A RMI-Security-Extension using the *PERMI* framework

Sven Schulz Michael Friedrich Wolfgang Küchlin
Wilhelm-Schickard Institute for Computer Science
Symbolic Computation Group
University of Tübingen
{schulzs,friedrich,kuechlin}@informatik.uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

Tobias Hüttner
MagicMaps GmbH
D-72124 Pliezhausen
Tobias.Huettner@magicmaps.de
http://www.magicmaps.de

Abstract. This work describes a method to make the Java RMI subsystem secure in heterogenous and distributed systems. The approach is based on conserving the access control context of the Java Security Architecture. Our framework uses pure Java functionality only and can be integrated unobtrusively into existent server applications.

1 Introduction

The Java Remote Method Invocation (RMI) API shields the developer from the details of communication within distributed systems by introducing an easy-to-use remote procedure call, allowing to concentrate on application specific code. Customizing RMI behavior was no central issue when RMI was designed, which makes adjustments to real-life applications at least hard, often impossible. However, functionality that is orthogonal to the application, like security, logging, accounting, and fault tolerance, has to be separated from application code to prevent unnecessary complexity on top of the application's intrinsic complexity.

In this paper, we discuss a mechanism to make RMI more flexible while maintaining RMI's API, thereby requiring no changes to client application code and negligible changes to server code. On top of that, we present a technique to preserve method invocation context across *Java Virtual Machine* (JVM) borders, thus harmonizing remote and local method invocation in terms of security aspects.

The rest of this section covers a description of how the results presented in this paper are used in a real-life business context (Section 1.1) and the objectives for *PERMI*. Section 2 presents an introduction to RMI (Section 2.1), and a short description of the Dynamic Proxy Classes API introduced in Java 2 Version 1.3, which is used later on (Section 2.2) and our approach to customizing RMI behavior, including an in-depth look at how our technique works. In Section 3, we introduce a technique to preserve context information between client and

Fig 1. The car display with the current location. Here: the center of Munich.

server on remote method invocation. After presenting the related work in Section 4, a closing discussion is given in Section 5.

1.1 Commercial Application Context

The work presented in this paper supports the distribution of a *rendering on demand* (ROD) computer system. The ROD service renders a 3D landscape view of a position anywhere in Germany with a given camera position and direction. These views can be displayed in browser windows on PCs, PDAs or smartphones. This service is provided by the MagicMaps company for the BMW 7 series cars as part of the built-in iDrive system (see Figure 1). The camera position is acquired by the GPS (global positioning system) receiver, and the 3D landscape view contains current local weather information.[1]

Rendering these views is very data intensive. The 3D topological model and its texture consists of large chunks of data even for a rather small geographical region. Unfortunately, storing this data in secondary storage like hard disk arrays is too slow to guarantee the required response time. Therefore, the data is split up into 49 geographical regions of Germany and 33 major cities.

The ROD service uses Java RMI as communication middleware. Each incoming request is analyzed in the web-portal and then delivered to the rendering service which in turn passes it on to the rendering node according to the geographical region. This system can be easily extended when more requests have to be handled or additional geographical regions are supported. In this case, the local hardware resources can be a limiting factor and a number of nodes have to be provided by external data processing centers.

RMI security is a vital issue when leaving the protected intranet. The work presented in this paper gives a solution to make the RMI middleware secure, allowing the heterogenous distribution of parts of the ROD system.

[1] for an example, see: http://alps3d.dyndns.org/intergeo/index.html

1.2 Objectives

The extension to Java RMI is designed to satisfy the commercial context above and therefore fulfil the following objectives:

- **Customization of RMI remote object behavior -** In standard RMI, there is no way to customize the behavior of a remote stub, which is determined by the methods of the abstract class `java.rmi.server.RemoteStub`. In our approach, *smart proxies*, i.e. proxies whose functionality exceeds simple invocation forwarding, can be attached to a stub in a pluggable and stackable manner. These proxies might reside in the server VM as well as accompanying the stub as it is serialized and transferred to the client VM. In this way, it is possible to transparently customize every aspect of remote object behavior even without recompiling the contributing classes.

- **Reduce complexity of server code -** Generally source code of C/S-Systems is made up of many intermixing parts with diverse purposes. Aspects to consider are security issues, failure and connection management, logging, caching and implementation of special remote semantics. By the modular approach provided by *PERMI*, system architects can split up complex systems into specialized parts of manageable size, where each aspect is treated by a dedicated module injected into the RMI stream.

