

Security Policy Management in Federated Computing Environments

Daniela Inclezan and Michael Sobolewski
Texas Tech University

Abstract- The default Java implementation for security policies based on policy files doesn't comply with the specific needs of metacomputing environments. Managing a large number of policy files for all Java runtime systems in the metacomputing system doesn't scale. This paper presents a federated approach for security policy management in Java-based metacomputing systems. Security policies are stored in a policy base, which is managed by its policy service provider (Policer). The policy base and its Policer are replicated and the replicated policy bases are synchronized with each other in order to avoid a single point of failure. Any bootstrapping service provider gets its security policy dynamically from any available Policer in the network. The proposed solution ensures uniform policy-based authorization for all the services in the SORCER metacomputing environment through the use of the dynamic policy management methodology.

I. INTRODUCTION

Built on the object-oriented paradigm is the Service Object Oriented (SOO) paradigm, in which the objects are distributed, or more precisely they are remote (network) objects and play some predefined roles. A service provider is an object that accepts remote messages, called *exertions*, from service requestors to execute an elementary item of work (network instruction) – a *service task*, or a composite item of work (network procedure) – a *service job*.

The exertion becomes an SOO program that is dynamically bound to all relevant and currently available service providers on the network. This collection of providers dynamically participating in this federated remote invocation is called an *exertion federation*. This federation is also called a *virtual metacomputer* as federating services executing hierarchically nested exertions are located on multiple physical compute nodes held together by an SOO infrastructure so that, to the individual requestor submitting the exertion, it looks and acts like a single computer.

The SORCER environment [11,12,13,14] provides the means to create interactive SOO programs and execute them without writing a line of source code [13]. Exertions can be created using interactive user interfaces downloaded directly from service providers. Using these interfaces the user can execute and monitor the execution of exertions in the SOO metacomputer. The exertions can be persisted for later reuse. This feature allows the user quickly to create new applications or programs on the fly in terms of existing tasks and jobs.

In this paper a security policy management system is described to allow the exertion federation for a secure

collaboration with presented policy services that enforce security permissions on all the federating providers.

The paper is organized as follows. Section 2 provides a background review of service oriented architectures with a related discussion on authorization in SOO federated environments; Section 3 describes the presented policy management methodology; Section 4 presents the design issues of required solution in SORCER; Section 5 provides concluding remarks.

II. BACKGROUND REVIEW

A. Service Object Oriented Computing

Various definitions of a Service-Oriented Architecture (SOA) leave a lot of room for interpretation. In general terms, SOA is a software architecture using loosely coupled software services that integrates them into a distributed computing system by means of service-oriented programming. Service providers in the SOA environment are made available as independent service components that can be accessed without a priori knowledge of their underlying platform or implementation. While the client-server architecture separates a client from a server, SOA introduces a third component, a service registry

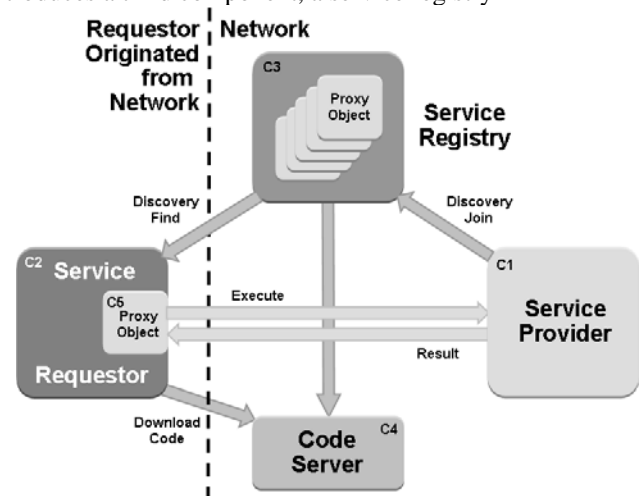


Figure 1. Service-Oriented Architectures

We can distinguish the *service object-oriented architectures* (SOOA), where providers, requestors, and proxies are network objects, from *service protocol oriented architectures* (SPOA), where a communication protocol is fixed and known beforehand by the provider and requestor. Based on that protocol and a service description obtained

from the service registry, the requestor can bind to the service provider by creating a proxy used for remote communication over the fixed protocol. In SPOA a service is usually identified by a name. If a service provider registers its service description by name, the requestors have to know the name of the service beforehand.

