

Using the Java Sandbox for Resource Control

Almut Herzog and Nahid Shahmehri

Department of Computer and Information Science, Linköpings universitet,
SE-581 83 Linköping, Sweden
{almhe, nahsh}@ida.liu.se

Abstract. Java’s security architecture is well known for not taking the security aspect of availability into account. This has been recognised and addressed by a number of researchers and communities. However, in their suggested resource-aware Java environments, policies for resource control have so far been stated in proprietary, sometimes hard-coded, or undocumented ways. We set out to investigate if standard Java permission syntax can be used to formulate policies for resource management of high-level resources and if the enforcement of resource policies can successfully be done by the standard Java access controller. Such a solution would neatly fit in the existing Java security architecture.

We have implemented resource control for the serial port and for the file system by using the Java permission syntax for stating policies and the standard Java access controller as the enforcement mechanism. The implementation was straightforward and resulted in an API useful also for control of other high-level resources than the serial port and file system. A performance test showed that such resource management easily leads to excessive invocations of the access controller and that optimisation steps are necessary to prevent performance penalties.

Keywords. Java; Availability; Resource Management; Policy; Performance.

1 Introduction

In comparison to other programming languages, Java has good architectural support for access control to resources. While the CPU is under the hard-coded control of the Java virtual machine thread scheduler, and memory is allocated through object allocation and deallocated by the garbage collector, initial access to higher-level resources is granted by an optional security manager. Higher-level resources are e.g. the file system, I/O device APIs (application programmer interface), threads, sockets or properties. The security manager can be switched on for applications and provides a so-called sandbox that consults a policy—based on the contents of system policy files and user policy files—prior to granting access to resources.

Despite the support for initial access control, the Java sandbox lacks control over the extent to which a resource can be used. This has been recognised in multiple sources, e.g. [1][2][3]. Once access to a resource has been granted to a piece of code, the code is free to use it to any extent. For example, if a Java application is allowed to

write in `/tmp`, there is no control over how many bytes it writes or how many files it creates in `/tmp`. However, such control is needed to guarantee availability of a resource and, consequently, to counter denial-of-service attacks. Otherwise, writing excessive amounts of data to a temporary directory fills up the disk and denies other applications the creation of files in that file system. On Unix systems, creating an excessive amount of files results in the exceeding of kernel limits and effects also other user processes.

This problem has been addressed by a number of contributions (cf. §4 *Related Work*) that result in resource-aware Java environments. However, little is mentioned on the syntax used for resource management policies. Yet other work ([4][5][6], cf. §4 *Related Work*) has been experimenting with new permission languages that allow advanced policies such as time-based, context-based, user-based or hierarchical permissions. However, these languages are no longer based on Java's permission syntax nor are they enforced by the Java access controller.

Our work sets out to prove that resource management of higher-level resources is possible using the syntax of Java permissions and the standard Java access controller. The advantage of this solution is that resource control is integrated in the standard Java security architecture and implemented within the existing sandbox.

In this work, we do not deal with resource control of low-level resources such as CPU (central processing unit) or memory. Low-level resources are not mediated by the Java virtual machine, i.e. there is no way to ask the Java virtual machine for the CPU or a piece of memory, instead thread priorities must be manipulated or objects allocated. Thus, these resources cannot well be handled by the high-level access controller, which itself is implemented in Java. Access control for CPU and memory is preferably addressed within the core Java classes through modification of the Java thread scheduler, the heap manager and the garbage collector. Such work has been done for real-time Java and is described in e.g. [7][8][9].

The rest of the paper is organised as follows. Section 2 recalls existing Java permissions, their syntax and the existing enforcement mechanism. Section 3 describes our implementation of resource control using Java permissions and the Java access controller. It also describes our performance test. Section 4 supplies detailed references to related work. Section 5 concludes this paper and gives an outlook to future work.

2 Existing Permission Syntax and Enforcement

Java's current access control model for high-level resources, i.e. resources that are mediated by the Java virtual machine, is built of a policy and an enforcement mechanism.

