionally to RMI API in XSOAP every client and server has access to a `soap-services` module. This module encapsulates all the SOAP services like SOAP serialization and deserialization that any object may need including access lower level information and functions that are not exposed by RMI API.

In order to transform XSOAP into RMIX transport service provider, several changes had to be made. Specifically, proprietary counterparts of the `Remote` interface and the `RemoteException` class were replaced with those from the standard RMI model. Also, the XSOAP's `UnicastRemoteObject` has been modified to subclass from the `RemoteObject` class provided by RMIX. Several new classes were added, and a number of existing files were modified, in order to provide support for multiple, customizable endpoints, awareness of alien remote stubs, and semantics of stub comparison.

6. PERFORMANCE EVALUATION

Our performance tests consisted of series of runs of benchmarks adapted from the KaRMI benchmark suite [7]. They

include throughput and latency tests and two application benchmarks. The runtime platform consisted of a local network of 11 Sun Blade 1000 workstations running 64-bit Solaris 8, each equipped with two Ultra-Sparc-II 750 MHz CPUs, a 150 MHz system clock, 1GB of RAM, and a Fast Ethernet network adapter. All executable and data files were installed on a shared NFS filesystem. Java platform used was J2SE 1.4 for Solaris/SPARC. The limit on the JVM heap size was set to 128 MB.

Simple ping test results are shown in Figure 6. Each tests consisted of 1000 method invocations with various arguments: (1) void, (2) null argument, (3) a struct of 4 integers, (4) a struct of 32 integers, (5) level 0 binary tree and (6) level 3 binary tree of simple objects. As can be seen in the figure, JRMPX has an overhead of the order of 20-50% over standard RMI, as a side effect of marshaling/unmarshaling additional objects necessary during JRMP tunneling. XSOAP's overhead varies between 2600 and 3500%, that is in fact not surprising and consistent with other reports [2].

Throughput tests, shown in Figures 4 and 5, involved unidirectional transmission of byte and float arrays, respectively, with increasing array sizes. In both it can be plainly seen that standard RMI is a little faster than the JRMPX protocol when small size arrays are considered, however, when greater amounts of data are sent both protocols saturate network bandwidth nearly at the same point. It should also be noticed that the similar shape of the curves (RMI and JRMPX) on both figures shows that encoding of float-point arrays have been improved since earlier RMI releases [8]. SOAP throughput is 40 times on average (20 with 200 floats and 62 with 200 000 floats) worse than the other two protocols; however, byte arrays are sent in a more efficient manner which is a result of the dedicated encoding (base64). On the other hand, SOAP encoding of float arrays requires separate XML elements for every array element, that introduces huge overhead both in terms of processing time and message size. In fact, the test case that involved transmission of 8 MB float array crashed due to "out of memory" conditions, thus its results are not shown.

The last part of the performance benchmark were application tests (Figure 7). They were performed on 1 server and 8 clients and they were solving two different tasks: (1) Hamming's problem that requires many client threads to communicate with the server, and (2) SOR where clients communicate with server to gain new portions of data to work on. The results of those tests show the JRMPX overhead of up to 100% in comparison to standard RMI, whereas appropriate XSOAP overheads approached 2000%.

The global picture that emerges in regard to JRMPX is that the benefits that it provides come for a price in the method invocation time. This may be acceptable for some applications; for those where it is not, there are probably better alternatives to JRMP in any case. Nonetheless, we believe that we will be able to improve the performance of JRMPX in the near future by careful profiling and optimization. SOAP figures prove once again that this protocol may not in any respect be considered high performance. However, SOAP delivers interoperability, and it is very well suited for event streams [10] or as a "first contact protocol"

