RMIX: A 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

With the increasing adoption of Java for parallel and distributed computing, there is a strong motivation for enhancing the expressive elegance of the RMI paradigm with flexible and adaptable communication substrates. Java RMI is an especially powerful and semantically comprehensive framework for distributed Java applications – but the default Java RMI implementation is bound to a concrete wire protocol, JRMP, that is neither interoperable nor very efficient. To address the first issue, libraries have been proposed that provide RMI semantics over different wire protocols such as SOAP or IIOP, making Java interoperable with Web Services and CORBA. Similarly, alternative high performance RMI implementations have been developed. However, none of these solutions are designed to work cooperatively, and each imposes specific constraints on developers. This paper describes RMIX: an RMI framework that supports a variety of dynamically pluggable wire transports underlying a common and uniform RMI facade. RMIX facilitates dynamic protocol negotiation in loosely coupled parallel and distributed systems, and enables the development and deployment of applications that are multiprotocol by nature. Additionally, RMIX offers some enhancements to RMI semantics that are particularly useful in multiuser environments. We describe the design and preliminary implementation of RMIX, present two prototype protocol providers based on the JRMP and SOAP protocols, and outline a transition path from legacy RMI applications to RMIX.

1. Introduction

Parallel and distributed computing has gained renewed attention with the advent of grid computing, peer-to-peer, and Web Services paradigms. Java-based frameworks to support parallel and distributed systems in particular, are receiving special attention due to their portability and interoperability. Remote Method Invocation (RMI) – an adaptation of Remote Procedure Call (RPC) for object-oriented environments – is undoubtedly one of the most popular programming paradigms in Java-based distributed systems, owing to its expressive elegance and comprehensively defined semantics. RMI allows clients to seamlessly invoke methods of objects instantiated within remote servers, with well understood syntax and semantics that are virtually identical to local method invocations. In Java RMI the implementation of this paradigm is based on a communication protocol stack that defines connection management, message formats, and data encoding. The "Standard Edition" Java Platform contains a standard RMI implementation [17] based on the Java Remote Method Protocol (JRMP) stack. While JRMP is full-featured and sophisticated, it assumes the Java object model on both the client and server sides; thus its usability is limited to pure Java systems. Alternatives to standard Java RMI do exist – for example, RMI-IIOP [18] enables connectivity with CORBA, while JAX-RPC [15] uses SOAP/HTTP [1] and provides interoperability with Web Services. Similarly, there are RMI implementations specifically targeted for high performance computing and optimized for low-latency networks [9, 11, 12, 22]. These alternatives partially address performance issues of standard RMI, but usually at the cost of decreased functionality and cross platform interoperability.

A common drawback of these alternative communication libraries is that they are not designed to work cooperatively. For instance, they usually bind clients to a particular wire protocol via stubs that must be generated at compile time, and they assume that a uniform communication fabric is used among communicating parties. This is appropriate for tightly coupled systems, but prevents the development of more dynamic and loosely coupled applications with run-time protocol negotiation. Also, the installation procedures and programming interfaces of these libraries are implementation-specific, contributing to a steep learning curve, and, in many cases, effectively limiting broad adoption.

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

In this project, we attempt to address this situation by introducing a flexible framework that provides a unified facade over various RMI implementations, both those that currently exist and those yet to be developed, enabling the dynamic selection of transports as appropriate to a given distributed computing scenario. The actual transports are supported by pluggable modules, referred to as *protocol service providers*. By unifying the programming API and module deployment process, the proposed RMIX framework strives to abstract applications away from the specifics of a communication protocol stack, as shown in Figure 1. It supports loosely coupled, interoperable systems, as well as applications that are multiprotocol by nature – e.g. those using efficient protocols to exchange large amounts of data but relying on SOAP for management and event notification. In addition, the framework introduces several enhancements to RMI semantics with a view to better supporting multi-user environments. In particular, RMIX permits the customization of endpoints of exported remote objects for different clients.