The policy normally resides in files. There is a *system policy file*, usually within the installation directory of the Java virtual machine, which sets a system-wide policy. A *user policy file*, usually residing in the home directory of the user, can set a stricter policy for this user. These policy files can be overridden by a command line option to the Java virtual machine. Also, the policy can be set from within the application (if the effective policy allows the application to do so).

permission is constructed and submitted to the access controller (line 5). This access control is only performed if a security manager is installed (lines 3-4). By default, Java applications run *without* a security manager. The access controller checks this permission against the effective policy (line 11). If the permission is implied by the current policy for every piece of code within the call chain (i.e. on the call *stack*), the access control successfully returns and access to the resource is permitted (line 7). If the permission is not granted, an access control exception is thrown.

```
from SUN's J2SDK 1.3.1: java.security.Policy.Java (our comments)
1  public static void setPolicy(Policy policy) {
2      // Do access control only if a security manager is installed
3      SecurityManager sm = System.getSecurityManager();
4      if (sm != null)
5          sm.checkPermission(new SecurityPermission("setPolicy"));
6      // The access check was successful, access resource now
7      Policy.policy = policy;
8  }

from SUN's J2SDK 1.3.1: java.lang.SecurityManager (our comments)
9  public void checkPermission(Permission perm) {
10     // Delegate access control to AccessController
11     java.security.AccessController.checkPermission(perm);
12 }
```

Fig. 3: The setting of a policy is subject to typical Java access control, example from SUN's J2SDK source code

Fig. 3 illustrates also our research problem for this paper. The access check is performed before the resource is accessed. Once the access check has successfully completed, the resource is accessed without further restrictions.

For complete descriptions of the Java security architecture refer to [10][1][2][11].

3 Resource Control with the Java Sandbox

Within this project, we identify three typical cases of resource control (cf. Table 1) that can be used for any resource. The cases can be roughly grouped into *irrevocable* and *revocable* resource control. By definition, the two groups are mutually exclusive. However, in our solution, parts of a resource can be granted irrevocably, while other parts of a resource can be granted in a revocable fashion. For instance, an application may be allowed to open a file for an arbitrary length of time (irrevocable) but is only allowed to write a maximum of 500k to it (revocable).

Irrevocable resource control (described in the first column of Table 1) is the existing, standard way of making use of Java permissions. The permission is constructed for the needed resource and the action that is to be performed on it. No accounting data is needed for the permission check; and access is granted without further limits.

The two revocable types are implemented by us and make use of accounting data.

In the first of these two revocable cases—shown in the second column of Table 1—a permission is checked at regular intervals through a timer thread. *This is needed in cases where a piece of code holds a lock on a resource* and the lock shall be revoked when a certain condition is met. For example, an open file shall be released after 2 seconds, or a database connection is to be terminated after 60 minutes.

The second case of using accounting data—shown in the third column of Table 1—is more advanced. Not only is accounting data needed for constructing the permission for the first resource access, it is also needed for any subsequent access and accounting data is updated depending on the outcome of the actual access. The typical example is the writing of a file where one wants to limit the number of bytes that can be written to disk —either in total or for one specific file. Accounting data must be updated depending on if the disk write succeeded or failed. *This second type of revocable resource control is needed for fine-grained, continuous monitoring of access to a resource.*

Table 1: Different types of access control to resources

Irrevocable access control	Revocable access control	
Construct permission, check	<i>Expiration</i> of the access to a resource	Construct permission with accounting data, check, perform checked action, <i>update accounting data based on outcome</i>
Examples		
Typical Java permissions as described in Fig. 3	Limiting the time a file can be open	Limiting the number of bytes written to a file
Accounting data		
–	Timestamp when the file was opened	Bytes written per file

In order to achieve resource control as shown in columns two and three of Table 1, the following issues need to be worked with.

Table 2. Steps needed for making the Java virtual machine resource-aware.