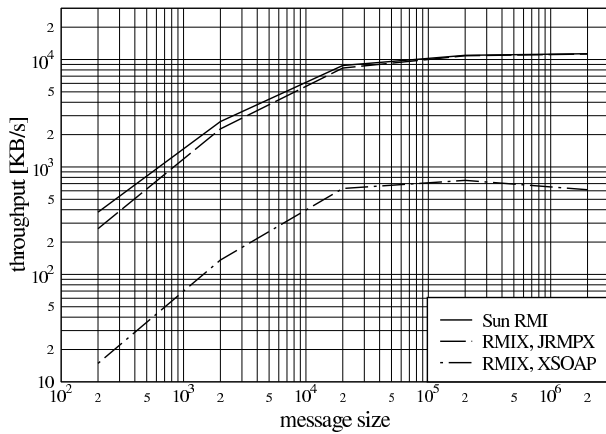


Figure 4: Throughput results; unidirectional transmission of byte arrays

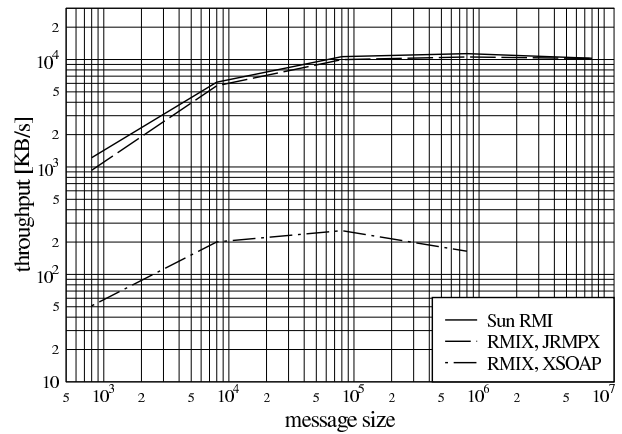


Figure 5: Throughput results; unidirectional transmission of float arrays

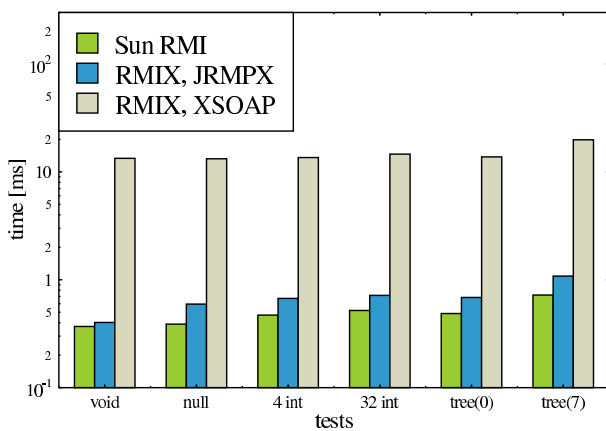


Figure 6: Ping tests with different arguments

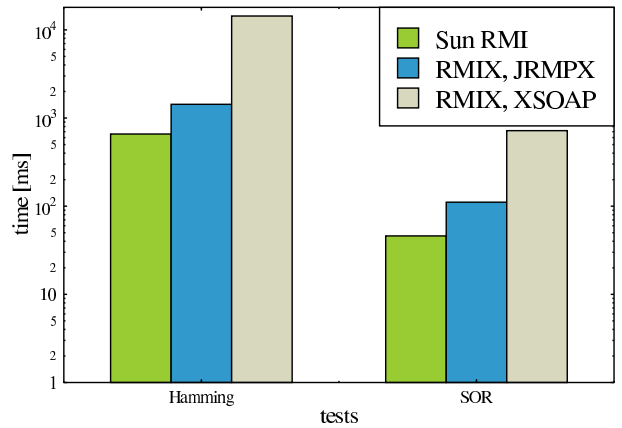


Figure 7: Application tests; Hamming's problem and SOR

in dynamic protocol negotiation scenarios. Also, it makes JVM capable of hosting Web Services. Therefore, we believe that SOAP is an extremely important part of a protocol suite in a multiprotocol framework.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have described the RMIX framework that provides a consistent RMI programming model over a suite of transport protocols. We show how the programming model is separated from implementation of the actual protocols that are independently handled by pluggable service providers. We stress that protocols may be added to the system at any time and become immediately available to applications via dynamic, run-time discovery. We show how interoperability between different protocols was achieved, including mandatory support for `rmiregistry`, well defined comparison semantics between remote stubs backed by different protocol providers, and guarantees of remote stub serializability across all protocols. Additionally, we propose enhancements to the RMI model that enable customization of remote object endpoints. We argue that the RMIX framework simplifies development of various kinds of distributed applications, e.g. if they need to accommodate dif-

ferent peers over different protocols, if the protocol independence is to be achieved, or if the dynamic protocol negotiation features are desired. We emphasize simplicity of the proposed API and describe a transition path of legacy RMI applications towards the RMIX framework, along with emerging benefits of such transition. We analyze features and performance characteristics of two supplied transport service providers that use JRMP and SOAP, appropriately, as the wire protocols.