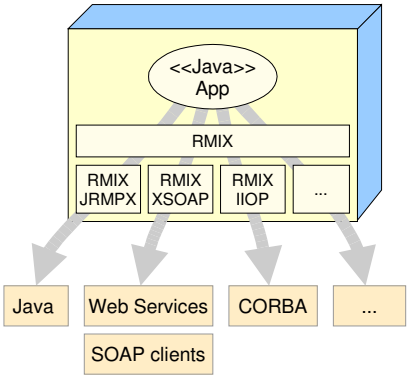

Figure 1. RMIX Interoperability

This work was motivated by, and provides a foundation for, the H2O project [19], a distributed computing platform supporting resource sharing among independent, geographically dispersed, heterogeneous peers in multiple administrative domains. In H2O, resources are represented by remote objects that offer services via remote method invocation. The H2O model assumes loose coupling between service providers and service clients, but it also strives to provide maximum communication performance. To achieve this goal, H2O uses SOAP as a *first-contact protocol*, with subsequent negotiation between the client and server for more efficient, specialized protocols to link actual application modules. Although RMIX was designed to address exactly this need, we believe that it will be applicable and useful in any situation where dynamic selection of transports within a unified RMI framework is needed.

This paper describes the design and architecture of the RMIX framework. In a companion paper [10], we present a complementary perspective related to the application of RMIX in heterogeneous systems. The remainder of this paper is organized as follows. In section 2, we discuss related work, and in section 3, we present the design and prototype implementation of the RMIX framework. In section 4, we describe two transport protocol service providers that are currently available as part of our framework: the JRMPX service provider based on standard Java RMI/JRMP, and the XSOAP service provider based on standalone XSOAP software [13] that uses SOAP as the underlying protocol. Section 5 discusses the transition path from legacy RMI applications to RMIX. Finally, section 6 concludes the paper with some discussion of ideas for future work.

2. Related work

The RMIX project aims to unify RMI API and semantics, and is designed to embrace various RMI implementations within a common framework. Such an approach can leverage the strengths of (multiple) existing RMI systems/libraries and provide users with the ability to exploit the best features of each. In particular, dynamic (re)selection of the best suited underlying RMI runtime is provided, thus enabling flexibility and efficiency without requiring additional programming, staging of stub and skeleton classes, or additional daemons.

The issues of universal connectivity and scenarios of dynamic protocol negotiation have previously been discussed in the context of the SoapRMI system [5], but to the best of our knowledge, a completely designed and implemented multiprotocol Java RMI system does not currently exist. In the research community, typical efforts to improve RMI begin with an analysis and investigation of RMI limitations and propose new, improved RMI frameworks [12, 9, 11]. KaRMI [12] is one such system that focuses on improving performance and offers some interesting capabilities. KaRMI is a drop-in replacement for standard Java RMI, written in pure Java and exploiting optimized serialization. KaRMI supports non-TCP/IP communication networks, e.g. Myrinet, to provide a very efficient tool for cluster-wide RMI computations. However, KaRMI is not interoperable with ordinary RMI applications and services. KaRMI is a mature system that has now evolved into a distributed framework called JavaParty [6].

There are also other approaches like Manta [22], that sacrifices Java portability and uses native code to achieve the best possible performance. Manta is a native Java compiler that compiles Java source code 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 requires additional modifications to execute in

new environments. Our proposed RMIX system is implemented in standard Java, therefore, although it minimally sacrifices performance, the full complement of Java portability features is retained.

Web Services [23] represent a general trend to simplify the integration and access of heterogeneous services on the Internet. One example of a commercial Web Service toolkit and hosting environment is Systinet WASP (Web Applications and Services Platform) [20]. 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 – which precludes its use in high performance applications.

When compared to the RMI model, Web Services present the user with a much lower level of abstraction for distributed computing. However, nothing prevents one from creating an RMI layer on top of Web Services, as exemplified by XSOAP [13], and as is further explored within the RMIX framework (see Section 4.2). Via such an approach, a simple and elegant RMI API can be provided for accessing heterogeneous resources over the Web.

3. RMIX Design and Implementation

The RMIX project strives to encompass various RMI protocol service providers within a single, unified framework. Such a problem definition leads to the major research question of how the framework's *interface contract* should be defined, i.e. common RMI semantics that providers are required to adhere to and clients can depend on. On the one hand, such semantics must be simple enough: (1) to be supportable even by non-sophisticated wire protocols; and (2) so that existing implementations may be ported to RMIX with small modifications. On the other hand, they must be powerful enough to allow clients to exploit the framework as seamlessly as possible, i.e. with minimal dependencies on actual provider implementations.

3.1. Invocation Model

To our advantage, most existing Java RMI implementations have adopted the base model and semantics defined by the native Java RMI specification [17]. Remote access is realized through client-side stubs of exported remote objects. Stubs implement *remote interfaces* that are identical to their target objects, and they forward invocations of *remote methods* to their target objects. Remote interfaces must extend `java.rmi.Remote`, and all of their methods must be declared to throw `java.rmi.RemoteException` to reflect potential communication failures. Due to the lack of shared address space, parameters of remote methods are passed by value, except for remote object references that are substituted on the wire by their stubs, being effectively passed by reference. In RMIX, these semantics must be fulfilled by all provider implementations.