1. A syntax for expressing a resource policy as Java permissions.
2. An enforcement mechanism for the permissions, which in our case is given with the standard Java access controller.
3. A decision of which code base is accountable for the resource use.

4. A data structure for keeping accounting data.
5. An identification of the places in the code where accounting data needs to be updated.
6. An identification of the places in the code where the enforcement mechanism must be invoked.

These issues are dealt with in the following sections. Section 3.1 describes the syntax we use for Java permissions for resource control. Section 3.2 deals with issues of permission enforcement and the timely update of accounting data. Section 3.3 provides performance data for our solution.

3.1 Permissions for Resource Control

In this section, we describe how we made use of the syntax for standard Java permissions to integrate resource control (cf. Table 2, item 1).

We integrate resource limits in the target and action strings of permissions. Following our approach described in Table 1, we use one syntax for *expiration* of access for the whole resource and another for fine-grained, *continuous* access control. The target string can be extended with a target-specific resource limit

```
<target>[:<global_target_limit>]
```

that restricts the access to the resource as a whole. The action string—if present—can receive an action-specific resource limit:

```
<action>[:<action_limit>]
```

For convenient handling, we build an abstract permission class¹ called `ResPermission`. This abstract class can then be subclassed for specific permissions. The `ResPermission` class extends `java.security.Permission` and supports e.g. the parsing of target and action string for resource limits. The implemented method `implies()` returns true when the resource limit of the implied permission is lower than the resource limit of the premiss. Targets and actions must be a full match. E.g.:

```
new ResPermission("/dev/term/b:1000").implies(
    new ResPermission("/dev/term/b:20")) returns true
```

```
new ResPermission("/dev/*:1000").implies(new
    new ResPermission("/dev/term/b:20")) returns false
```

because wildcard-matching is not implemented. This must be done in a suitable subclass.

Below are some instantiations of subclasses to `ResPermission`. They further illustrate the use of resource limits within target and action string.

Example: The serial port on `/dev/term/a` shall only be accessible for 50 seconds.

```
almhe.sec.ResCommPortPermission("/dev/term/a:50000")
```

¹ An abstract class is a class that cannot be instantiated, only subclassed. It consists of method signatures but can also contain fully implemented methods.

Example: The file /tmp/a shall only be open for reading or writing for 20 seconds.
`almhe.sec.ResFilePermission("/tmp/a:20000", "read, write")`

Example: The file /tmp/b can be open for an unlimited amount of time. A maximum of 1K may be written to the file.
`almhe.sec.ResFilePermission("/tmp/b", "write:1024");`

Example: The code executing under the following permission is only allowed to create a total of 500k in all the files it creates.
`almhe.sec.ResFilePermission("<<TOTAL>>", "write:512000");`

3.1.1 Extended Resource Control

So far we have only been dealing with one dimension of resource access, i.e. at the moment only one limit is allowed per target and action. However, further dimensions can be thought of e.g. not only the *length of time* a resource is accessed but also the *number of times* a resource can be accessed, created, modified or deleted. This could be solved using the same syntax but iterating the resource limit, preferably under the presence of an explanatory tag, e.g.

```
<target>[:<<tag_name>><<global_target_limit>>]*.
```

Following such an extended syntax, the following permission
`ResCommPortPermission("/dev/term/a:<length_ms>50000:<num_times>10")`
would allow the access to /dev/term/a for a maximum of 5000ms but no more than 10 times during the life of the Java virtual machine. This syntax could also be used for the action string limit. This extended syntax has not been implemented yet, it simply shows the way our proposed syntax could develop to comprise even finer-grained resource control.

3.2 Enforcement of Resource Control

The enforcement mechanism for our approach was given with the Java access controller. However, the access controller must be invoked at the correct position in the code with the correct permission that is to be checked.

To keep a clean design, we subclass or—where subclassing is not possible—wrap the concerned classes, i.e., `java.io.FileOutputStream`, `java.io.FileInputStream`, and `javax.comm.CommPortIdentifier`, and make all resource-specific changes in these subclasses or wrappers. We are aware that such a solution is not secure, because programmers can simply avoid using these classes. However, we set out to prove that resource control is possible using the standard Java security architecture. How to securely deploy the solution is left to future work.