In SOOA, a proxy—an object implementing the same service interfaces as its service provider—is registered with the registries and it is always ready for use by requestors. Thus, in SOOA, the service provider publishes the proxy as the active surrogate object with a codebase annotation, e.g., URLs to the code defining proxy behavior (RMI and Jini ERI), as illustrated in Figure 1. In SPOA, by contrast, a passive service description is registered (e.g., an XML document in WSDL for Web/Globus services, or an interface description in IDL for CORBA); the requestor then has to generate the proxy (a stub forwarding calls to a provider) based on a service description and the fixed communication protocol (e.g., SOAP in Web/Globus services, IIOP in Corba). This is referred to as a bind operation. The binding operation is not needed in SOOA since the requestor holds the active surrogate object obtained from the registry.

Web services and Globus services cannot change the communication protocol between requestors and providers while the SOOA approach is protocol neutral. In SOOA, how an object proxy communicates with a provider is established by the contract between the provider and its published proxy and defined by the provider implementation. The proxy's requestor does not need to know who implements the interface or how it is implemented. So-called smart proxies (Jini ERI) grant access to local and remote resources; they can also communicate with multiple providers on the network regardless of who originally registered the proxy. Thus, separate providers on the network can implement different parts of the smart proxy interface. Communication protocols may also vary, and a single smart proxy can also talk over multiple protocols including application specific protocols.

Crucial to the success of SOOA is interface standardization. Services are identified by interfaces (service types); the exact identity of the service provider is not crucial to the architecture. As long as services adhere to a given set of rules (common interfaces), they can collaborate to execute published operations, provided the requestor is authorized to do so.

Let's emphasize the major distinction between SOOA and SPOA: in SOOA, a proxy is created and always owned by the service provider, but in SPOA, the requestor creates and owns a proxy which has to meet the requirements of the protocol that the provider and requestor agreed upon a priori. Thus, in SPOA the protocol is always a generic one, reduced to a common denominator—one size fits all—that leads to inefficient network communication in some cases.

In SPOA, each provider can decide on the most efficient protocol(s) needed for a particular distributed application.

Service providers in SOOA can be considered as independent network objects finding each other via a service registry and communicating through message passing. A collection of these object sending and receiving messages—the only way these objects communicate with one another—looks very much like a service object-oriented distributed system.

Do you remember the eight fallacies of network computing? [18] We cannot just take an object-oriented program developed without distribution in mind and make it a distributed system, ignoring the unpredictable network behavior. The challenge related to authorization enforcement in metacomputing environments is based exactly on this dynamic nature of the environment. Security permissions cannot be specified and then granted to service providers in relation to code location since code location is generally not known to requestors *a priori* and can change dynamically (see Figure 1). Therefore static permissions based on code location must be replaced by dynamic permissions.

B. Default Authorization With Policy Files

The default implementation for security policy management in a Java runtime system relies on policy configuration files. These files have a simple hierarchical syntax composed of grant statements, each of which can be associated with a code base, a set of principals (optionally) and a set of permissions. A *grant* statement as a whole specifies the security permissions allowed to code downloaded from the code base—a static location—on the local Java runtime system. When a set of principals is included, the permissions are granted only to the entities corresponding to those principals. Since policy files have such a simple syntax and are saved in plaintext, they can be created manually using a text editor. Another option is to use the graphical utility called *Policy Tool* [16]. By default there is only one system policy file and one (optional) user policy file [15]. In order to enforce checking of the permissions stored in policy files, the security manager must be enabled at runtime. It ensures that a static Policy object is instantiated and populated based on the information coming from the system and user's policy files.