- **Compatibility -** For the acceptance of a framework like *PERMI* its compliance with established standards is important. Because application developers do not want to change their habits of programming, we took great care to make *PERMI* compatible to standard RMI on the interface level. Additionally, an approach touching system classes would hamper application deployment and may not be deployed at all according to the license agreements of VM vendors. Hence, an application level solution, like *PERMI*, is preferable. Furthermore, no modification of legacy applications is necessary, if no extra functionality provided by *PERMI* is desired.

2 *Pluggable Extension to RMI*

In this section, we present our *Pluggable Extension to RMI* (*PERMI*), an extension of Java standard RMI. Our approach is based on a technique presented by Rickard Öberg in a contribution to the RMI Mailing List hosted by Sun Microsystems (see [6, 5]). By replacing stubs, generated by Sun's RMI Compiler (RMIC), with generic stubs created at runtime, we are able to customize an object's remote behavior dynamically, on both the client- and on the server-side. At first, we describe the foundation of *PERMI*, the basics of RMI and dynamic proxies, then *PERMI*'s architecture, and finally its implementation.

2.1 RMI Overview

Remote Method Invocation (RMI) [7] [12] adds the power and flexibility of remote procedure calls (RPC) to Java. It provides a framework within which

Java objects in distinct Java Virtual Machines (JVMs) can interact in a way which preserves the object-oriented paradigms of Java.

Both RMI and RPC are based on client-side and server-side stubs which encapsulate the remote call. They are generated at compile-time. Each client application has a local copy of the client stub. In RPC, the client stubs are distributed manually whereas in RMI, they are stored within the registry and transferred automatically. When the client application performs a lookup of a service registered at the RMI registry it gets the address of this service as well as the required client-side stub. Therefore, the distribution of client-side stubs is independent of the client application which allows modifications of the stubs after the distribution of the client application. Furthermore, additional functionality can be injected into the communication stream.

This behavior is related to the Unix System V streams where *line disciplines* can be injected [1]. These line disciplines are modules which transform a stream from a process to a device driver and vice versa. Both in streams and RMI, this behavior is used to adapt the middleware to special needs. We instrument this feature for security concerns in our approach.

Additionally, RMI offers sophisticated method parameter handling, which allows different RMI object semantics depending on class anatomy. Either call by reference or call by value can be used. Furthermore, methods can even pass objects that the remote virtual machine has never encountered before by allowing the dynamic loading of unknown classes.

2.2 Dynamic Proxy Classes

Our approach is based on the *dynamic proxy classes* API [11] which Sun Microsystems introduced with JDK 1.3. A *dynamic proxy class* is a class that implements a given set of interfaces specified at runtime when the class is created. A method invocation through one of these interfaces of an instance of the dynamic proxy class is delegated to its invocation handler. The method call is encoded by a `java.lang.reflect.Method` object and an array of type `Object` containing the parameters of the invocation and this encoding is passed to a single method of the instance's invocation handler. Like pre-generated classes created with compile-time tools, *dynamic proxy classes* are type-safe in such a way, that the expression

```
proxy instanceof SomeInterface
```

yields true, if and only if `SomeInterface` or a derived interface was contained in the interface list of the constructor when `proxy` was created. Besides the interfaces to be implemented by a dynamic proxy class, one can specify a class loader, which is used to define and resolve the proxy class. This is convenient when we have to replace RMIC generated stubs during remote object export. Dynamic Proxy Class API usage is summarized in Figure 2.

2.3 The *PERMI* Concept

The basic idea of our proposal is to replace stubs created with RMIC at compile time with dynamically created stubs at runtime. Additionally, smart proxies