Resource-accounting and policy-checking code has to be added when the resource is initially seized, e.g. in the constructor of the streams and the `open()`-method of the serial port, and when it is released and modified (e.g. bytes are written to or read from a file) (cf. Table 2, items 5 and 6). Unlike the policy-checking code, resource accounting must also consider the *failure* of a resource access. If e.g. no bytes were written due to a disk failure, this must also be reflected in the accounting data.

The required additions in the code are rather small and one path of future development is to find general ways of inserting resource control hooks without the need to subclass, e.g. through an event listener interface.

In order to enable our new permissions for the file system, we create a new security manager by subclassing the existing security manager and overwriting `SecurityManager.checkWrite()` and `SecurityManager.checkRead()`. The new security manager is extended by a new class field that is used for keeping accounting data.

A problem prior to the design of the class for accounting data is to decide which code—according to the code base entry of the policy file—is made responsible for resource use (cf. Table 2, item 3). It could be (1) the top caller or (2) the lowest caller that does not belong to a core Java class or (3) all callers. The Java security model would demand to make all callers accountable. But in that case, when it is time to check a permission, this permission does not remain static during the permission check but needs to be updated with accounting data for each protection domain. This is not possible with the standard access controller but would require a re-definition of the `checkPermission()`-method. In addition, the dynamic behaviour makes it difficult for an end user to understand why a resource permission fails because there is no way to provide an error message to the Access Controller—it only returns “Access denied” but no reason. Because of this, we make the *top protection domain* accountable (i.e. choice (1) above, cf. also Table 2, item 3) which also implements a semantics of trust between protection domains [12]. However, the top protection domain is not necessarily the protection domain that is on top of the call stack. When the resource access is done by privileged² code, or by code called by privileged code, then the privileged code is considered the top domain.

We store accounting data in fast and thread-safe hash tables. There is one hash table per accounted resource. The contents of the hash tables is specific per resource. However, the accountable protection domain must always be part of the accounting data. An example hash table that shows accounting data when limiting the size of files is shown in Table 3.

Table 3. Hash table with accounting data for writing files

Key	Value
Protection domain of the accountable code source + file name	Bytes written to the file
Example data	
Protection domain identifier 1: /tmp/a	2000
Protection domain identifier 2: /tmp/b	10
Protection domain identifier 2: /tmp/c	50000

We comprise all accounting data in one class, the `ResourceManager`. Thus, one can use the resource manager when information is needed about current resource use. By

² Just as the Unix operating system supports privileged code with the `setuid` feature, Java allows the marking of certain code within a class as privileged. This code ignores the permissions from its callers and only considers the permissions stated for this specific code and its called subroutines.

outputting the resource manager object, a snapshot overview over all resource usage is supplied.

3.3 Performance

In order to have a hint as to the performance penalty introduced by our resource management, we conducted a simple but deliberately stressful performance test by writing a 1MB file in chunks of 1, 10, 100, 500, and 1000 bytes. We tested the performance by letting the application perform under the Solaris utility *truss* that can be used to trace execution time. The test was run on a Sun Sparc work station running Solaris 8. The application was run with two versions of our resource-aware security manager, with the standard Java security manager and without a security manager.

Due to the fact that our resource management excessively invokes the access controller, namely three times per write due to three protection domains on the call stack, the performance penalty was high, especially in the time spent in user code. With a first version of the resource manager, there was even an invocation rate of five times per resource access. This was eliminated with a small change in the access controller that returns the top protection domain without invoking the access controller (but unfortunately makes the solution less portable). See Fig. 4 for details.