Such an approach is sufficient in the vast majority of cases, but being a default implementation, it may not be adequate for special types of applications. Policy files have a well-known syntax, are saved in plaintext and the location where they are stored is commonly known [7]. This means that even unauthenticated and unauthorized personnel can make harmful changes to their content when the policy file *write* access is not set correctly. Thus, policy files can create a breach in the security of a system and do not represent an adequately secure solution for applications that require high level security.

As well, policy objects created from policy files are static objects and changes made to the policy files are not reflected by the Java runtime system unless it is restarted. In this case, the default implementation with policy files shows a lack of flexibility that might be essential for some distributed applications.

Furthermore, as permissions are defined in policy files based on static code source, this approach is not compatible with code mobility and the dynamic nature of federated computing environments.

C. Authorization in Jini

Jini services [8], which employ the SOO paradigm, also use policy files to handle security permissions. In this case though, the policy object is dynamically created when the service is discovered [8]. Since permissions cannot be granted based on code location, they are constructed based on what is called *protection domains*. A protection domain is defined by principals forming a subject (who is executing the code), code source and class loader (the object responsible for loading classes). When granting permissions in Jini the code source is actually not so important. Permissions are granted to a combination of context class loader for the current thread (Figure 2) and subject on whose behalf the thread is running.

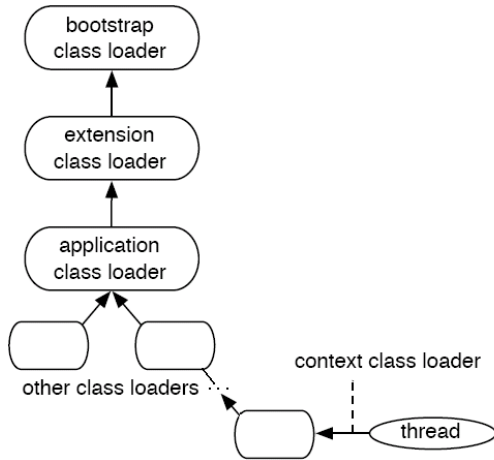


Figure 2. Thread context class loader

The policy object created at bootstrapping for service providers in Jini is an instance of the `AggregatePolicyProvider` class, which supports the association of sub-policies with context class loaders (Figure 3). The sub-policy associated with the current context class loader or any of its parents is the active policy. If no such policy is found, then the fallback sub-policy (`initialGlobalPolicy`) becomes the active one.

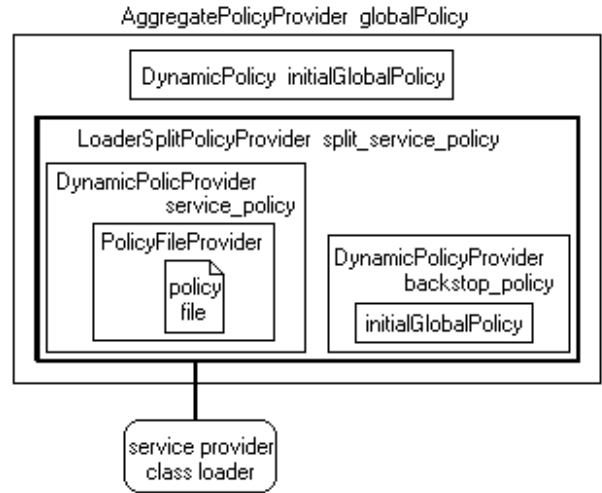


Figure 3. Service provider policy in Jini

The class loader used to load the service provider’s classes is associated with the `split_service_policy`, which is an instance of the `LoaderSplitPolicyProvider`. The `split_service_policy` helps dividing the permission queries in queries involving the service provider’s specific classes, and other permission queries. Based on this division, a different policy will be used for each of the two cases. Whenever there is a permission query that involves code belonging to the service provider’s loaded classes, the `service_policy`, which was created based on information coming from a service provider specific policy file, is used. For permission queries for code coming from other classes, the `initialGlobalPolicy` is used.

Therefore, the Jini approach is to wrap policies into an aggregated policy in order to enforce different permissions depending on the class loader that is used. The code source doesn’t need to be known when permissions are specified. On the other hand, policy files are used for service provider related permission storage.