3.2. Serialization

Different RMI implementations vary significantly with respect to serialization of parameters passed to remote methods. At one extreme, standard Java RMI/JRMP, based on the Java platform serialization, is able to handle any serializable Java object whose state may be otherwise inaccessible from outside. At the other extreme, some useful RMI implementations may depend on restricted encodings (like XDR [8]) imposing significant limitations on serialization. For maximum flexibility, RMIX mandates only that the following to be serializable by all protocol providers: primitive types, strings, arrays of primitive types, arrays of strings, remote references, and final classes with all fields being primitives or strings that additionally have *getter* and *setter* methods (conforming to the Java bean design pattern [16]) for all of these fields that are not public. In particular, providers are not required to support polymorphism and object graphs, although many of them will.

However, providers *are* required to support serializable remote references, including *alien* references, i.e. stubs created by other providers. This feature is essential for dynamic protocol switching: for instance, it is necessary to support the transmission of JRMP references over SOAP, and vice-versa. To aid in fulfilling this requirement without introducing explicit inter-provider dependencies, RMIX defines a *unified remote reference format*, that serves as a protocol switching *lingua franca*.

3.3. Registry

During the lifetime of a distributed application, remote references can be directly exchanged between a client and a server. However, a bootstrap mechanism is usually needed to establish first contact. The exact means of this mechanism varies. In the original RMI, it takes the form of the `rmiregistry` utility. Similarly, the notion of a naming service is used in CORBA environments. Web Services introduce a more versatile registration and lookup mechanism in terms of WSDL service descriptors [3], and WS-Inspection [7] documents grouping them along with additional metadata, UDDI registries [21], and more. In fact, even some out-of-band mechanisms (like e-mail, or voice telephony) may be used. Considering this variety, we believe that the choice of an appropriate strategy should be left to the application. Therefore, RMIX does not mandate any particular type of discovery mechanism. However, for backward compatibility and to provide some common ground, compliance with `rmiregistry` is provided,

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

i.e. application may store any RMIX remote reference in the `rmiregistry`. This feature exploits the aforementioned unified remote reference format, so it does not require any additional support from the providers.

3.4. Enhancements of Export Semantics

In standard Java RMI, a remote object has a single endpoint that exposes its complete functionality, i.e. it permits the invocation of any of the object's remote methods. In RMIX, remote objects may have multiple endpoints. Some of these endpoints may be independently supported by different RMI protocol providers.

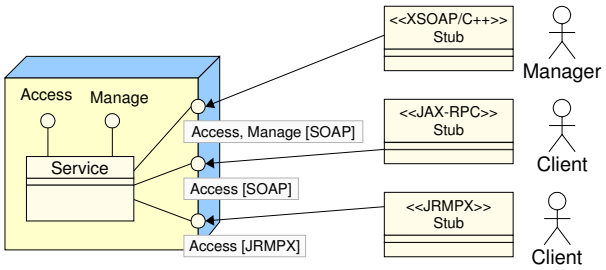

Figure 2. Customizable multiple endpoints

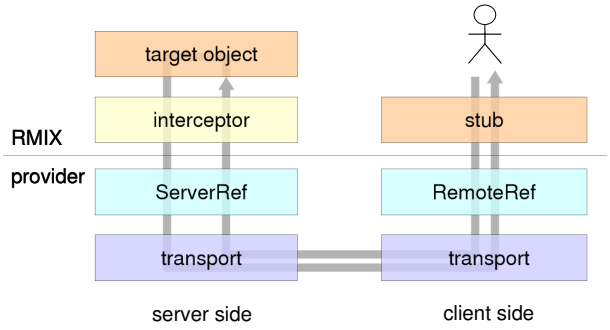

Figure 3. RMIX invocation stack

A common application-level usage scenario would be to publish the information about multiple endpoints within a registry service (e.g. in the form of a WSDL document) leaving the choice of a specific remote binding to clients. Even within a single protocol, RMIX allows the creation of multiple, customizable endpoints per remote object. Specifically, it is possible to restrict the set of exported remote interfaces on a per-endpoint (and effectively per-user) basis, as shown in Figure 2. Further, any endpoint may be associated with an application-defined *interceptor*, guarding remote method invocations on that endpoint, as shown in Figure 3. Common use of interceptors is to implement dynamic access control policies, e.g. to allow or deny method invocations depending on custom, dynamically changing criteria. Also, provider-specific customization is possible; e.g. different socket factories may be used for different endpoints. These enhancements (in conjunction with standard Java security APIs) aid the development of multi-user distributed applications, as they enable isolating authorization policies at the level of a remote endpoint, separating them from the application logic.