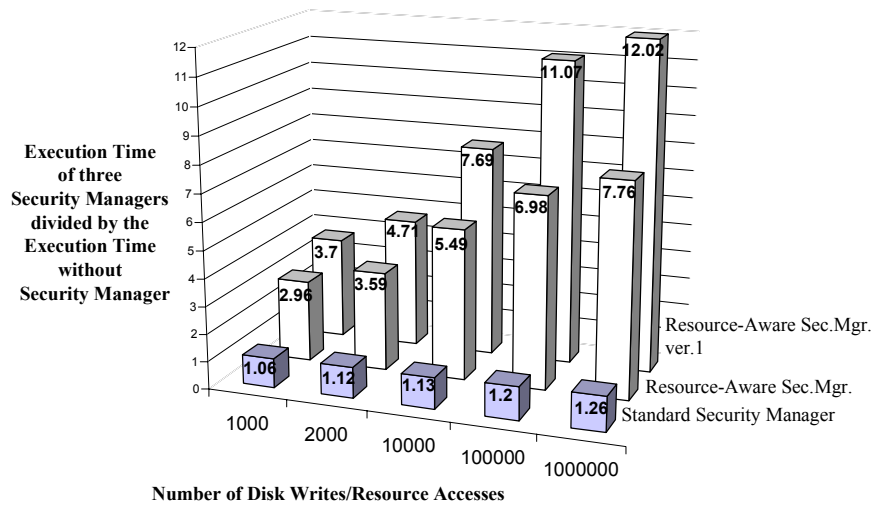


Fig. 4. Performance results when accessing the disk resource 1000, 2000, 10,000, 100,000 and 1,000,000 times. The more resource accesses, the slower perform the resource-aware security managers. In the worst case of 1 million resource accesses, the resource-aware security manager slowed the program execution down with a factor of 12. It is interesting to note that even the standard security manager is slightly affected by the number of resource accesses, although the access controller is invoked the same number of times (namely 41) in all five runs.

A conclusion drawn from the performance test is that the access controller must perform very fast if resource management with the help of the access controller shall be feasible. To unburden the access controller, additional intelligent assumptions about the outcome of an access control check could be implemented (e.g. if a small disk write is attempted within a certain time window after a successful disk write, the probability that the new write will succeed is high). However, such reasoning leads to an undeterministic behaviour of the access controller and could possibly be exploited by attacks.

We conclude this section by a critical view of our approach and give an outlook to one area of application. Our solution adds complexity to the already complex Java security architecture. Not even the standard security manager is much used for applications or servlets because often there is a trust relationship—e.g. through contracts—between code producer and platform owner. But also, because the policy management is considered cumbersome and error-prone. When there is a trust relationship, the “trusted” code is often given full permissions without much thought. Our solution may suffer from the same problem.

However, new platforms are emerging that run untrusted code, possibly without a trust relationship. In such cases, resource control is an important issue for guaranteeing the availability of the platform. An example for such new architectures are residential gateways or service gateways [13] that run services from multiple vendors as threads within one Java virtual machine with dynamic policies. Security problems in such a platform and their possible solutions have been dealt with in greater detail in [14].

4 Related Work

As mentioned in the introduction, much work has been done to bring resource-awareness to Java. The focus of previous work is rather on the goal of arriving at a resource-aware Java environment than on the integration of the solution in the existing architecture, which was our prime goal. Solutions are often used for real-time applications, i.e. for quality-of-service and not so much for availability as part of security. Also, research has been conducted in the area of formulating advanced policies for Java. We start with references to resource-aware Java virtual machine implementations.

JRes [15], a resource-accounting interface for Java, adds resource accounting and limiting facilities to Java as a class library, using bytecode rewriting. JRes introduces a resource manager class that co-exists with the Java class loader and the security manager. The resource manager contains native code for CPU and memory accounting. Overuse callbacks are generated and throw exceptions when resource limits are exceeded. The policy is hard-coded as Java code and resource limits apply to any code base.

J-Kernel [16] addresses the problem of untrusted servlets or Java plug-ins executing in one Java virtual machine, e.g. a web server. J-Kernel implements capabilities—access tokens—that represent handles to resources, i.e. a resource can only be accessed when the requester holds a corresponding capability. Unlike Java

object references, capabilities can be revoked at any time. J-Kernel is implemented as a Java library and sets out to prove that Java language features can be used to secure communicating pieces of Java code in a Java server. The access control policy is hard coded.