We are currently working on several improvements to RMIX. These include the specification of semantical conformance levels to be revealed by transport protocol service providers during dynamic discovery. This enhancement will obviate otherwise inevitable dependencies of non-trivial applications on particular transport protocols. Also, we will design and implement the uniform notion of endpoint removal, i.e. an "unexport" operation. The semantics of such an operation is not obvious in the presence of multiple endpoints possibly backed by different transport providers, and it requires careful design. Also, current version of RMIX does not address the problem of collaboration between independent distributed garbage collectors. For instance, if a JRMPX stub

is returned from a SOAP invocation, it is currently the application's responsibility to keep the target object alive until the stub reaches the receiver and is reconstructed there. We are presently investigating possibilities to address this issue.

This work is a part of broader initiative that includes design and specification of new protocol bindings to the WSDL language. Currently, there are few WSDL bindings defined, and none of them is appropriate for scientific computing. The envisioned goal is to be able to export a Web or Grid Service on multiple standardized protocols described within a single WSDL document, using RMIX as a server tool to create that various endpoints and as a client tool to bind to them. In addition to existing JRMPX and XSOAP service providers, we will develop provider modules (and define WSDL bindings) for such protocols as RPC/XDR, IIOP, and some instance of RMI protocol optimized for fast networks, like the one used in KaRMI. At the same time, we hope that such RMIX features as its extensible architecture, simplicity of transport provider API, and unified environment that enables protocol coexistence and cooperation but without interference, will encourage other parties to independently supply their other transport protocol service providers.

8. REFERENCES

- [1] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl/>.
- [2] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and Evaluation of RMI protocols for scientific computing. In *Proceedings of the IEEE/ACM SC2000 Conference*, Dallas, Texas, USA, Nov. 2000. Available at http://www.extreme.indiana.edu/xgws/papers/sc00_paper/.
- [3] B. Haumacher and M. Philippsen. JavaParty. <http://www.ipd.uka.de/JavaParty/>.
- [4] IBM, Microsoft. Web Services Inspection Language (WS-Inspection). <http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html?dwzone=webservices>.
- [5] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient implementations of Java remote method invocation (RMI). In *4th USENIX Conference on Object-Oriented Technologies and Systems*, pages 19–36, 1998.
- [6] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java's Remote Method Invocation. In *Principles Practice of Parallel Programming*, pages 173–182, 1999.
- [7] C. Nester, M. Philippsen, and B. Haumacher. RMI benchmark suite. Available at <http://www.ipd.uka.de/~hauma/KaRMI/benchmarks.html>.
- [8] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for java. In *Java Grande*, pages 152–159, 1999. Available at <http://citeseer.nj.nec.com/nester99more.html>.
- [9] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based interoperable RMI system: SoapRMI C++/Java. In *Proceedings of Parallel and Distributed Processing Techniques and Applications Conference*, Las Vegas, NV, USA, June 2001. Available at <http://www.extreme.indiana.edu/soap/rmi/design/>.
- [10] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. SoapRMI events: Design and implementation. Technical Report TR549, Department of Computer Science, Indiana University, May 2001.
- [11] Sun Microsystems. Dynamic proxy classes. <http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>.
- [12] Sun Microsystems. Java API for XML-based RPC. <http://java.sun.com/xml/jaxrpc/>.
- [13] Sun Microsystems. Java Remote Method Invocation specification. <http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html>.
- [14] Sun Microsystems. Java RMI over IIOP. <http://java.sun.com/products/rmi-iiop/>.
- [15] V. Sunderam and D. Kurzyniec. Lightweight self-organizing frameworks for metacomputing. In *The 11th International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002. (to appear).
- [16] Systinet. WASP (web applications and services platform). <http://www.systinet.com/>.
- [17] Universal Description, Discovery and Integration (UDDI). <http://www.uddi.org>.
- [18] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. *ACM SIGPLAN Notices*, 36(7):34–43, 2001.
- [19] V. Vasudevan. A Web Services primer. <http://www.xml.com/pub/a/2001/04/04/webservices/>.