Relying on policy files to enforce authorization on Jini services can cause a scalability problem. For example, if the same service is deployed on hundreds of hosts and later on some modification to the policy file is required, this change will need to be manually replicated on all related hosts. This can be not only time consuming and error-prone but as well difficult to perform in a quick and correct manner.

In federated computing environments in general, the scalability and security problems raised by enforcing authorization with policy files still exist. On top of that, there is another issue regarding the rigid syntax of policy files [7]. A security policy management based on roles may be needed for metacomputing systems, but it cannot currently be represented in a policy file. A more flexible syntax is required for such environments.

III. SECURITY POLICY MANAGEMENT METHODOLOGY

In the federated computing environments that use policy files to handle authorization, a large number of policy files

reside on hosts. For each service provider that has registered with the service registry there is one policy file (see Figure 4). This means that on the whole federated environment there can be hundreds of policy files to manage. Modifying all these files and especially keeping the policy files synchronized is definitely cumbersome and, not secure enough either.

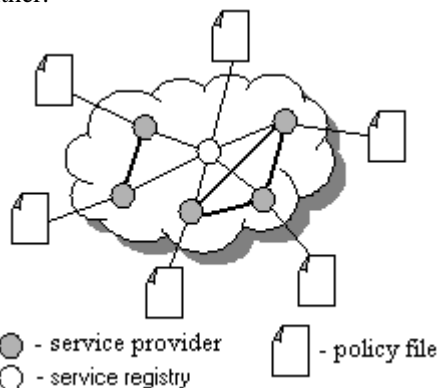


Figure 4. Default approach for authorization

The solution to these security and scalability problems in the federated computing environment is to centrally manage the security policies. In order to do that we propose to store all the security permission information in a central base managed by a special service provider (Policer). Having all the permissions stored in the central base eases the management and updating operations, since they are performed in one place only, not on all the hosts. On the other hand, having all the security policy information stored in a single place can generate the one-point failure. This can be avoided by replicating the Policer and its policy base in the federated computing environment. The policy bases of all the active policers must be synchronized. Thus, when a policer bootstraps, it contacts any other active policer and synchronizes its policy base to that of the active policer.

Any policer (master or slave) is able to provide the security policy object to a requesting service provider. At service bootstrapping, every service provider contacts dynamically an active policer and asks for its policy object. After mutual authentication between the bootstrapping service provider and the contacted policer, the policer retrieves the security policy information for the bootstrapping service provider from the policy base. It then creates a policy object, populates it with the security information retrieved from the policy base and passes this policy object to the service provider. On its side, the bootstrapping service provider reinforces all permissions contained in the received policy object.

While most systems treat authorization in a static way, the approach presented here is dynamic: authorization information can be obtained by the service providers from any policer, running on any host. In fact, the location where the contacted policer is running is unknown to the bootstrapping service provider. The location isn't static either, it can change over time since a policer can go down and be replaced by some other policer started on a different

host. The policer is dynamically discovered by the service provider trying to obtain its authorization information through the use of the service registry. The service registry has the policer's proxy, which was obtained during the policer registration phase. This proxy is passed to the bootstrapping service provider, which needs to verify whether it is secure to use the policer proxy or not. Both the data and the code of the policer proxy are verified in this stage. Only after proxy verification the communication with the policer is enabled.

The security information stored in policer's bases can be modified by an authenticated and authorized administrator through the administrative user agent (see Figure 5). This is a graphical interface that allows the administrator to modify, insert and delete security policies in the policy base. If all policers would provide the administrative user agent, this could create problems in the case of concurrent modifications in multiple policers. The same security policy should be present in all policy bases at all times for the same service provider. But, for example, if at the same time two different authorized administrators make conflicting changes on the same security policy, then the policers would not know which one of the two modifications is the correct one, the one to persist in the policy base.