3.5. Remote Object Identities

In RMI, any two stubs pointing to the same remote object are equal in terms of the `equals` and `hashCode` methods. RMIX extends this feature even further, over independent protocol providers. This is achieved by introducing a common identification scheme based on globally unique identifiers (GUID) and derived from appropriate mechanics found in standard RMI. This feature aids in identification of remote aliases and it allows to store remote references in hash tables.

3.6. Deployment of RMIX Providers

RMIX unifies not only how different protocol providers are used, but also how they are installed and managed. Every protocol provider implementation must be bundled within a JAR file, with appropriate metadata stored in the manifest. To install a new RMIX provider, it is sufficient to place a JAR file containing the required provider classes in the appropriate directory. Depending on that placement and on the run-time options, this installation may be system-wide, user-wide, or application-wide. After installation, providers become visible to applications (including those currently running) via dynamic discovery.

4. JRMPX and XSOAP Providers

4.1. JRMPX

JRMPX is the RMIX transport protocol provider that maintains full Java RMI semantics, e.g. compliance with the Java serialization specification, support for class annotation, dynamic remote class loading, and distributed garbage collection. Support for these features have been realized in RMIX by building JRMPX *upon* the standard Java RMI (i.e. using it for tunneling remote calls). Such a design approach has contributed to the simplicity of JRMPX, which consists of fewer than 20 simple Java classes. The JRMPX transport does not depend on any non-public APIs, so it will remain compatible with upcoming releases of standard Java RMI. Furthermore, JRMPX will reflect any performance

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

improvements, new features, and bug fixes introduced into future release of Java RMI.

One drawback of this approach is that it introduces some invocation overhead. In our preliminary performance tests, this overhead ranges from 0.5% to 43% per invocation as compared to the standard RMI. Detailed performance results can be found in the companion paper [10]. However, it is important to note that JRMPX, similarly to standard Java RMI, is *not* aimed at performance-critical applications, trading performance for richer semantics [12]. We thus believe that the JRMPX overhead is acceptable whenever the standard RMI overhead is acceptable, and that performance-critical applications require specific, simpler but more efficient communication protocols.

4.2. XSOAP

The XSOAP toolkit (formerly SoapRMI [13]) is an RMI system implemented in Java and C++ that uses SOAP as its wire protocol. This toolkit, besides supporting standard SOAP 1.1 functionality, provides a convenient remote method invocation abstraction that works both in Java and C++. This permits the creation of SOAP based Web Services as easily as using the `UnicastRemoteObject` to export remote objects in Java RMI. In addition to the RMI API, every client and server in XSOAP has access to a SOAP-services module. This module encapsulates all the SOAP services including SOAP serialization and deserialization that any object may need, including access to lower level information and functions that are not exposed by the RMI API.

In order to transform XSOAP into an 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, 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.

Our preliminary experiments indicate that the communication performance of RMIX-XSOAP is very close to the original XSOAP toolkit, although they are both about an order of magnitude behind standard Java RMI. The poor performance characteristics of SOAP is a well-known issue [2], intrinsic to its XML ancestry that affects encoding/decoding time and inflates message sizes. However, because in part of that very ancestry, SOAP has become increasingly popular in loosely coupled heterogeneous systems, both in commercial and scientific domains [14, 4, 3]. For that reason, we consider SOAP to be a crucial component of any multi-protocol framework. In particular, RMIX-XSOAP enables Web Service hosting within the "Standard Edition" Java platform.

5. Migration to RMIX

The main benefit of migrating legacy RMI applications to RMIX is to enable the use of dynamic stubs and to remove dependency on the `rmic` tool. This may simplify code maintenance (as the need to distribute and synchronize stub classes between clients vanishes) and/or reduce startup overhead in applications that would otherwise load stub classes from the network. Additionally, an application migrated to RMIX may be able to exploit the best transport protocol available within a given domain, without a need to recompile the application itself.