Chander, Mitchell and Shin [1] introduce bytecode instrumentation to remedy Java's lack of resource control. New, safe classes and methods will automatically—through the bytecode modifier—replace the original Java code prior to execution. The approach is as follows.

- * Identify the Java classes that control the resources that need further monitoring.
- * Subclass this class and add additional security checks and resource control to the subclass.
- * Invoke the bytecode filter on the original class so that all references to the original class are replaced with references to the new, safer class.

Handling final classes is more complex than described above—it needs method-level bytecode instrumentation—but follows the same approach.

No examples of policy statements or a description of their syntax are provided. The specification language is said to be proprietary. It allows listing all classes and methods that need to be replaced. In addition, a graphical user interface is provided to edit resource limits such as number of windows, network connections or threads.

Much work deals with CPU and/or memory management. Real-time Java (www.rtj.org) allows resource control of the CPU through installable thread schedulers. Bernadat, Lambright and Travostino [8] have implemented a Java virtual machine that controls CPU and memory according to a policy in a policy file (with undocumented syntax). Another case of undocumented policy syntax comes from the Aroma Java virtual machine [9], which allows resource control for CPU, disk and network. SOMA [7] allows control over the thread scheduler and garbage collector and makes use of the Java VM Profiler Interface (java.sun.com/j2se/1.3/docs/guide/jvmpi/). Spout [17], originally designed for applet security, allows resource control for CPU, memory, threads and windows.

The specification of policies for Java is a related field to our work. Even before Java 2's security model with permissions in a policy files was introduced, researchers have been looking at ways of expressing advanced access control policies for the Java security manager. Nimisha, Mehta and Sollins [4] propose a constraint language that allows for three different kinds of permissions.

- * Subject-based permissions base the access control on caller credentials—e.g. an applet's signature or code base.
- * Object-based permissions consider the resource that shall be controlled—e.g. the total number of created windows.
- * History permissions express e.g. that an applet is not allowed to connect to the network after it has read a private file.

This system was implemented in Java 1.0 by extending the security manager.

Java Secure Execution Framework (JSEF) [5] introduces a syntax for negative permissions—permissions that disallow access. JSEF also comprises support for hierarchical policy files that allow more than the Java 2 system policy file and user policy file. In addition, JSEF allows the negotiation of permissions at run-time. A piece of Java code could e.g. ask the user or a policy server at run time if a specific access should be completed.

Corradi, Montanari, Lupu et al. [6] propose a new language (Ponder) for Java permissions. Ponder supports negative permissions, support for subjects and objects, time restrictions and conditional expressions. Also the instant revocation of granted privileges is discussed.

5 Conclusion and Future Work

In this paper, we have shown that resource control of high-level resources is possible by using extensions of Java permissions and an unmodified Java access controller. Thus, resource control can be done within the existing Java sandbox. This contributes to a clear design of the security architecture. The Java permission syntax is so versatile that it can be extended to also support resource limits. However, the major work lies in finding all places in the code where the resource limit should be enforced or accounting data updated.

Still, the solution we present here is far from being deployed. In this paper, we only wanted to give a proof of concept. In order to work with it, ways must be found that force programmers to adhere to resource control and to use resource-aware classes. Ways of achieving this are modified Java virtual machines or automatic class modification at runtime (so-called bytecode modification).

The performance penalty in code that excessively accesses resources is mostly due to the fact that each resource access implies a number of invocations of the access controller—the number depends on the number of protection domains on the call stack. If the performance is to be improved, the number of calls to `AccessController.checkPermission()` needs to be reduced, for instance by intelligently avoiding to check permissions that have only a slightly changed resource limit.

Other performance bottlenecks can be synchronisation issues on the accounting data object. If different execution threads need access to a common accounting object, only one thread gets the access at one time; others have to wait. Also the used hash tables may be bottlenecks. Other structures, for instance containers that are similar to database tables, should be explored and tested for their performance.