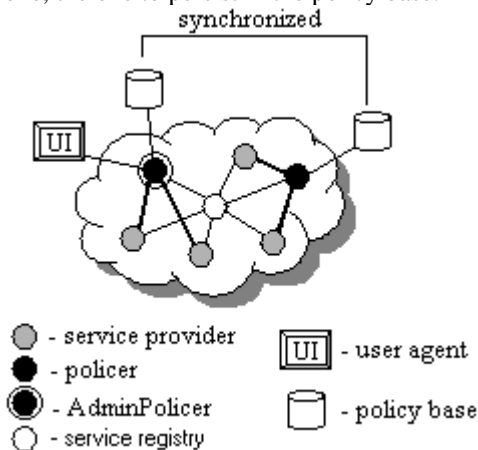


Figure 5. Proposed solution

This concurrency problem is solved currently by allowing only one of the active replicated policers in the environment to provide the user agent at all times. If the active policer enabling policy administration (AdminPolicer) suddenly becomes inactive, a new AdminPolicer is elected from the group of already active policers. This master-slave approach (where the master is the AdminPolicer and the slaves are all the other policers) simplifies the synchronization of the policy bases. Whenever the AdminPolicer receives a modification from an authorized administrator through the user agent, it notifies all other active policers about the current update. Through this mechanism all active slave policers continually maintain their policy bases synchronized with that of the AdminPolicer. This also implies that two policers must be active in the environment all the time: the master

policer and a slave policer that can take over the role of the master whenever the current master becomes inactive.

Any change made by an authorized administrator to the security policy of a service provider is immediately enforced on all active service providers of that policy type. The AdminPolicer receiving the change through its user agent is responsible for notifying all the corresponding active service providers of the policy modification and for passing the new policy object on to them. The service providers receiving the modified policy object dynamically reinforce the new security policy on their side. The major components of the security policy management methodology and the interactions between them are presented in Figure 6.

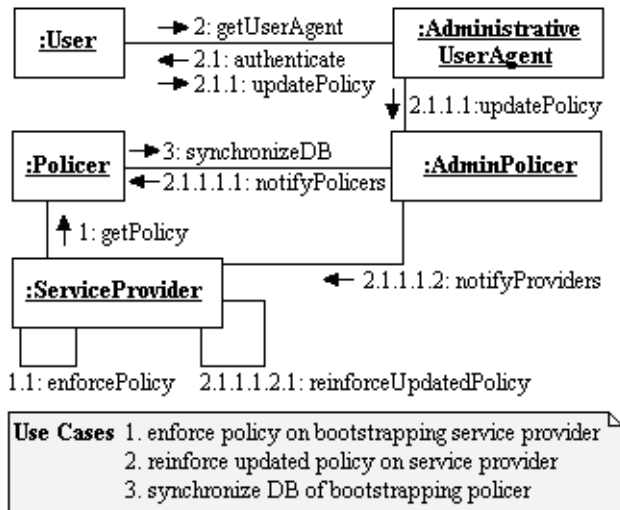


Figure 6. Major components and interaction diagram

In the case of disconnected operations a strict policy object containing the minimal set of permissions that would allow normal functioning should be available to any service provider. Therefore, when the bootstrapping service provider fails to contact any of the policers due to intermittent lack of connectivity, the provider gets the default minimal policy object and reinforces it on itself. The provider will try to get its proper policy from any available policer later.

The federated policy management system is protected against outside intrusions by its own security solution. Administrators are authenticated and authorized before being allowed to access the administrative user agent and make any modifications to the existing policy base. Confidentiality and integrity is enforced on all remote communication channels. The actual schema of the policy base is hidden to the administrators and all other users.

IV. DESIGN OF THE POLICY MANAGEMENT SYSTEM

A. Policy base

The policy base is represented by a relational database, which for now mimics the structure of a policy file (see Figure 7). Later on, the database schema can be easily

modified to reflect any additional needs of a federated computing environment, for example role management.

The main table of the schema is *Policy*. This contains all the information needed to identify a service provider's security policy and to distinguish between different security policies stored in the database: service ID, service provider name, main published interface, location, host where the service is running and user directory where the service was started. A combination of these attributes is passed by the bootstrapping service provider to an active policer and is used by the contacted policer to retrieve the right security policy from the database (step 1 in the interaction diagram in Figure 6).