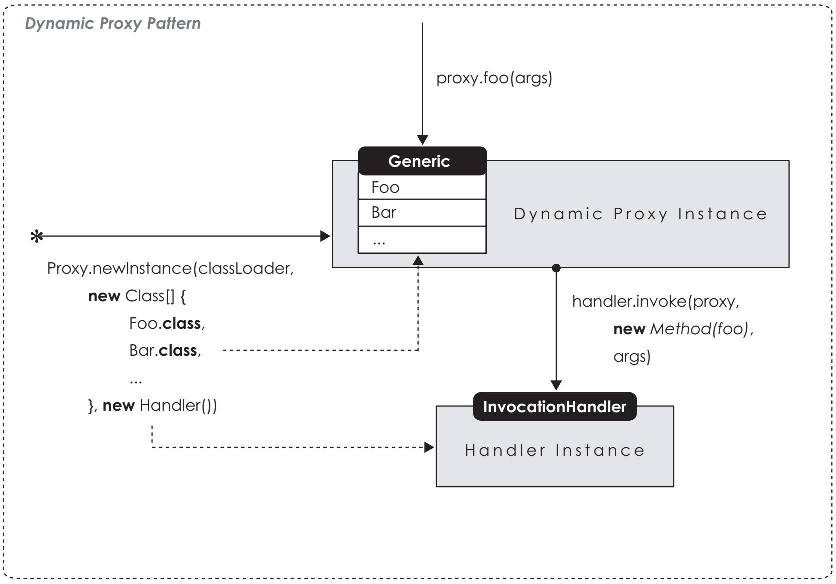


Fig 2. Usage of the Dynamic Proxy Classes API. An instance of a dynamic proxy implements all given interfaces and delegates method calls to the InvocationHandler.

represented by Dynamic Proxy Classes can be attached to remote objects in a pluggable and stackable manner. A C/S-System's sample configuration resulting from attaching proxies with *PERMI* is depicted in Figure 3.

With *PERMI*, we provide a framework and an API to simplify proxy creation, proxy attachment to remote objects, facilities to export and unexport remote objects to and from the RMI System, and to establish application level transparency on the client- and to some extent on the server-side.

2.4 Architectural Approach

In order to implement the functionality outlined above, we add another layer, the proxy management layer, on top of the RMI protocol stack (see Figure 4). Proxy management layer functionality is realized through implementations of a so called `Exporter` interface, which serves as an abstraction of the *PERMI* exporting process. This process surrounds standard RMI exporting of remote objects with special actions necessary to deploy its associated proxy instances.

2.5 Implementation

To explain how *PERMI* works, we describe below the main actions performed on remote objects during their lifetime. This includes exporting a remote object, the processes of binding and looking up stubs through a directory service, and the way in which proxies are attached to a remote object.

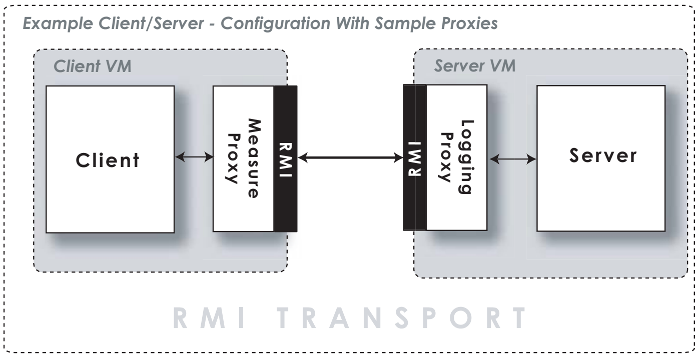


Fig 3. C/S-System composed of stacked smart proxies between regular client and server.

2.5.1 Object Export

When exporting an object RMI consults its class loader to find the associated stub class by invoking the class loader's `findClass` method with the class name concatenated with `_Stub` as parameter. The RMI subsystem can be forced to use another class instead of the precompiled stub class by providing a special class loader. In order to associate another class loader with a class, we have to ensure that it is loaded initially by the new class loader rather than the system class loader which is usually consulted. The method `findClass` of our custom class loader first checks, whether a requested class is suffixed by `_Stub`. In this case a special subclass of `java.rmi.server.RemoteStub` is returned and all the remote interfaces of the original class specified by the name without the `_Stub` suffix are saved to a `ThreadLocal` variable to be accessed later. Otherwise the request is forwarded to the parent class loader.

The stub class determined by `findClass` is instantiated by RMI and returned to the caller of `UnicastRemoteObject`'s `exportObject` method. In case of our special `RemoteStub` the previously saved remote interfaces implemented by the exported object are stored in an instance member field during construction.

2.5.2 Object Lookup

An exported RMI object is usually bound to some kind of *directory service*, which can be used by clients to ask for stubs by specifying decorations which were associated with the stub when it was bound to the directory. The RMI REGISTRY, as a very simple representative of a directory service, associates stubs with plain text names. In *PERMI*, the `RemoteStub` returned by `exportObject` has to be bound in contrast to standard RMI, where the remote object itself must be bound. When a client requests an instance of a stub, it is serialized, transmitted to the client, and deserialized. In fact, the Registry itself uses RMI to serve binding and lookup requests. When a serializable object is deserialized, its class is first checked for implementing a method called `readResolve`, which can be used to replace the object normally created via deserialization