So far we have only been dealing with the length of time a resource can be accessed. Another dimension is the number of times a resource can be accessed, created, modified or deleted. Further development includes the expansion of this technique for other high-level resources such as windows, microphone, speaker, etc. More thorough performance tests using recognised benchmarks should be done.

However, one of the more important future tracks is an investigation of what new vulnerabilities are introduced by such high-level resource management.

References

- [1] Marco Pistoia, Duane F. Reller, Deepak Gupta et al. *Java™ 2 Network Security*. 2nd Edition. Prentice-Hall. 1999.
- [2] Gary McGraw, Edward Felten. *Securing Java—Getting Down to Business with Mobile Code*. Wiley & Sons. 1999.

- [3] Ajay Chander, John C. Mitchell, Insik Shin. *Mobile Code Security by Java Bytecode Instrumentation*. Proceedings of the 2001 DARPA Information Survivability Conference & Exposition II, DISCEX. Vol.2. pp. 27-40. IEEE. 2001.
- [4] Nimisha V. Mehta, Karen R. Sollins. *Expanding and Extending the Security Features of Java*. Proceedings of the 7th USENIX Security Symposium. January, 1998.
- [5] Manfred Hauswirth, Clemens Kerer, Roman Kurmanowytch. *A Secure Execution Framework for Java*. Proceedings of the 7th conference on computer and communications security. pp. 43-52. ACM. November, 2000.
- [6] Antonio Corradi, Rebecca Montanari, Emil Lupu, Morris Sloman, Cesare Stefanelli. *A Flexible Access Control Service for Java Mobile Code*. Proceedings of the 16th Annual Conference on Computer Security Applications (ACSAC). pp. 356-365. IEEE. 2000.
- [7] Paolo Bellavista, Antonio Corradi, Cesare Stefanelli. *How to monitor and control resource usage in mobile agent systems*. Proceedings of the 3rd International Symposium on Distributed Objects and Applications, DOA'01. pp. 65-75. IEEE. 2001.
- [8] Philippe Bernadat, Dan Lambright, Franco Travostino. *Towards a Resource-safe Java for Service Guarantees in Uncooperative Environments*. In Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications. Madrid. IEEE. Dec. 1998.
- [9] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, et al. *State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine*. Proceedings of the Java™ Virtual Machine Research and Technology Symposium. Usenix. April, 2001.
- [10] Li Gong, *Inside Java™ 2 Platform Security*. The Java Series. Addison-Wesley. 1999.
- [11] Scott Oaks. *Java Security*. 2nd Edition. O'Reilly. 2001.
- [12] Dirk Balfanz, Drew Dean, Mike Spreitzer. *A Security Infrastructure for Distributed Java Applications*. Proceedings of the IEEE Symposium on Security and Privacy (S&P), 2000. Pages: 15-26. May, 2000.
- [13] Kirk Chen, Li Gong. *Programming Open Service Gateways with Java Embedded Server™ Technology*. The Java Series. Addison-Wesley. 2001.
- [14] Almut Herzog. *Secure Execution Environment for Java Electronic Services* (tentative). Licentiate thesis. Dept. of Computer and Information Science. Linköping university. To be presented, fall 2002.
- [15] Grzegorz Czajkowski, Thomas von Eicken. *JRes: A Resource Accounting Interface for Java*. Proceedings of OOPSLA'98. Pages 21-35. ACM. 1998.
- [16] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, Thorsten von Eicken. *Implementing Multiple Protection Domains in Java*. Proceedings of the USENIX 1998 Annual Technical Conference. pp. 259-272. Usenix. 1998.
- [17] Tzi-cker Chiueh, Harish Sankaran, Anindya Neogi. *Spout: A Transparent Distributed Execution Engine for Java Applets*. Proceeding of the International Conference on Distributed Computing Systems (ICDCS). pp. 394-401. IEEE. April, 2000.