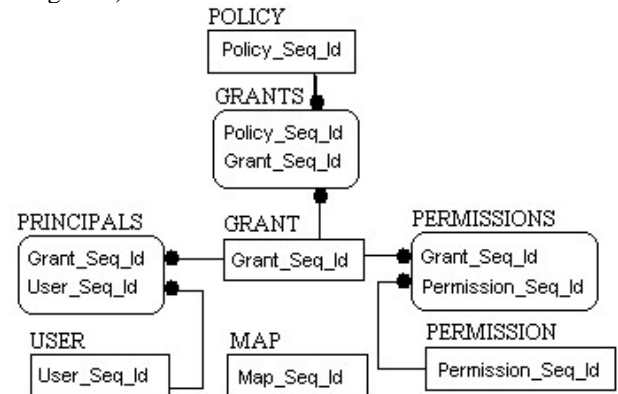


Figure 7. Policy database schema

The information that would otherwise appear in a grant statement in a policy file is stored in the *Grant* table. The *User* table stores information about the (optional) principals. The *Permission* table contains records similar to the permission statements of a policy file. All the strings that would appear repeatedly in different tables or records (such as class names, target names, etc.) are stored in the *Map* table. The association tables *Grants*, *Permissions* and *Principals* allow for a flexibility and reusability of records.

B. Administrative user agents

Only authenticated and authorized administrators are allowed to get the administrative user agent (steps 2, 2.1, and 2.1.1, Figure 6). There are different solutions for personnel authentication and authorization. The simplest one is based on login and password verification against security information stored on a policer. A more secure and advanced solution would require the use of smart cards, case in which keys are stored on the actual smart card only. A third solution worth considering would be the Kerberos protocol, which provides strong authentication by using secret-key cryptography [4, 6] and never passing the actual password through the network. In order to confer more flexibility to the environment, the best solution is to have all these approaches implemented and let the client select the suitable one for him.

The graphical interface of the Policy Tool utility [16] is used as a Service UI [17] model for the administrative user agent. But, in order to comply with the specific needs of the database schema focused on federated metacomputing,

some major changes have been made to the GUI inspired by Policy Tool. Fields identifying the service provider to which the edited policy belongs are added on the policy editing window: service ID, service provider name, implemented interface, etc. The first window to appear is a window listing all policies in the database. The changes to the policy information are persisted in the policy base by the AdminPolicer instead into a file.

C. *Policer replication and synchronization*

The RIO framework [9] is used for policer provisioning in order to ensure that at least two active policers exist in the environment at all times. In case the master policer fails, a protocol for the election of the new master is applied: the remaining active slave policers send a message to the other active policers requiring to be elected as the new master. The first slave policer sending this message becomes the new master.

The databases of all active policers must contain the latest security policy information at all times. The synchronization of the databases of active policers is managed by the master policer. It remembers all the active slave policers in a continuously updated structure and notifies all the policers in this structure of any policy modification coming from an administrator (step 2.1.1.1, Figure 6). The synchronization of bootstrapping policers (step 3, Figure 6) relies on the order of database events rather than on an unreliable real time clock. The bootstrapping policer compares its latest event number with that of an active policer's to know how much behind it is. Then, either the modified records are updated or, if too many changes have occurred in the meantime, the whole database is copied from the active policer to the bootstrapping policer.

D. *Policy reinforcement on service providers*

The policy object coming from a contacted policer can be either statically or dynamically reinforced on the bootstrapping service provider (steps 1.1 and 2.1.1.1.2.1, Figure 6). A dynamic enforcement would imply adding the new policy object under the umbrella of the existing AggregatePolicyProvider object. In this case, the new policy object passed by the policer would add a new layer of restrictions on top of those already existing.