In most cases, the migration only requires mechanical changes to the source files, usually only at the server side. These include replacement of `java.rmi.server.UnicastRemoteObject` with its counterpart from the RMIX package, and introduction of explicit invocations of the RMIX API for bootstrap purposes (i.e. within `Naming.bind()` calls). Even though RMIX supports multiple, dynamically created endpoints per remote object which leads to slightly modified export semantics, the RMIX version of `UnicastRemoteObject` emulates legacy behavior (associating a remote object with a single, default endpoint), so that the above substitutions are usually sufficient. Manual corrections may be required in cases when the application explicitly depends on the inheritance hierarchy (e.g. relying on the fact that a given remote object is an instance of `java.rmi.server.RemoteObject`), although RMIX tries to mimic the legacy hierarchy to minimize that impact. Also, problems may arise if an application depends on a new policy (introduced in J2SE 1.3) of sending remote objects by value if they have not been exported. RMIX always sends remote objects by reference, exporting them dynamically if required. However, we believe that depending on the new policy (i.e. attempting to send remote objects by value) is at best questionable practice anyway.

6. Conclusions and Future Work

In this paper, the new communication framework for Java has been described. The framework, called RMIX, is an incarnation of the Remote Method Invocation paradigm that enhances capabilities of standard Java RMI. In particular, RMIX enables simultaneous use of multiple method invocation protocols, handled independently by pluggable and dynamically managed service provider modules. It has been shown how interoperability between different protocol providers 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 protocols. Additionally, RMIX introduces enhancements to the RMI model that enable customization of remote object endpoints. The paper argues that the RMIX framework simplifies development of various kinds of distributed applications, including those that need to accommodate different peers over different protocols, if protocol independence is to be achieved, or if dynamic protocol negotiation features are desired. The transition path for legacy RMI applications towards RMIX has been described, along with emerging benefits of such transition. Features of two supplied transport service providers has been outlined.

We are currently working on several improvements to RMIX. Some aspects related specifically to interoperability are discussed in the companion paper [10]. In terms of architectural improvements, the possibilities of further abstracting applications away from protocol provider implementations are investigated. The proposed solution is to introduce per-provider *properties* and *RMI conformance levels*, enabling description of capabilities in a generalized form. This would allow applications to select providers based on application-specific requirements (that may exceed least-common-denominator guarantees) without introducing explicit dependencies. Additionally, the ongoing work focuses on providing support for message-level security via APIs and implementations realizing message encryption and digital signing.

References

[1] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/SOAP/`.

[2] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 246–254, Edinburgh, Scotland, 2002. Available at `http://www.extreme.indiana.edu/~mgovinda/research/papers/soap-hpdc2002.pdf`.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl/`.

[4] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the Grid: An Open Grid Services Architecture for distributed systems integration, Jan. 2002. Available at `http://www.globus.org/research/papers/ogsa.pdf`.

[5] 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/`.

[6] B. Haumacher and M. Philippsen. JavaParty. `http://www.ipd.uka.de/JavaParty/`.

[7] IBM, Microsoft. Web Services Inspection Language (WS-Inspection). `http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html?dwzone=webservices`.

[8] Internet Engineering Task Force. ONC Remote Procedure Call. `http://www.ietf.org/html.charters/oncrpc-charter.html`.

[9] 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.

[10] D. Kurzyniec, T. Wrzosek, and V. Sunderam. Heterogeneous access to service-based distributed computing: the RMIX approach. In *12th International Heterogeneous Computing Workshop*, Nice, France, Apr. 2002.

[11] 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.

[12] 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`.

[13] 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/`.

[14] 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.

[15] Sun Microsystems. Java API for XML-based RPC. `http://java.sun.com/xml/jaxrpc/`.

[16] Sun Microsystems. Java Beans. `http://java.sun.com/products/javabeans/docs/beans.101.pdf`.

[17] Sun Microsystems. Java Remote Method Invocation specification. `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`.

[18] Sun Microsystems. Java RMI over IIOP. `http://java.sun.com/products/rmi-iiop/`.

[19] V. Sunderam and D. Kurzyniec. Lightweight self-organizing frameworks for metacomputing. In *The 11th International Symposium on High Performance Distributed Computing*, pages 113–122, Edinburgh, Scotland, July 2002.

[20] Systinet. WASP (web applications and services platform). `http://www.systinet.com/`.

[21] Universal Description, Discovery and Integration (UDDI). `http://www.uddi.org`.

[22] 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.

[23] V. Vasudevan. A Web Services primer. `http://www.xml.com/pub/a/2001/04/04/webservices/`.