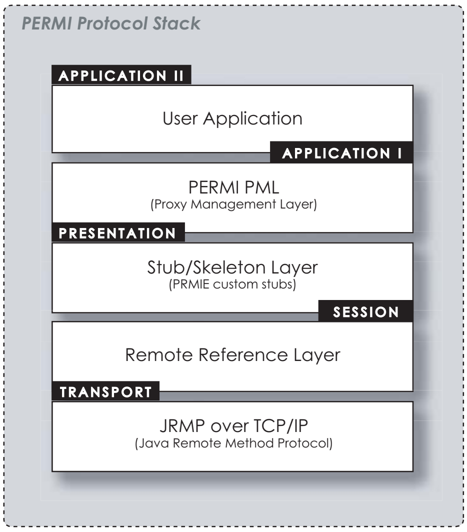


Fig 4. *PERMI*'s additional Proxy Management Layer on top of standard the RMI protocol stack

with a custom object. In our case, this custom object is a dynamic remote proxy created within `readResolve` implementing the remote interfaces stored in the `RemoteStub` instance's member field. This proxy class dispatches method invocations made by the client as usual to its invocation handler, represented by an instance of a special handler class constructed with the remote reference which has been stored likewise in a member field of our remote stub. This handler class is responsible for RMI internal processing of method invocations on the enclosed remote reference normally done by the RMIC generated stub, e.g. creating a special method hash code used to encode method invocations.

2.5.3 Attaching additional proxies

Our system objectives claim the possibility to attach custom proxies to a remote object, which may reside in both client and server VMs, intercepting method invocations and process them on their behalf before dispatching them on the next proxy, respectively the server object itself. This is achieved by wrapping dynamic proxies around the remote object, whereas the position relative to the exporting statement is the decisive factor for the proxies position within the execution sequence during the processing of a remote method call. Wrapping a proxy around the not yet exported remote object makes it reside in the server VM, while wrapping after exporting the remote object leads to a migration into the client VM with the intermediate steps of binding and lookup over the registry described in the last section. For that purpose it comes in handy that instances of a dynamic proxy class are serializable as long as their invocation handler is. Thus, handling of wrapped proxies turns out to be straightforward, because they are serialized with the enclosed remote object, or remote stub, respectively.

2.5.4 Pluggable RMI API

We already provided an in-depth description of how *PERMI* works under the hood, but the internal details are hidden behind a simple API, making the use of *PERMI* just as easy as that of RMI. In fact, *PERMI* is designed to show nearly no difference between RMI and *PERMI* code.

The `RemoteObject` class is a central structure to *PERMI*, which provides methods to attach smart proxies in a standardized way. Export of an instance of this class is done by static methods of `UnicastRemoteObject`, which is assisted by an `UnicastExporter`, an implementation of the more general `Exporter` interface. The `UnicastExporter` takes care of the necessary steps to properly export the object, i.e. attaching the server-side proxies, exporting the object using `UnicastRemoteObject`'s `export` method, and finally attaching the client side proxies.

As described in Section 2.5.1, the object to be exported must be loaded by our special class loader to ensure that a generic stub is deployed. If a server side proxy has been attached using `RemoteObject`'s corresponding method, this condition is fulfilled by defining the dynamic proxy class using our special class loader. Otherwise the responsible `Exporter` wraps a simple delegation proxy around the object just before it is exported.

3 *Secure Extension to RMI*

In this section, the *Secure Extension to RMI* (*SERMI*) is described in detail. Given the *SERMI* framework, the application described in Section 1.1 can be made secure. Actually, this framework is applicable to all RMI-based Client-Server-Applications. In the following, we describe what is aggregated in an invocation context, delineate various naive approaches and finally present our *PERMI* based solution.

3.1 Anatomy of the Java Method Invocation Context

Every method invocation is processed within an invocation context maintained by the executing VM. The information contained in this invocation context is central to Java's security checking concepts. In case of a remote method invocation, there are at least two VMs involved. Nevertheless, standard RMI turns out to be incapable of dealing with the split up invocation context between client and server VM. As a result, Java's fundamental access control algorithm is partially disabled.

In Java, the context in which a method invocation takes place is composed of several parts (cf. Figure 5):

- The *stack frame execution context* (EC) is made up of the sequence of method calls leading to the current one, with each method call represented by a stack frame composed of the method signature, its declaring class and, if available, the line number within the declaring source file. An EC, also known as a *stack trace*, can be acquired by calling `java.lang.Throwable`'s `getStackTrace` method.

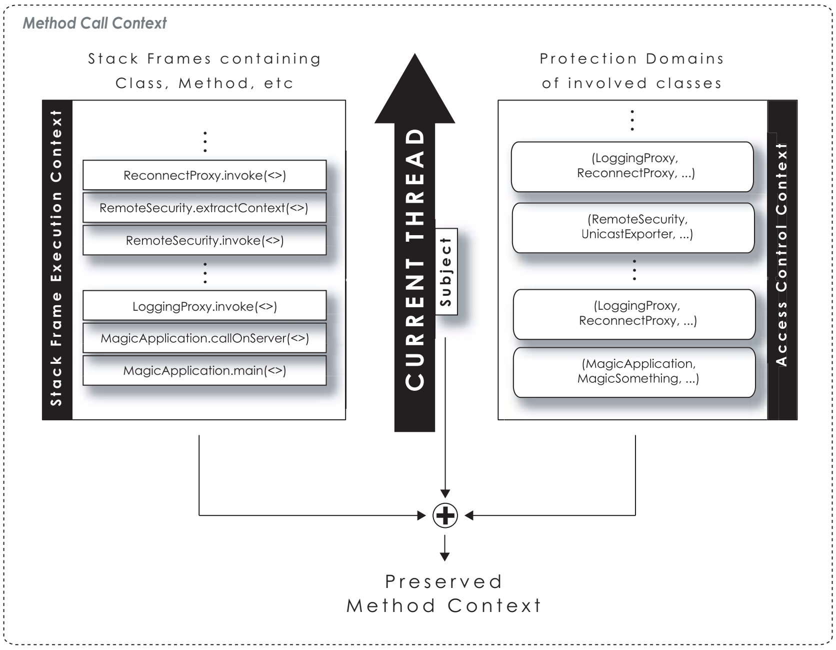


Fig 5. Information composing the Java method invocation context

- The *access control context* (ACC) contains the ordered sequence of involved protection domains. A protection domain is involved, if and only if a class defined within was called during execution. The order of the entries is determined by their position in the EC. Java's fundamental access control algorithm works on the ACC to serve access control requests made through an invocation of `AccessController`'s static method `checkPermission` and can be acquired through its static `getContext` method.
- Since version 1.4, the Java Authentication and Authorization Service (JAAS) is part of Sun's JDK. By associating a subject to the current thread of execution, it permits the enactment of subject-based policies.

3.2 Unsophisticated Approaches

Knowing about the threats arising from even partially disabling Java's security mechanisms, one may decide to ignore the fact that RMI does not preserve context information across VM borders. As a result, clientside code is never checked for the appropriate rights when a security check is performed on the server side during a remote method invocation. This is an implicit application of the concept of privileged actions implemented in the Java 2 Security Architecture (J2SA). As with privileged actions, improper use might cause significant security threats (for details see [3], pp. 95-106). Besides complications arising from the

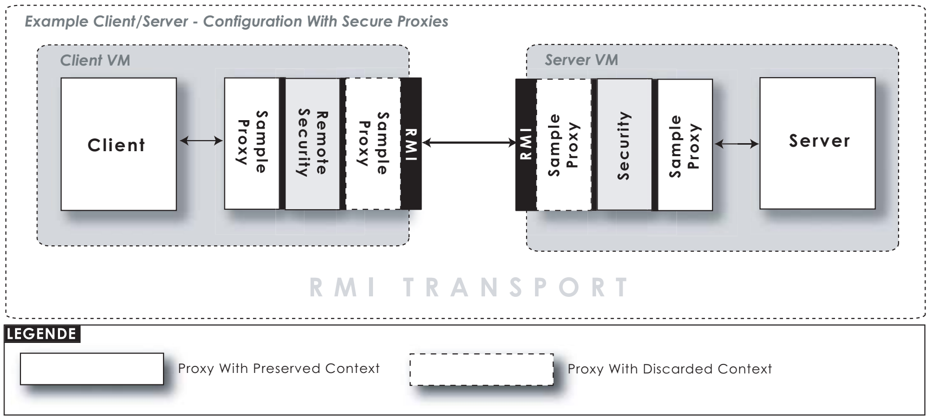


Fig 6. Invocation context propagation through smart proxies

improper use of privileged actions, the lack of unprivileged method calls hampers secure programming at all.

Another simple approach to deal with security issues when using RMI is to incorporate security related information into the API of remote objects. When making RMI calls over a secure socket implementation provided by the JAVA SECURE SOCKET EXTENSION (JSSE), information describing protection domains involved in a method call and the subject on whose behalf the method is invoked could be included in the signature of sensitive methods, thus enabling the restoration of the invocation context prior to the execution of critical actions. Although this approach serves the purpose of retrofitting Java's basic access control algorithm, it's appliance is error-prone and places a burden on application developers.

3.3 *PERMI* based solution

Our implementation of secure method invocation using *PERMI* is quite straightforward. As depicted in Figure 6, our solution depends on injecting a client- and a server-side proxy into the proxy chain between client and server. In the course of a remote method invocation, the current invocation context is extracted and passed on as a hidden argument to succeeding proxies by an instance of `RemoteSecurity` residing in the client-side VM. As control flow reaches the server-side proxy, i.e. an instance of the `Security` class, the encoded client-side invocation context is restored and merged with the server-side context. Both client and server code are unaware of the additionally performed context propagation.

3.3.1 Technical Aspects

The current ACC can be acquired using `AccessController`'s static `getContext` method. Unfortunately, `AccessControlContext` is neither declared serializable nor can protection domains contained within be determined through regular method invocation. Therefore another approach has to be applied to extract context information from the ACC. Firstly, we use reflection to access the private field containing the array of protection domains. For that purpose special permissions must be granted to the protection domain containing the `RemoteSecurity` class. To restrict the number of classes to grant these critical permissions, the reflective access is enclosed within a privileged action. Regarding the fact that no direct natural matching between protection domains in different VMs is possible, we have to match classes instead of protection domains.

Within the J2SA there is an asymmetric association between protection domains and classes. Starting from a class object, its protection domain can be easily determined by calling `Class`' `getProtectionDomain` method. Regrettably, it is not possible to do it the other way round. Our approach uses the EC accessible through `java.lang.Throwable`'s `getStackTrace` method to determine classes involved in the method invocation context. Due to privileged actions potentially included within the call stack, a match between EC and ACC entries must be performed. If a protection domain of a class contained in an EC entry is also represented in the list of protection domains extracted from the ACC, the class is considered necessary to restore the invocation context on the server side. Class matching between different VMs is an open topic. Our current implementation distinguishes between classes based on their fully qualified class names. Other approaches using version information and code origin might be considered in future releases.

The resulting classes are stored along with the subject associated with the current thread within an instance of `InvocationContext`, which is forwarded as a hidden argument to succeeding proxies, specifically the server-side `Security` instance.

In order to restore the invocation context, the `Security` instance determines the protection domain for each class and associates the information with the current thread using `Subject`'s `doAsPrivileged` method. To restore the invocation context an ACC containing the subject and the protection domains described by the classes within the `InvocationContext` received from the client-side VM has to be created. Up to now, classes which aren't available in the server-side VM are discarded and therefore not included in the restored ACC. Approaches using class annotation similar to RMI class annotation are being evaluated for further releases.

3.3.2 Performance Enhancements

The performance of our solution is determined mostly by the speed of context processing within `RemoteSecurity` and `Security`, and by the amount of extra information passed using serialization between them due to invocation context preservation. Both, context processing and data transmission scales with the average depth of the execution stack. Although we have no data for the average depth of the execution stack, common applications were examined concerning maximum stack depth [2]. It seems that even large and complex applications like `javac` show maximum stack depths of about 100. Since the number of different execution paths grows exponentially with code size and assuming that the common rule of thumb holds that 90 percent of the execution time is caused by 10 percent of the code, we may assume that the average stack depth is rather small and ECs split up into rather few classes with high similarity, if execution time maps to execution path coverage. However, costs of context processing and transmission are likely to be high in comparison to the raw costs of a remote method invocation. Therefore, several optimizations were implemented to reduce the overhead imposed by *SERMI*.

First of all, *SERMI* can be configured to apply context preservation to different extents from none over subject-only to full-blown context preservation for each method of the remote interface separately. That kind of fine-grained control assists efforts to reduce costs, by applying context preservation only to those methods that include critical operations.

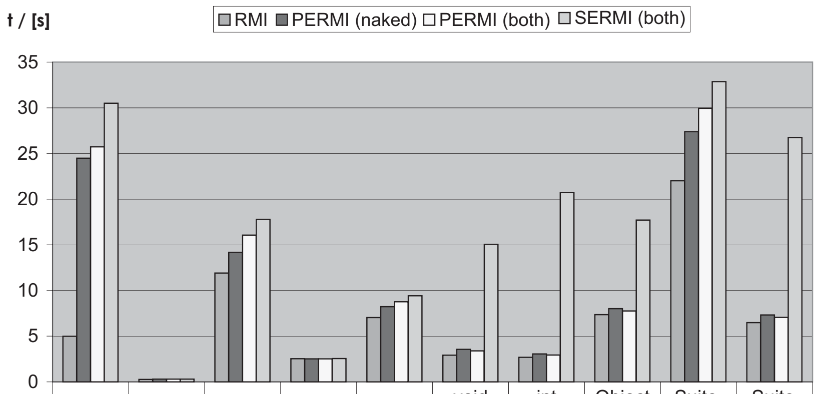
Additionally, overhead can be eliminated by reducing the number of redundant stack frames within the EC. Techniques to remove duplicates and resolve recursion are implemented. Furthermore system classes, i.e. classes loaded by the system class loader from the local filesystem, can be encoded with a compact identifier. Enhanced protocols may also use some sort of entropy encoding to further decrease the amount of transmitted data and bypass performance weaknesses imposed by Java's serialization protocol by providing a single optimized representation. Beside these efforts, we have implemented caching on both the `InvocationContextEntry` and the `InvocationContext` level to minimize overhead by only sending each relevant object one time.

3.4 Empirical Results

To determine the costs of integrating user customizable behavior into RMI on the one hand, and to transparently secure remote method calls on the other hand, we performed different synthetic benchmarks on RMI and our extensions *PERMI* and *SERMI* . The results are shown in Figure 7.

Our benchmarks include administrative actions, i.e. exporting and unexporting RMI respectively *PERMI* and *SERMI* objects to the runtime system, and performing bind, unbind and lookup operations on a standard RMI Registry, and executive actions, i.e. remote method invocations for different types of parameters and return values. Finally, the actions are combined to an administrative suite (export, bind, unbind, unexport) and an executive suite (lookup and multiple remote method callls). The benchmarks were performed on a Dual-Celeron-466 system with 256 MB of Main Memory connected with a 66 MHz Frontside-Bus running Sun's JVM 1.4.

The results for *PERMI* are quite impressive showing nearly equal performance to standard RMI in both administrative and executive sections. As expected *SERMI* clearly falls behind *PERMI* and RMI due to the impact of context extraction and restoration. This effect might even become more dramatic when the bandwidth of a network connection is limited. This behavior will fall off with increasing method calls performed on the same object, because the optimizations described in Section 3.3.2 come into effect. These efforts to enhance *SERMI*–Performance produced an effective speedup of 250% in the executive section of our benchmarks. Using *SERMI* with subject-only context preservation yields identical performance as *PERMI* does. The possibility to define the extent of context preservation on a per method basis might be helpful to application de-



	export [10000]	unexport [10000]	bind [1000]	unbind [1000]	lookup [1000]	void call(void) [2000]	int call(int) [2000]	Object call(Object)	Suite (admin.) [1000]	Suite (exec.) [1000]
RMI	4,986	0,271	11,906	2,538	7,048	2,932	2,687	7,360	22,018	6,487
PERMI (naked)	24,475	0,295	14,175	2,516	8,229	3,556	3,064	8,018	27,396	7,319
PERMI (both)	25,719	0,303	16,068	2,514	8,766	3,395	2,935	7,770	29,950	7,050
SERMI (both)	30,501	0,304	17,799	2,552	9,427	15,059	20,719	17,718	32,859	26,748

Fig 7. Differences in execution time between standard RMI, *PERMI* without any additional proxies (naked), *PERMI* with additional proxies with void functionality on client and on server side (both), and *SERMI* . Values in brackets denote the number of repetitions for each call.

velopers. This feature is not implemented yet, but can be easily integrated into our system.

4 Related Work

In this section we discuss other solutions to both customizing RMI behavior and securing remote method invocations. At first, we will have a look at alternative ways of customizing RMI behavior, then we describe recent activities on securing RMI.

4.1 Customizing RMI behavior

Loton [4] presents an approach that, along with many other approaches not discussed here, uses stub-rewriting by hand to integrate smart proxies with RMI. In contrast to our solution, advanced concepts like proxy chaining or dynamic proxy creation at runtime are not supported. Thus, the separation of application specific and orthogonal code is not supported. Additionally, the RMIC processing of classes containing remote objects is still necessary, as stub classes to be modified must be created preceding to application deployment.

In [13] RMI's handling of parameters is exploited. As exported remote objects are passed by reference, while not exported remote objects and those not derived from `RemoteObject` are passed by value, those parts of a chain of objects that are serializable will be transferred to the client, while remote objects will reside in the server-side VM. By introducing the stub created with RMIC conveniently into such a chain, remote method calls can be propagated along the elements of the chain. Depending on another exploit of RMI behavior, this approach allows the same possibilities for stacking smart proxies as *PERMI* does.

4.2 Securing Remote Method Invocation

In late 2002 Santos et al. presented a framework for integrating smart proxies and interceptors with RMI (see [8]). Although not explicitly stated, they adopted the approach described by Wilson [13] in conjunction with dynamic proxy classes. As a practical example they show how to use their framework to implement user authentication and fine-grained access control, which is close to our solution restricted to subject-only context preservation. ACC preservation however, is not addressed by their implementation.

With RMI Security for J2SE (registered as JCP 76, see [9]), an application was submitted to the JAVA COMMUNITY PROCESS (JCP) that tries to define a high-level API for controlling network security in RMI. The API should have provided mechanisms to plug different algorithms and protocols underneath applications without the need to recode them. The topics covered were authentication, confidentiality and integrity. In comparison to *SERMI*, which assures security at the lowest possible level, these aspects are much less fundamental and are built rather on top of RMI, depending on a model of trust between authenticated parties. In early 2001, the proposal was rejected after being reviewed by the community.

However, the project was not abandoned, but migrated to the Jini(TM) platform, being included in the more comprehensive DAVIS project (see [10]). While still not being approved by the JINI COMMUNITY DECISION PROCESS (JDP) and implementations still remaining in beta status, Davis provides security in the presence of dynamically downloaded code, constraints to control method invocations, mechanisms for building and deploying configurable applications and various abstractions for RMI specific processes like exporting of remote objects and secure class loading. The part of DAVIS closest to *PERMI* is the JINI EXTENSIBLE REMOTE INVOCATION (Jini ERI) allowing to customize each layer of the RMI protocol stack per exported object. Security aspects touched by the DAVIS project are the same as those covered by RMI Security described above.

5 Conclusion

We presented a generic framework called *Pluggable Extension to RMI* (*PERMI*) to enhance standard Java RMI. This framework was implemented in pure Java without any change of the standard JVM. A *Secure Extension to RMI* (*SERMI*) which preserves the access control context of the whole call stack was described on top of *PERMI*. The details of plugging so called *Smart Proxies* in the call stack of remote objects was illustrated in detail.

The key benefits of this architecture are the seamless integration of security into existing RMI server applications. Furthermore, the framework allows a fine grained distribution of application logic. Given these features, the performance of an application can be tuned to compensate for the security overhead.

Several issues are being addressed right now. Among these are performance, distribution of policies and caching.

References

[1] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[2] L. Fox. Measuring maximum stack frame count. Available at: http://www.cs.wustl.edu/ doc/RandD/ITR/Experiments/StackFrame/.

[3] L. Gong. *Inside Java 2 Platform Security - Architecture, API Design, and Implementation*. The Java Series. Addison-Wesley, 2nd edition, October 1999.

[4] T. Loton. The smart approach to distributed performance monitoring with Java. *Java World*, September 2000. Available at: http://www.javaworld.com/javaworld/jw-09-2000/jw-0901-smart.html.

[5] R. Öberg. Exporting remote objects with dynamic proxies: simplified. Available at: http://swjscmail1.java.sun.com/rmi-users.html.

[6] R. Öberg. Howto: Use dynamic proxies as rmi stubs + howto: Export dynamic. Available at: http://swjscmail1.java.sun.com/rmi-users.html.

[7] E. Pitt and K. McNiff. *java.rmi - The Remote Method Invocation Guide*. Addision-Wesley, 2001.

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

[9] B. Scheifler. JCP 76 - RMI Security. Technical report, Java Community Process (JCP), 2000. Available at: http://jcp.org/en/jsr/detail?id=076.

[10] B. Scheifler. Jini(TM) extensible remote invocation (Davis). Technical report, Jini Decision Process (JDP), 2003. Available at: http://davis.jini.org/proposals.html.

[11] Sun Microsystems. *Dynamic Proxy Classes*. Available at: http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[12] Sun Microsystems. *Fundamentals of RMI*. Available at: http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html.

[13] M. J. Wilson. Get smart with proxies and RMI. *Java World*, November 2000. Available at: http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-smartproxy-p3.html.