V. CONCLUSIONS

Using policy files for authorization in Java-based metacomputing environments doesn't scale well. A scalable solution for security policy management in federated metacomputing environments is proposed here. It has the advantage of flexible database management of all security policies that say what system resources can be accessed, in what fashion, and under what circumstances. Thus, it provides for uniform authentication, authorization, and access control for all federating service providers.

Confidentiality and integrity of policy information is guaranteed by securing all network communication channels. Consistency of policy data is ensured by policy base synchronization mechanisms. A friendly user agent is provided for the administrators to create and update policy information persisted in the database and then synchronized with other policer databases. Replication and autonomic provisioning of policers prevents service unavailability from occurring.

The presented methodology is designed especially for federated computing environments characterized by code mobility. Permissions are not defined with respect to the code source, but in relation to the entity running the code and the class used to load the code. The location of the security policy service provider (policer) is not fixed and is not known before hand to the other service providers. The service providers discover an active policer dynamically, through the use of the service registry.

This scalable methodology can be similarly applied to other aspects of security in metacomputing environments, for example federated authentication. A *KeyStorer* service provider persisting keys in a database can be designed following the same approach.

REFERENCES

- [1] Diehl and Associates, Inc., "Mckoi SQL Database", 2005. Retrieved December 27, 2006, from <http://mckoi.com/database/>
- [2] L. Gong, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, Prentice Hall PTR, 2nd edition, 2003.
- [3] J. Grams, and D. Somerfield, *Professional Java Security*, Wrox Press, Birmingham, UK, 2001.
- [4] Ch. Kaufman, R. Perlman, and M. Speciner, *Network Security: Private Communication in a Public World*, Second Edition, Prentice Hall PTR, 2 edition, 2002
- [5] J.F. Koopmann, "Embedded Database Primer", 2005. Retrieved December 27, 2006, from: <http://www.dbazine.com/ofinterest/oi-articles/koopmann5>
- [6] Massachusetts Institute of Technology, "Kerberos: The Network Authentication Protocol", 2003. Retrieved December 27, 2006, from <http://web.mit.edu/Kerberos/>
- [7] T. Neward, "When "java.policy" Just Isn't Good Enough", 2001. Retrieved December 28, 2006 from: www.javageeks.com/Papers/JavaPolicy/JavaPolicy.pdf
- [8] J. Newmarch, *A Programmer's Guide to Jini Technology*, Apress, Berkley, CA, 2000.
- [9] Project Rio, A Dynamic Service Architecture for Distributed Applications. Retrieved December 27, 2006, from <https://rio.dev.java.net/>
- [10] Red Hat Middleware, "Hibernate: Relational Persistence for Java and .NET", 2006. Retrieved December 27, 2006, from <http://www.hibernate.org>
- [11] M. Sobolewski, *Federated P2P services in CE Environments, Advances in Concurrent Engineering*, A.A. Balkema Publishers, 2002, pp. 13-22.
- [12] M. Sobolewski, *FIPER: The Federated S2S Environment, JavaOne, Sun's 2002 Worldwide Java Developer Conference*, 2002. Retrieved

December 27, 2006,
from <http://sorcer.cs.ttu.edu/publications/papers/2420.pdf>

- [13] M. Sobolewski, R. Kolonay, Federated Grid Computing with Interactive Service-oriented Programming, International Journal of Concurrent Engineering: Research & Applications, Vol. 14, No 1., pp. 55-66, 2006
- [14] SORCER, Laboratory for Service-Oriented Computing Environment, Retrieved December 27, 2006, from <http://sorcer.cs.ttu.edu>.
- [15] Sun Microsystems, Inc., "Default Policy Implementation and Policy File Syntax", 2002. Retrieved December 27, 2006, from <http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html>
- [16] Sun Microsystems, Inc., "Policy Tool – Policy File Creation and Management Tool", 2001. Retrieved December 27, 2006, from <http://java.sun.com/j2se/1.3/docs/tooldocs/win32/policytool.html>
- [17] The ServiceUI Project, Retrieved March 15, 2007, from <http://www.artima.com/jini/serviceui/index.html>
- [18] Fallcies of Distributed Computing. Retrieved March 15, 2007, from http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing.