Security Solutions for Jini-Based Applications

Ghita Mostéfaoui
Department of Informatics, University of Fribourg, Switzerland

Abstract: *Since its first release, Jini became a promising technology to build fault tolerant distributed systems. The actual Jini architecture however lacks a strong security model. Based on a concrete example, this paper aims at reviewing the main security architectures that have been proposed by the research community and presents an evaluation of them. This work may serve as a basis for securing Jini-based systems by selecting the set of solutions provided by each model, depending on the security needs introduced by each specific application.*

Keywords: *Distributed systems, Jini, security architectures.*

Received January 28, 2003; accepted May 24,2003

1. Introduction

The Jini networking technology [5, 12, 30] developed by Sun Microsystems, is an innovative technology for building reliable, fault-tolerant distributed applications. It allows to easily form networks to share services without previous planning, installation or administration effort.

This work is part of a collaboration project between the SOFTENG (Software Engineering Group) at the University of Fribourg and the LIP6 (Laboratoire d'Informatique Paris 6) at the University Pierre et Marie Curie Paris VI. We are interested in designing and developing a software framework for context-based security in distributed systems. The resulting framework is intended to be a generic prototype used by distributed applications in order to integrate dynamic security solutions [15, 16]. Our framework is developed using the Java programming language and the Jini technology. However, due to the ad hoc nature of Jini, security is of main concern. Until now, only few efforts partly deal with the Jini security model.

This paper aims at reviewing some of the main security architectures for Jini and provides an evaluation of them. For clarity reasons, we base our actual study on a concrete and simple example: sending a protected document to a network printer. We begin by identifying the main threats in the actual Jini architecture. Then, we propose a set of requirements for a secure Jini-based system. Section 4 is dedicated to presenting the standard security concepts in the Java language. They are not directly related to Jini but some of these concepts may be useful for future integration with the Jini model. An example of a centralized model for securing Jini-based systems is presented in section 5. Section 6 presents an example of a decentralized model. Section 7 discusses a security framework based on the use of self-signed certificates for services and user authentication. An authentication and authorization architecture for Jini services achieving client transparency is discussed in section 8. Section 9 discusses the efforts made by Sun to add security to Jini. It includes a recent security model proposed by the Jini Project team. Section 10 is an attempt to evaluate the above security models based on the printer example in order to retain a set of basic propositions for our future implementation of the context-based security framework. Finally, section 11 concludes this paper.

2. Problem Statement

Our study is based on a simple example: A Jini-system in which a user wants to print a confidential document using a Jini-enabled printer available on the network. However, Jini lacks a strong security model. Our research purpose is to build a security model for the example described earlier. The first step, which is the aim of this paper, is to identify the main security threats in the actual Jini infrastructure and to review the main solutions already proposed by the research community. These models may provide solutions for some specific security threats in our system. We shall use our printer service as an illustrative example to discuss each of these models, even if they were originally intended to a more general Jini-based system.

3. Conventional Jini-Based Printer Service Behavior

The rest of the discussion is based on a concrete scenario; a printer service. In this framework, a set of clients with different roles interact with the Jini service in order to print documents. For simplicity reasons we will assume that only one client is actually interacting with the printer service. The following is a typical scenario (Figure 1).

At some previous time, the printer service has instantiated a proxy and registered it in the lookup service.

1. The client wishing to use a Jini printer service

performs a lookup (in this example, searching for printer services) by contacting the lookup service. A list of available services is returned to the application.
2. The user selects the desired service. A serialized proxy object is transported to the client virtual machine and the corresponding byte code is downloaded.
3. The user calls some method on the service proxy. In this example, it sends the document and asks the proxy to print that document.
4. The proxy sends the request to the service.
5. The printer service prints the received document and sends back a confirmation to the client (in the form of a pop-up window or e-mail message).

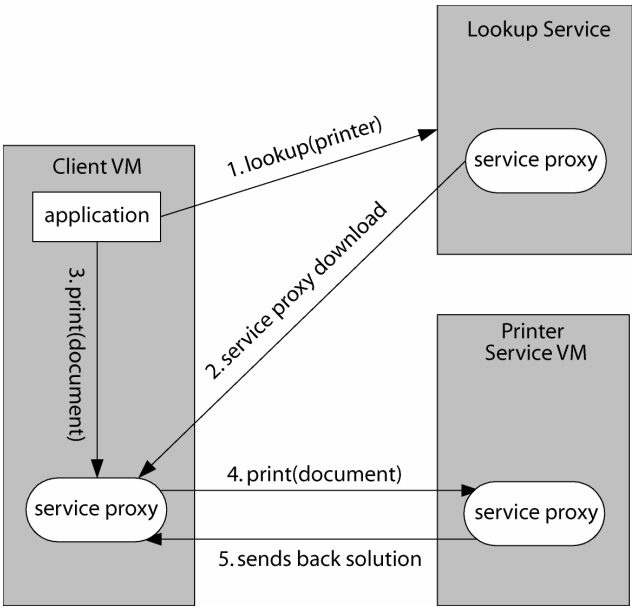

Figure 1. A Jini-based printer system.

3.1. Security Threats

In the actual Jini infrastructure and based on the above example, we can identify the following threats:

- *Interception*: It refers to the situation that an unauthorized party is listening to a communication between the client and the service. For example, if a user sends its document to the printer service, nothing prevents an unauthorized user from stealing this information. This issue is of main concern, especially in critical applications where the documents have to be kept confidential.
- *Interruption*: Actually, there is no way to prevent any user from shutting down the lookup service or any other service (the printer service in our example).
- *Modification*: It involves unauthorized changing of data or modifying a service behavior.
- *Malicious lookup services*: Nothing prevents a particular user from launching a 'bogus' lookup service that contains proxies representing services that implement the printer service interface and sends the received client document to an unauthorized user instead of printing it.
- *Malicious services*: Even if the lookup service fully trusted, it is still possible to have malicious printer services registered with this secure lookup service.
- *Malicious proxy code*: Running proxies in the client virtual machine may need special permissions. The Java security model already provides the basic mechanisms for running the downloaded code inside a "sandbox".
- *Services visibility*: It is not possible to control who is able to discover particular Jini services from a particular lookup.
- *Services access control*: In the actual Jini model, a security mechanism has to be explicitly added to Jini in order for a service to allow some of its operations and to deny others depending on the client identity (see section 4.3 for a possible solution). In our example, the printer service implements a set of operations: the *print* method which is invoked by the clients to print a document; the *modify* method that allows a user to modify the default parameters of the printer (start-up, shut-down,…etc) in addition to other operations. We want to restrict some methods such as *modify* to be invoked only by the administrator of the system. It is thus important to control access to services operations depending on the client identity.

This is a non-exhaustive list. It is based on our printer service example. We believe, though, that for other specific applications, only a subset of these requirements may be sufficient or that new considerations may be introduced. It is also important to keep in mind that some requirements may conflict.

The actual work does not cover security aspects of distributed events, leases and transactions. Identifying security threats in these cases is a complex task and is left to a future research project. However, a potential security issue could be to manage services certificate expiration depending on their lease time and to prevent deletion of events at runtime.

3.2. Security Requirements

In the following, the discussed security threats are mapped into low level security mechanisms:

- *Message encryption*: Exchanged messages between services and clients and between services and the lookup service must be encrypted to protect them from eavesdropping. Message encryption provides a solution for both, interception and modification threats.
- *Lookup service authentication*: The lookup service has to be authenticated by clients and services before any of its proxy code is executed.
- *Services authentication*: As for the LUS (Lookup Service), the clients must authenticate all other services before their code is executed. As the only interconnection of the service and the client is the service's proxy, it is straightforward to authenticate the proxy instead.
- *Proxies authentication and integrity*: Proxies identities have to be authenticated. To control if they

come from the right service. Proxies need also to be verified if they have been modified on route.

- *Access control mechanisms for local resources*: Some mechanisms are required to protect local resources such as hard disks, user information and machine file system from dangerous operations.
- *Clients authentication*: Clients may be authenticated in order to control their access rights to a given service.
- *Services visibility*: Some services should be invisible to un-privileged users. A mechanism to control services visibility is thus required.
- *Access control mechanisms for services operations*: Services may be able to control what kind of operations are allowed, based on the client identity. This solution may correct both problems, interruption and service access control.

In the remaining sections, we discuss six main approaches for securing Jini-based systems. We begin by the standard Java security model (Java sandbox, security APIs and policy files), which is not especially intended for the Jini infrastructure, but may solve some of the problems discussed before. The second approach is the result of a project elaborated at Darmstadt University of Technology and is based on a centralized security infrastructure [10]. The third approach relies on a decentralized security model and was elaborated at Helsinky University [6]. The next model makes use of self-signed certificates to secure Jini-based systems and has been introduced by Andersson and Karlsson in [2]. Another architecture has been initiated as a project at the International Computer Science Institute in Berkeley. It aims to secure the Lookup service, Jini services and to ensure message confidentiality. Finally, we present Sun Microsystems contributions. The first contribution is the Remote Method Invocation Extension. It was an attempt to add security aspects to RMI, and was intended to serve as a basis for adding security to Jini. The second contribution is the Davis Project.

4. The Java Security Model

Security features are missing in Jini. Sun refers to the underlying security features of the Java programming language (JDK 1.2), which initially ensures that an un-trusted and possibly malicious application cannot gain access to system resources (the Java sandbox). The standard security model, however, does not provide all the necessary security requirements such as authentication of participating parties, communication protocols, confidentiality and integrity of data.

To satisfy these requirements, Sun released a set of optional packages: JSSE (Java Secure Socket Extension), JCE (Java Cryptography Extension) and JAAS (Java Authentication and Authorization Service) which are now integrated into the actual Java 2 Software Development Kit (J2SDK) v 1.4.0. However, even if these packages are now part of the Java SDK, they are not yet part of the Jini Technology Starter Kit v 1.2. Therefore, their contribution for securing Jini-based systems has to be explicitly included.

At time of writing, the Jini Team at Sun Microsystems is working on a security model for Jini, known as the Davis Project [21]. Since this project is still under development, instead of presenting it as a standard part of the Java security model, we will discuss its main features in section 9.2.

4.1. The Java Sandbox

The Java security model restricts running downloaded code to its own sandbox. Thus, the Java virtual machine allows for executing un-trusted applications in a safe environment. The Java sandbox is a set of three interrelated components: the class loader, the byte code verifier and the security manager.

1. *The class loader:* It is the first line of defense in the Java security model [22, 34]. The class loader is responsible for importing the code from the remote machine, defining Java namespaces in order to isolate trusted class libraries (Java APIs) from un-trusted ones, and verifying that the code has the appropriate permissions in order to access or define classes. The last functionality is achieved by cooperating with the security manager. A JVM (Java Virtual Machine) may run multiple class loaders; each class loader has its own namespace.
2. *The byte code verifier:* It is built into the virtual machine and cannot be accessed by Java programmers or Java users [13]. A Java program is compiled down to platform-independent Java byte code contained in class files. Before the byte code is run into the virtual machine, a set of tests are applied to it by the verifier to ensure that the incoming byte code stream conforms to the specifications of the virtual machine [35]. The byte code verifier checks for: stack overflow, type correctness, class format correctness, illegal casts, pointer forging and protected class access.
3. *The security manager:* It is the most important component of the Java sandbox and serves as a guardian for its boundaries. The security manager is a Java object; a subclass of the *java.lang.SecurityManager* class that is consulted by the Java code before any potentially dangerous operation is executed. The main role of the security manager is to control access to protected resources such as files and personal data, to control all socket operations and to prevent the installation of new class loaders [22]. Developers may customize the security manager to a specific security level depending on their applications. Figure 2 illustrates the collaboration of the three Java sandbox components in a concrete case: the Printer service proxy downloaded from the lookup service in order to be executed in the client virtual machine. The proxy code is first checked by the byte code verifier, then, it is loaded into a namespace by the class loader in

order to prevent access to resources the proxy does not have the right to see. All the operations initiated by the printer service proxy are controlled by the security manager.

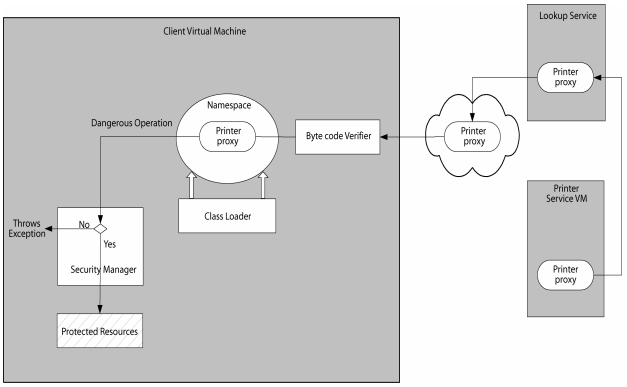

Figure 2. The Java sandbox components.

4.2. Java Security APIs

Sun has released optional packages to support additional security features for the standard Java security model, such as encryption, authentication and authorization. Since version 1.4.0, these packages are a core part of the Java 2 Software Development Kit (J2SDK).

1. *JCE*: The Java Cryptography Extension [23] is a set of security packages from Sun. It supports data encryption, key generation and key exchange. The JCE framework allows new cryptography libraries and algorithms to be added seamlessly.
2. *JSSE*: The Java Secure Socket Extension [24] is a standard package that provides a Java implementation for the Secure Socket Layer (SSL) and
3. *JAAS*: The Java Authentication and Authorization Service [25] enforces the Java security model by enabling user-based, group-based and role-based authentication and access control.

Figures 3, 4 and 5 show where each of the security APIs described above may contribute.

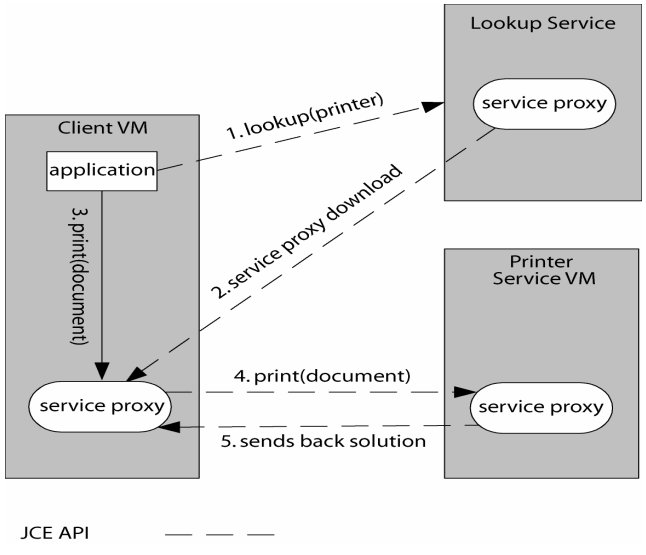

Figure 3. The JCE API provides tools to encrypt data and to ensure message integrity between the client and the lookup service, as well as between the client and the printer service.

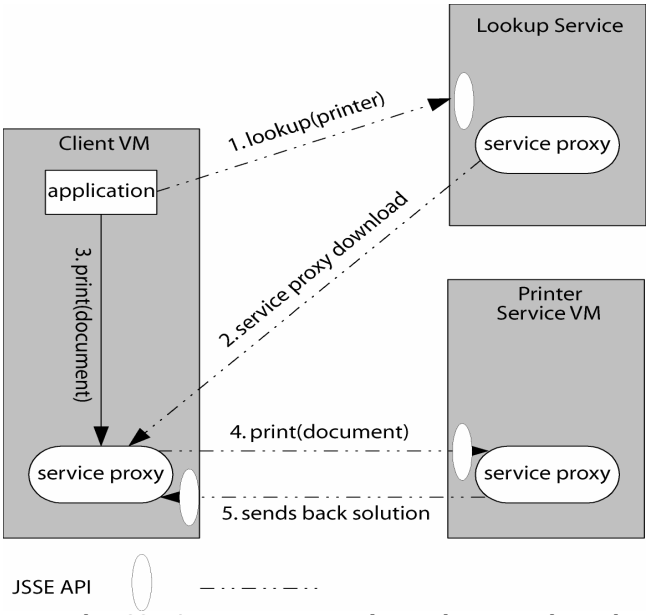

Figure 4. The JSSE API ensures secure data exchange -at the socket level- between clients and services. It provides also ways to authenticate servers (services providers) and clients.

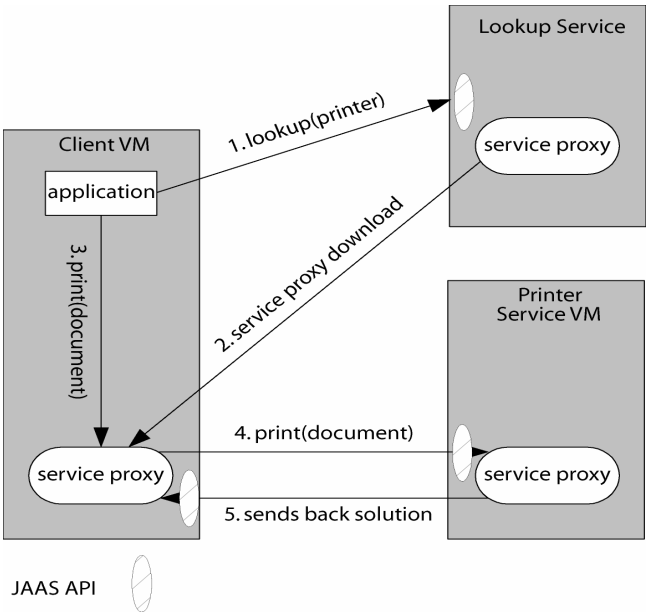

Figure 5. The JAAS API manages user authentication and access control.

4.3. Java Policy Files

Policy files do not implement a new security model by themselves, but are rather static configuration files used by Java applications in order to specify what permissions (access to a system resource) are given to Java code depending on its source (location), the signer of the code, or both. Version 1.4 of the J2SDK includes new protection mechanisms by allowing a new policy implementation. This implementation supports principal-based grant entries, which means that the code is also considered to be run by a specified user (principal) [26]. Policy files have a concise syntax. In order to eliminate the need to know this syntax, Sun provides a graphical *policytool* utility that comes with the JDK for creating and editing policy files. They may also be edited by hand using any text editor.

The syntax of a java policy file is the following:

grant signedBy "signer_names", codeBase "URL", principal principal_class_name "principal_name", principal principal_class_name "principal_name", …{
permission permission_class_name "target_name", "action",
signedBy "signer_names";
permission permission_class_name "target_name", "action",
signedBy "signer_names";
…
};

Here is a sample policy file:

grant signedBy "Mike", codebase "http://www.unifr.ch", principal javax.security.auth.x500.x500Principal "cn=Olivia"{
permission java.io.FilePermission "/home/Java", "read,write";
};

This allows code signed by "Mike", downloaded from "http://www.unifr.ch", and executed by "cn=Olivia", permission to read and write into the directory "/home/Java". Java allows users to implement their own permissions. However, it comes with a set of built-in permission types. A *java.io.FilePermission* represents access to files and directories. Its corresponding actions are *read*, *write*, *execute* and *delete*. A *java.net.SocketPermission* represents access to a network via sockets.

Given a host specification, this permission allows the following actions: *accept*, *connect*, *listen* and *reprint*. We refer the interested reader to [27] for an in depth discussion about Java built-in permissions.

5. A Centralized Jini Security Model

The authors Hasselmeyer, Kehr and Voss in [10] propose an extension to the Jini architecture, which enables secure lookup of services and trust establishment between parties involved in a Jini federation, namely services and clients. It relies on an off-line central certification and authentication authority. In order to ease the administration of access rights, the notion of "groups" is introduced. This notion allows restricting the visibility of services registered at the lookup service.
Two main components are added to the initial Jini infrastructure (Figure 6):

- *Certification Authority (CA)*: It provides certificates for authentication of all participants (services and clients). For security reasons, the CA is implemented as a stand-alone application. In a real-world environment, it should run in a physically secured place on a machine with no connection to the internet (i.e. outside the secure intranet).
- *Capability Manager (CM)*: The capability manager is implemented as a separate Jini service. It administers a list of names and the associated access rights (capabilities) for each user. All the information has to be given to the CM by an administrator. The CM plays the role of a delegate of the manager that is responsible for handing out capabilities. Capabilities are transferred as Java signed objects (java.security.SignedObject). Capabilities provided to services and clients are used for access control in the lookup service (for a service during a registration phase, and for a client when looking up for a specific service).

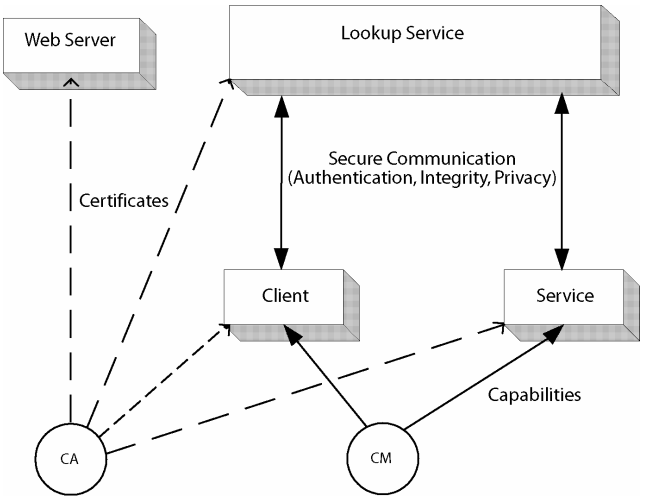

Figure 6. The overall architecture of Hasselmeyer's centralized security model.

5.1. Implementation

This new architecture requires some modifications in the source code of the lookup service implementation and the classes that handle the discovery protocols.

First modification (Secure Lookup Service Discovery): Looking up services or joining a federation requires interaction with the LUS, which might be malicious. Therefore, the lookup service has to be authenticated to its clients before any of its proxy code is executed. This concerns the lookup service implementation. Sun's implementation of the lookup service is called *Reggie* (package com.sun.jini.Reggie). It consists of two parts: the actual directory service *RegistrarImpl* which is the class of Reggie's server implementation, and a proxy object *RegistrarProxy* used by clients to access the Reggie's server. These two parts communicate via Java RMI mechanism. The idea is to protect the RMI message exchange by tunneling RMI traffic through the SSL (Secure Socket Layer) protocol [32]. An SSL socket is then used instead of the standard socket. The SSL protocol ensures privacy, identity authentication and message integrity between the client/service and the lookup service.

Second modification (adding groups and capabilities): It consists of modifying the lookup service functionality (*ServiceRegistrar* in package net.jini.core.lookup), by adding new lookup and register methods, which take the user's capability and a group name as additional parameters. We see in the next

section how these capabilities and groups are used in order to manage visibility of services.

5.2. Example Scenario

In the following, we use the same example presented in section 2. The actual scenario consists of a printer service which wants to register itself in the group "protected services" (Figure 7) and a client, which performs a lookup in this group.

Pre-configuration: We assume that all certificates and capabilities have already been set up and that the capability manager is registered at the LUS in the special group "capability".

Service Registration

1. *Lookup service discovery*: The service sends a unicast discovery request message and gets an extended response from the lookup service. This response contains the signature for the lookup service proxy and the signer's certificate. Before using the proxy object, the printer service checks the certificate and the signature.
2. *Secure communication/authentication*: (not shown in Figure 7) The lookup service proxy establishes a secure communication session between the printer service and the lookup service with mutual authentication. The connections between the lookup service and its clients (services/users) should be encrypted to prevent unauthorized parties from observing the in-traffic service descriptions.
3. *Capability manager lookup*: Before registering itself in the LUS, the printer service must look for the capability manager (CM) in order to obtain capabilities. The printer service calls the LUS proxy's lookup method with the parameter "capability" to find an instance of the capability manager.
4. *Obtaining capabilities*: The printer service asks one of the CMs for its capabilities. The CM consults its database and creates an adequate capability object containing the permissions of this service. The capability is delivered inside a signed object using the CM's private key to guarantee its authenticity.
5. *Registering at the LUS*: The printer service calls the LUS proxy's register method with additional parameters: the desired group "protected services" and its signed capability. The capability is only accepted if the contained name equals the distinguished name presented during the authentication phase. Upon success, the LUS adds the service description to this group.
6. *Client side service lookup and use*: steps 1 to 4 are the same as above. The next steps are described below and illustrated in Figure 8.

Service lookup

1. The client calls the LUS proxy's lookup method with the group "protected services" and its signed capability as additional parameters. The LUS verifies the capability and checks if the permission for the specified group is implied. Upon success, it returns all services of this group which match the given service template (printer services).
2. *Service use*: The client selects one service from the result and uses the service proxy for further interaction.

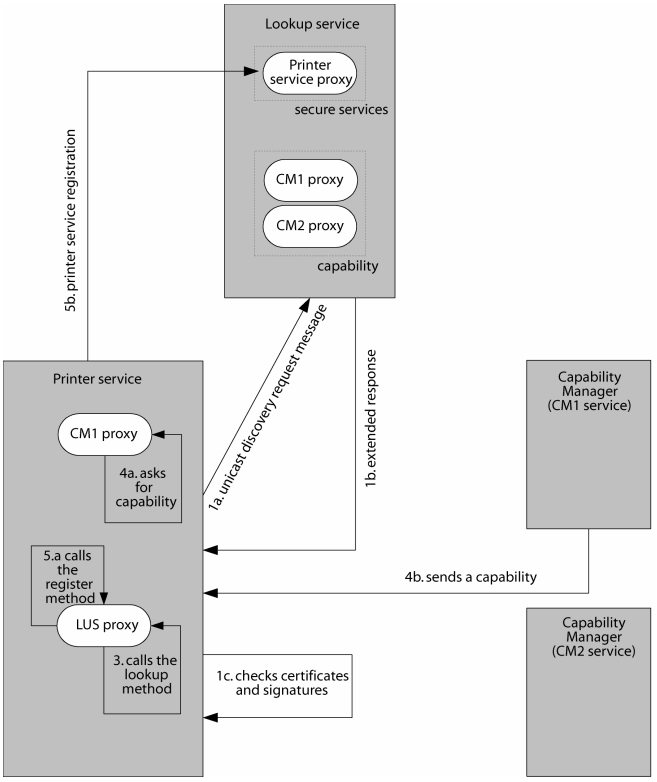

Figure 7. Service registration in Hasselmeyer's centralized security model.

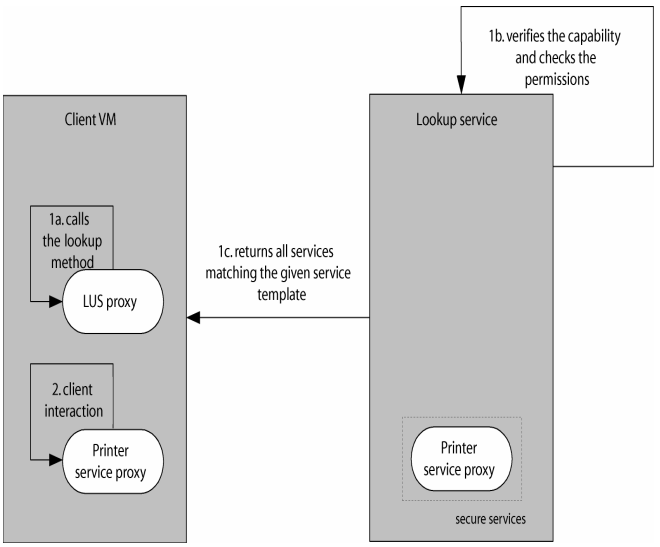

Figure 8. Client side service lookup and use.

5.3. Advantages

This solution presents the following advantages:

- It ensures communication privacy and data integrity by the mean of the SSL protocol.
- It ensures lookup service authentication.
- It ensures services authentication.
- It provides proxies integrity by the mean of data encryption between the lookup service and its clients (Jini users/services).

- It ensures clients (Jini users) authentication.
- It introduces the notion of "groups" in order to control services visibility to clients.
- It provides mechanisms to protect local resources of the client machine.
- It offers a fine-grained authorization scheme based on capabilities.

5.4. Limitations

This solution presents the following limitations:

- In order to be able to verify the identity of the communication partner, the certificate of the other end has to be signed by a commonly known certification authority (implemented especially for the purpose of this architecture) meaning that all communication partners need an *a priori knowledge* of the certification authority's public key. This is not a usable way to check the certificate's validity in Ad Hoc networks, since it introduces a partial loss of "spontaneity" of client/service interactions, which is one of the main advantages of Jini.
- Before interacting with this extended Jini infrastructure, a pre-configuration phase must be accomplished (setting up certificates and capabilities and registering the capability manager at the LUS in the special group "capability"). The system has to be unavailable for other services and clients during this time. There is no indication how to prevent services and clients from interacting with the LUS during this phase.
- The proposed solution is bound to a specific communication protocol SSL, thereby hampering the protocol independence of Jini.
- There is no mechanism to control access to the operations of a given service.
- It requires modification of the Jini source code.

6. A Decentralized Jini Security Model Based on SPKI

Respecting the had hoc nature of Jini, the model proposed by Eronen in his master thesis [6], as well as in other papers [7, 9], presents a fully decentralized security architecture for Jini based on trust management [3, 8]. It uses Simple Public Key Infrastructure (SPKI) certificates for authorization [20], and provides access control for Jini clients, service proxies and services. This implementation relies on the Java 2 security model and the Java Socket Security Extension (JSSE).

6.1. Implementation

In this approach, clients and services are identified by public keys. These keys do not have any centralized certification infrastructure. Anyone can start a service and create a new key pair for it.

- *Proxies verification*: One of the key assumptions in this design is that if a service signs a proxy, this does not guarantee that either the proxy or the service itself is not malicious, but only that the signer service trusts the signed proxy. Furthermore, no authority certifies that the service itself is "well behaved".
- *Clients authorization*: In order to grant authorization to use a service, the actual infrastructure uses SPKI (Simple Public Key Infrastructure) chains. When actually accessing a service, the chain is completed to a loop.
- *Certificate Chaining*: The printer service maintains an Access Control List (ACL) which contains a set of valid clients allowed to access. For instance:

(acl (subject UNIFR)(tag access))
(acl (subject administrator)(tag access modify))

The service's ACL says that only UNIFR[1] users are allowed to access and that only the administrator is allowed to access and modify the printer service. Now, let us consider a client "Mike", who has a set of certificates:

- a certificate saying that he is a student at the DIUF[2]
- a second certificate saying that DIUF students are UNIFR students.

Mike wants to use the printer service. Here is how the authorization phase is performed:

1. Mike sends first his signed request to the printer service.
2. The service checks its ACL and rejects Mike's request by sending back its ACL to Mike.
3. Using the certificates he already owns, Mike performs a certificate chain discovery starting from an ACL entry and ending with his public key. This sequence is of the form: *Mike's public key - DIUF's public key - UNIFR's public key - access Printer Service*
4. A second request is then sent to the printer service with the chain.
5. The service then authorizes Mike (his public key) to perform the requested operation (see Figure 9).

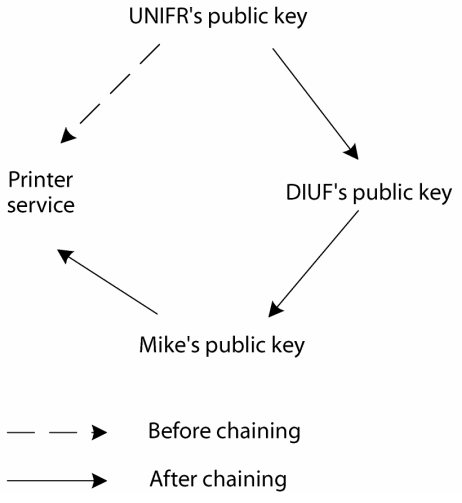

Figure 9. A simple SPKI chain.

[1] University of Fribourg
[2] Department of Informatics of the University of Fribourg

6.2. Example Scenario

The following scenario consists of a user requesting a printer service to print a document (Figure 10).

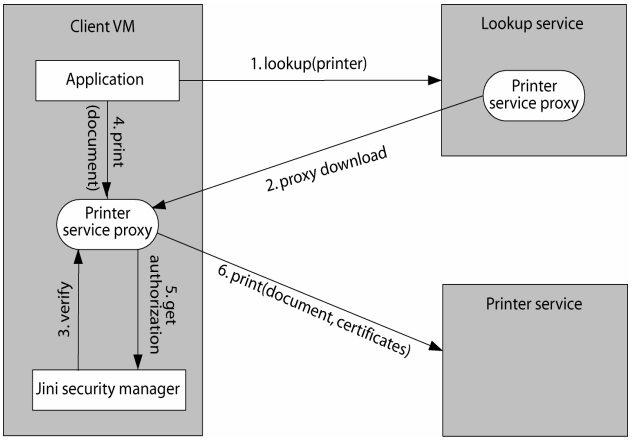

Figure 10. Accessing a Jini service with Eronen's decentralized security model.

Service Registration: the service registration steps are not shown in Figure 10.

1. *Lookup Service Discovery*: The printer service sends a unicast discovery request message and gets a conventional response from the lookup service.
2. *Registering at the LUS*: The printer service signs the proxy using its private key and registers the proxy to the lookup service.

Client side service lookup and use: the discovery step is the same as above.

1. *Service lookup*: The application calls the lookup method of the LUS proxy to perform an appropriate lookup of the desired service (printer services in our example). A list of services is returned to the application. No special security features are assumed here.
2. *Obtaining the service proxy*: The application (the user) selects one of the listed services. A serialized proxy object is transferred to the client and the corresponding byte code is downloaded.
3. *Service proxy verification*: The client security manager (Jini security module) asks the proxy for the printer service's public key and checks that this proxy indeed represents that service. An additional proxy authentication step is performed in order to verify that the name of the printer service shown to the user is correct.
4. *Service method invocation*: The application calls a given method on the proxy object. In this example, it asks the proxy to print a document.
5. *Proxy authorization (application access control)*: The proxy asks the client security manager for authorization. The security manager checks that the proxy is really trying to access the service it represents and that the application is allowed to perform this operation on behalf of the user.
6. *Service use*: The proxy opens a secure connection to the server by implementing any protocol it chooses. The actual architecture uses RMI over TLS (Transport Layer Security) [32]. The proxy sends the certificates and the service request to the server. The server security manager checks the certificate chain using the public key of the proxy and its certificates. Upon success, the service performs the requested operation.

6.3. Advantages

This solution presents the following advantages:

- It ensures communication privacy using the TLS protocol.
- It ensures services authentication by the mean of digital signatures.
- It ensures proxies authentication by verifying that it was signed by the service back-end key.
- It provides clients authentication by the mean of SPKI certificates.
- It ensures protection of local resources of the client machine.
- In this architecture, no central certification authority is required. It relies on a simple trust policy model using SPKI certificates.
- It allows delegation. For instance, a student at the DIUF does not need to have an explicit authorization to use a service. This authorization is deduced from the ones he already owns from DIUF and UNIFR (see section 6.1).
- Authorizations are specified in flexible user-defined tags using ACLs.
- There is no modification of the Jini source code.

6.4. Limitations

This solution presents the following limitations:

- This model does not ensure lookup service authenticity.
- There is no mechanism to control services visibility.
- There is no mechanism to control access to the operations of a given service.
- The use of SPKI certificates introduces some latency problems, since the chain discovery consumes time. This of course depends on the complexity of the Jini system.

7. A Decentralized Jini Security Model Based on Self-Signed Certificates

To avoid the requirement of having a central CA (Certification Authority), Andersson and Karlsson in [2] suggest the use of self-signed certificates to authenticate both services and clients. In order to check the certificates validity, however, the receiver must calculate the fingerprint[3] and check if it matches the fingerprint

[3] A fingerprint is a sequence of characters computed from the contents of the certificate. It uniquely identifies the certificate as being genuine. For example: 85:67:3B:72:D8:4A:CE:83:F4:10:44:C4:E0:C8:BE:43

received earlier (printed on the back side of a business card). A key exchange algorithm used to encrypt/decrypt data is implemented as a Jini service to make it easily available to other services. Its proxy (the key proxy) is automatically downloaded as a part of other services proxies. This work has been done at Ericsson Research Communication Security Lab in Kista, Sweden.

7.1. Implementation

The authors provide their own implementation of the algorithms used to encrypt/decrypt data. Thus, the security model is entirely built using the Java standard library and does not need extra Java libraries, such as JCE or JAAS.

7.2. Example Scenario

We use the same example as in the previous sections. Namely, a user requesting a printer service to print a document.

Pre-configuration: At previous time, the clients and services providers have exchanged their business cards. On the back of each card, the fingerprint of their certificate is printed.

Service Registration

1. *Lookup service discovery*: The printer service sends a unicast discovery request message and gets a conventional response from the lookup service.
2. *Signing the proxy code*: The printer service bundles the proxy code into a jar file for faster transfer. This jar file is then digitally signed using the jarsigner utility that comes with the JDK. The jar file contains the proxy code and the service certificate.
3. *Registering at the LUS*: The printer service registers its proxy to the Jini lookup service.

- *Client side service lookup and use*: The registration phase is the same as above.

The remaining steps are detailed below and illustrated in Figure 11.

1. *Service lookup*: The client performs a lookup to find printer services. A list of them is returned to the client.
2. *Obtaining the service proxy*: The client downloads the printer service proxy (along with the service's certificate). The key-exchange proxy is automatically fetched with the printer proxy (see item 4).
3. *Service proxy authentication*: The client checks the certificate fingerprint used to sign the proxy code. The authentication passes when the services certificate fingerprint received with the proxy and the one already available at the client side (printed in the backside of the service provider business card) are equal. These fingerprints are authenticated by the mean of a GPen; a digital assistant that reads and processes printed text.
4. *Secure communication between services and clients*: In order to ensure privacy and integrity of the communications, exchanged data are encrypted using the Diffie-Hellman algorithm. As its name suggests, the key-exchange proxy (already present in the client VM) is responsible for creating the keys used for encryption and decryption. The key-exchange proxy acts on behalf of the client, so the client has to be authenticated. The same technique described earlier is used; comparing the client's certificate fingerprint received by the server and the one already present at the server side.
5. *Service method invocation*: The client application calls some method on the proxy object (print the document).

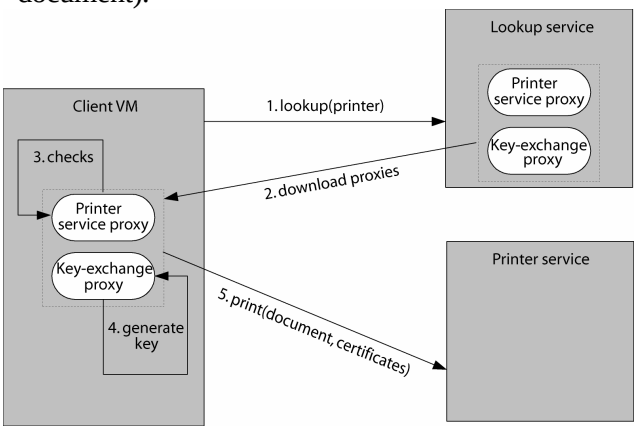

Figure 11. Accessing a Jini service with self-signed certificates (Andersoon's model).

7.3. Advantages

This solution presents the following advantages:

- Data privacy and integrity between clients and services is ensured (using the Diffie-Hellman algorithm).
- This model provides services authentication mechanisms that relies on certificate fingerprint check.
- Proxies are also authenticated by verifying their signature.
- This model provides also clients authentication in the same way as services.
- It provides ways to protect local resources of the client machine.
- Encryption functionalities are implemented as a Jini service, which eases further changes and updates.
- This model does not rely on a central certification authority which is more adequate for ad hoc networks such as Jini networks.
- There is no modification of the Jini source code.

7.4. Limitations

This solution presents the following limitations:

- The lookup service is not authenticated.
- No services visibility mechanism is provided.
- Access control to the operations of a given service.
- Even if this model does not rely on a central certification authority, the management of trust is

based on an a priori exchange of certificates fingerprint printed on the back of business cards. This limits a bit the spontaneity of Jini.

8. An Authentication and Authorization Architecture for Jini Services

The main part of this architecture [17, 18] has been developed at ICSI (International Computer Science Institute) in Berkeley, California. The authors focus on the following three security goals: providing services authentication and authorization mechanisms, message confidentiality and client authentication. The main concern was to develop a security architecture transparent to clients, which means that the existing Jini clients code do not need to be modified. This is achieved by packing all the security functionalities into the service proxy and the service back-end. A simple policy file at the client side is needed to make sure that the received proxies are signed by trusted parties before executing them in the client virtual machine.

8.1. Implementation

This security model for Jini is developed using the cryptographic functionalities provided by the JCE 1.2.1 API, the authentication and authorization mechanisms offered by the JAAS 1.0 API, and JavaCard 2.0 [28], an API that enables to run java on devices with limited memory. The main parts of this architecture are:

- *SubjectAuthenticatorService*: Implemented as a Jini service, the *SubjectAuthenticatorService* is the central entity of the whole security infrastructure. Its role is to manage communications between the *LoginPolicyDB*, the *RemoteCallbackHandler* and the *UserDB* in order to handle the authentication process [17].
- *RemoteCallbackHandler* : It is instantiated in the client virtual machine and initiates a login interface to authenticate the user.
- *LoginPolicyDB*: is a Jini service. Its task is to return what login policy should be used to log on a user. The returned login policy depends on the input parameters such as the identity of the client host and a string representation of the service to be used.
- *UserDB*: It is implemented as a Jini service and its role is to authenticate the user by verifying the data (username/password) he provides via the *RemoteCallbackHandler*.

8.2. Example Scenario

Based on the printer service example used until now, we present in the following the usual scenario using the security model presented in [17, 18] in a slightly simplified manner. This scenario is illustrated in (Figure 12).

Pre-configuration: At previous time, the clients and services providers have exchanged their certificates. The trusted certificates (signed by a well-known certification authority) are added to a Java keystore.

Service Registration (not shown in Figure 12)

1. *Lookup service discovery*: The printer service sends a unicast discovery request message and gets a conventional response from the lookup service.
2. *Signing the proxy code*: The printer service generates a key pair. The private key is used to sign the proxy and remains at the service back-end. The public key is transferred to the service proxy.
3. *Registering at the LUS*: The printer service registers its signed proxy to the Jini lookup service.

Client side service lookup and use: The lookup discovery step is the same as above.

1. *Service lookup*: The application calls the LUS proxy's lookup method to perform an appropriate lookup of the desired service (printer services in our example). A list of services is returned to the application.
2. *Obtaining the service proxy*: The client application selects one of the listed services. A serialized proxy object is transferred to the client and the corresponding byte code is downloaded.
3. *Secure communication between the service and its proxy*: To ensure communication confidentiality between the proxy and its back-end, messages are encrypted using the Diffie-Hellman algorithm. So both the proxy and the service back-end must agree on a symmetric secret key. In the client JVM, the proxy generates its own key pair. Using its own private key and the back-end public key, it generates a symmetric secret key [4]. The proxy sends its own public key to the service back-end, so that the back-end can generate the secret key and stores it in a session database.
4. *Service proxy authentication*: Using the public key of the printer service back-end, the client authenticates the signed proxy.
5. *Service method invocation*: The client application calls some method on the service proxy. In our example, it asks the proxy to print a document.
6. *Client authentication*: When the printer service receives a request via its proxy, it contacts the *SubjectAuthenticatorService*. The *SubjectAuthenticatorService* then requests a login policy from the login policy database: LoginPolicyDB. This login policy specifies how to authenticate the user. The *SubjectAuthenticatorService* sends the login policy to the *RemoteCallbackHandler* (instantiated by the proxy at the client side). The *RemoteCallbackHandler* initiates an authentication scheme depending on the login policy. The client is then prompted for a username and a password. More sophisticated authentication schemes may be implemented. The information provided by the client to the *RemoteCallbackHandler* is then sent back to the *SubjectAuthenticatorService*. The latter contacts the

user database: UserDB. The UserDB checks if the data supplied by the user is correct and returns the data needed to build a *Subject.* A subject is then built by the *SubjectAuthenticatorService* and is returned to the service. The Service returns a token to the proxy, to be used by the client.

7. *Service use*: The proxy opens a secure connection with the service (see point 3) and invokes the desired method *print()*. In this security architecture, access control to the operations of a given service is implemented. The printer service back-end has its own policy file that specifies the permissions granted to users for invoking an operation. Each code that needs authorization is encapsulated into a *run* method of an extension of a *PrivilegedAction* class. In order to perform an action that need authorization, the service runs the *Subject.doAs()* method for the client. This method checks if the appropriate permissions have been granted to the given client. If not, an exception is thrown (see section 5.4.3 in [18] for more details).

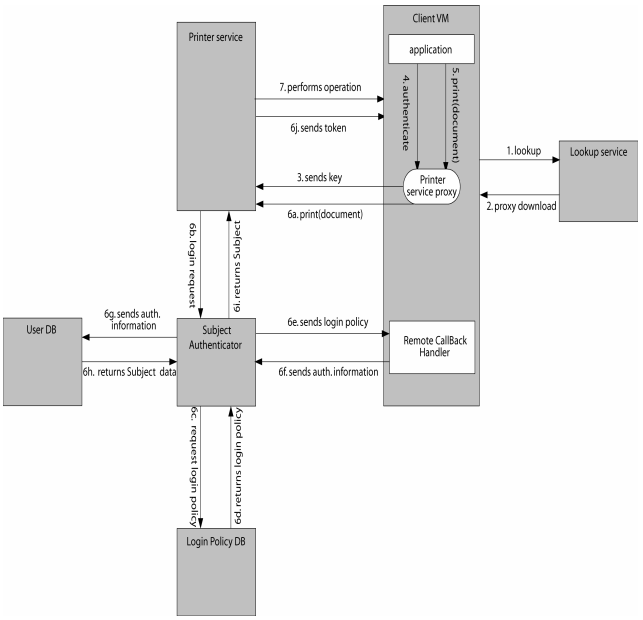

Figure 12. Components of the security architecture (Schosh's model).

8.3. Advantages

This solution presents the following advantages:

- This security architecture provides communication encryption using the Diffie-Hellman algorithm.
- It ensures services authentication.
- It ensures proxies authentication.
- It ensures clients authentication.
- It ensures control of local resources of the client's machine.
- Actually, this security infrastructure is the only one that affords access control to services operations.
- It achieves clients transparency; which means that existing clients do not need to change their code to fit within the security infrastructure.
- It provides flexible login policies that allow anonymous and single sign-on [18].
- All security components are implemented as Jini services which eases further modifications in the security infrastructure.
- There is no modification of the Jini API code.

8.4. Limitations

This solution presents the following limitations:

- In this architecture, certificates of communicating parties have to be signed by commonly known certification authorities meaning that all communication partners need an *a priori knowledge* of the public keys of the certification authorities. This certificates distribution scheme is, however, more flexible than the one presented in section 5 since it relies on CAs whose public keys are already known by the Java environment; the JDK comes with a set of ten CAs public keys.
- It does not secure the lookup service.
- This security architecture does not provide any mechanism to control services visibility.
- This security architecture relies on a central CA (certification authority) which limits a bit the spontaneity of Jini. Certificates are stored in a Java keystore then fetched from it when needed.

9. Sun Solutions for Jini

In this section, we present the solutions proposed by Sun Microsystems to secure Jini. The first attempt was the Remote Method Invocation Extension. It was intended to add security to RMI and to use these features to secure Jini. This project was rejected. We will, however, present its initial objectives and the reasons of its rejection. The second attempt is the Davis Project. It is intended to build a security model for Jini and is actually in a development phase. We discuss its main features in section 9.2.

9.1. Remote Method Invocation Security Extension

This standard extension is identified by the Java Specification Request JSR 76 [31]. Even if the original specification was intended to add security to RMI, it is the basis for adding security to all types of remote services defined in terms of interfaces like Jini. It builds on JAAS (Java Authentication and Authorization Service) and defines a high level API, where the implementation of cryptographic mechanisms and protocols are not exposed, so code written to the API is more portable.

The RMI Security Extension allows:

- mutual authentication between the server and the client during remote calls
- communication integrity
- information confidentiality
- delegation

- secure registry
- Unfortunately it was rejected for several reasons. The first one is clearly the main one and is mentioned on the JSR 76 website [31]. The three next ones are discussed in [10] and their pertinence would require further investigations.
- There is no separation between the security information and the business logic. This means that, in this specification, the security functionalities have to be a core part of the application code.
- Trust establishment is performed only when the objects have been already instantiated, thus, security holes may exist in the constructor of the proxy.
- The specification is intended to add security to RMI in general and does not address Jini issues in particular. For instance, there is no way to control Jini services visibility.
- Different security levels can only be enforced after downloading the service proxies and depend on their enforcement by each client.

9.2. The Davis Project

The Davis Project [21] is a recent attempt from the Jini Project team to create a security model for Jini. The main security requirements for this project are to ensure message integrity, provide message confidentiality and allow mutual authentication between the client and the server. It allows also the implementation of a secure lookup service in the same way as other secure services. This architecture is not bound to a specific implementation. It is intended to support plugging different protocols and algorithms to be used by the network security programming model. At time of writing, the implementation of the Davis Project is not yet complete. An overture release (v 0.05), however, is available for download [29]. This release contains the basic Davis security architecture. It does not define new protocols or algorithms to support the security requirements described above, but is actually based upon JSSE (Java Secure Socket Extension). The actual early access release contains the following main components: A network security programming model and API for remote calls and to support exporting remote objects, extensions to the RMI activation framework to support the network security programming model and tools for generating message digests [29].

10. Evaluation

This section presents an evaluation of the four working security models discussed earlier, namely, the centralized model, the SPKI-based model, the self-signed certificates-based model and the authentication and authorization security architecture. This evaluation is based on the security criteria we discussed in section 3.2 along with a set of design requirements relevant to our context. These additional requirements are: modification of the Jini source code and the certificates distribution scheme. We deduce from Table 1 that message encryption has been realized by all the security models, even if the encryption protocols differ from a model to another. Access to the client's local resources is controlled using the standard Java security mechanisms and policy files. Clients, services and proxies authentication is based on certificates and digital signatures. Certificates distribution may be centralized or decentralized. Lookup service authentication and services visibility are realized only in the first architecture. These requirements, however, costs some modifications of the Jini source code and re-implementation of the Lookup service. Access control to the individual operations of a given service has been implemented in the authentication and authorization security architecture (see section 8), by defining new permissions specific to each service operation. Based on the actual stand of our evaluation, we state that none of the above models fits entirely with our context, we may, therefore, combine a set of functionalities from each model in order to build our security framework.

This list of functionalities can be considered as preliminary design requirements, but in no case as final implementation decisions:

Table 1. Evaluation of security models for Jini.

Security and design requirements	Centralized Model	SPKI-Based Model	Self-Signed Certificates Model	Authentication and Authorization Model
Message encryption	x	x	x	x
Lookup service authentication	x	-	-	-
Services authentication	x	x	x	x
Proxies authentication	x	x	x	x
Access control for local resources	x	x	x	x
Clients authentication	x	x	x	x
Services visibility	x	-	-	-
Access control for services operations	-	-	-	x
---	---	---	---	---
No modification of the Jini API code	-	x	x	x
Certificates distribution scheme	Centralized	Decentralized	Decentralized	Centralized

- Our design may consider data encryption between all the parties included in the communications. Namely, between Jini users and services and between the lookup service and its clients (users/services). We tend for the use of an asymmetric encryption algorithm that make use of a pair of keys; one to encrypt data and is public and the second to decrypt data and is kept secret. Even if symmetric (en)decryption is much faster then an asymmetric (en)decryption, the latter is more reliable.
- The lookup service needs to be authenticated in order to prevent the intrusion of malicious lookup services. This will be done using certificates and digital signatures.
- Jini services and Jini clients need to be authenticated in the same manner as the Lookup service.
- The proxies need to be authenticated as being sent by the services they claim to belong. This will be achieved by signing the proxy by its service back-end. This authentication phase may be performed twice, during the registration phase into the lookup service and during the download process of the proxy in the client VM.
- We will use the facilities available in the Java security model (see section 4) in order to protect local resources of the client VM.
- We would like to afford services visibility and access control to services operations. The first requirement, however, need further intensive research on our part.
- One of the main design requirements in our framework is to preserve the Jini code from internal modifications.
- Our Jini-based system is deployed in a local area network and a user role is not related to a fixed identity, but rather to an authorization key (anonymous authorizations). Therefore, We tend for the use of SPKI (Simple Public Key Infrastructure) which is a more flexible scheme for building authorizations and distributing certificates.
- An additional design requirement would be to pack all the security functionalities into an additional Jini service (see section 8) in order to ease further updates and changes.
- Since the security APIs such as JCE, JSSE and JAAS are now standard and included in the JDK v.1.4.0, there is no need to add additional packages. We will then use the functionalities they provide in order to implement our security requirements.
- We would like our framework to afford a customizable security policy at runtime. This design requirement avoids recompilation of the whole framework.

11. Conclusion and Future Work

Jini is an elegant framework for building highly dynamic distributed environments. The actual state of the technology, however, does not provide additional tools to avoid the security threats that exist in such environments beyond the standard Java security model. In this paper, we detailed the main architectures that have been proposed to secure Jini. We based our study on the printer example in order to ease the evaluation of these security architectures. The evaluation phase allowed us to establish a primary list of design considerations for implementing our security framework. Some other attempts have been made but are more application-specific. In [1], El-Muhtadi and al. present a security model based on the combination of Tiny Sesame and Jini. Tiny Sesame is a component-based Java implementation of a subset of Sesame [11], which is itself an extension to Kerberos [14]; a network authentication protocol created at the MIT. This model is intended to be used with Jini-enabled devices in a smart home environment. It relies on non-standard Java APIs which forbids it from fitting entirely with our purpose. The next step in our work is to build our security framework based on the design requirements identified in section 10 with a great emphasis on the clarity of the software design. This implies the potentially use of security patterns [19, 36], a recent methodology in the software engineering discipline. Another future direction is to identify security requirements related to leases, distributed events and transactions.

Acknowledgments

We would like to thank Thomas-Marcus Schoch and Peer Hasselmeyer for their comments on the earlier drafts of this paper.

References

[1] Al-Muhtadi J., Anand M., Mickunas M. D., and Campbell R., "Secure Smart Homes using Jini and UIUC SESAME," *in Proceedings of the 16th Annual Computer Secrity Applications Conference (ACSAC'00)*, New Orleans, Louisiana, pp.77-85, 2000.

[2] Andersson F. and Karlsson M., "Secure Jini Services in Ad Hoc Networks," *Master Thesis*, Royal Institute of Technology, Stockholm, 2000.

[3] Blaze M., Feigenbaum J. and Lacy J., "Decentralized Trust Management," *in Proceedings of the IEEE Symposium on Security and Privacy*, pp. 164-173, 1996.

[4] Diffie W. and Hellman M. E., "New Directions in Cryptography," *in IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644-654, 1976.

[5] Edwards W. K., *Core Jini*, Prentice-Hall, 2nd Edition, 2001.

[6] Eronen, P., "Security in The Jini Networking Technology: A Decentralized Approach," *Master Thesis*, Department of Computer Science, Helsinki University of Technology, 2001.

[7] Eronen P., Gehrmann C., and Nikander P., "Securing ad hoc Jini services," *in Proceedings of the 5th Nordic Workshop on Secure IT Systems (NordSec'2000)*, Reykjavik, Iceland, pp. 169-177, 2001.

[8] Eronen P., Lehtinen J., Zitting J., and Nikander P., " Extending Jini with Decentralized Trust Management," *in Proceedings of the 3th IEEE Conference on Open Architectures and Network Programming (OPENARCH'2000)*, TelAviv, Israel, pp. 25-29, 2000.

[9] Eronen P. and Nikander P., "Decentralized Jini Security," *in Proceedings of the Network and Distributed System Security Symposium (NDSS'2001)*, pp. 161-172, 2001.

[10] Hasselmeyer P., Kehr R., and Voss M., "Trade-offs in a Secure Jini Service Architecture," *in Trends Towards a Universal Service Market (USM'2000), Lecture Notes in Computer Science (LNCS)*, vol. 1890, Springer Verlag, 2000.

[11] Kaijser P., Parker T., and Pinkas D., "SESAME: The Solution to Security for Open Distributed Systems," *Journal of Computer Communications*, pp. 501-518, vol. 17, no. 4, 1994.

[12] Li S., Ashri R., Buurmeijer M., Hol R., Flenner B., and Scheuring J., *Professional Jini*, Wrox Press Inc., 1st edition, 2000.

[13] McGraw G. and Felten E., *Securing Java, Getting Down to Business with Mobile Code*, John Wiley and Sons, 2nd Edition, 1999.

[14] MIT's Kerberos Homepage, http://web.mit.edu/kerberos/www/, July 2002.

[15] Mostéfaoui G., "Security in Pervasive Environments, What's Next?," *in Proceedings of the 2003 International Conference on Security and Management (SAM'03)*, Las Vegas, Nevada, USA, June 2003.

[16] Mostéfaoui G. and Brézillon P., "A Generic Framework for Context-Based Distributed Authorizations," *Fourth International and Interdisciplinary Conference on Modeling and Using Context (Context'03), in Lecture Notes in Computer Science*, Springer Verlag, 2003.

[17] Schoch T.,"An Authentication and Authorization Architecture for Jini Services," *Diploma Thesis*, ETHZ, October 2000.

[18] Schoch T., Krone O., and Federrath H., "Making Jini Secure," *in Proceedings of the Fourth International Conference on Electronic Commerce Research*, pp. 276-286, 2001.

[19] Schumacher M. and Roedig U., "Security Engineering with Patterns," *in the proceedings of the 8th Conference on Pattern Languages of Programs (PLoP'2001)*, 2001.

[20] Simple Public Key Infrastructure Working Group, *Simple Public Key Infrastructure*, http://www.ietf.org/html.charters/spki-charter.html, July 2002.

[21] Sun Microsystems Inc., "The Davis Project," *accepted*, http://developer.jini.org/exchange/projects/davis/index.html, July 2002.

[22] Sun Microsystems Inc., *Secure Computing with Java: Now and the Future*, White Paper, http://java.sun.com/marketing/collateral/security.html, July 2002.

[23] Sun Microsystems Inc., *The Java Cryptography Extension*, http://java.sun.com/products/jce/, July 2002.

[24] Sun Microsystems Inc., *Java Secure Socket Extension*, http://java.sun.com/products/jsse/, July 2002.

[25] Sun Microsystems Inc., *The Java Authentication and Authorization Service*, http://java.sun.com/products /jaas/, July 2002.

[26] Sun Microsystems Inc., "*Default Policy Implementation and Policy File Syntax*, http://java.sun.com/j2se/1.4/docs/guide/security/PolicyFiles.html, July 2002.

[27] Sun Microsystems Inc., "*Permissions in the JavaTM 2 SDK*, http://java.sun.com/j2se/1.4/docs/guide/security/permissions.html, July 2002.

[28] Sun Microsystems Inc., *Java Card (TM) Technology*, http://java.sun.com/products/javacard/, July 2002.

[29] Sun Microsystems Inc., *The Davis Project: Overture 0.05 Release*, http://developer.jini.org/exchange/projects/davis/overture.html, July 2002.

[30] Sun Microsystems Inc., *Jini (TM) Architecture Specification*, Version 1.2, http://wwws.sun.com/software/jini/specs/jini1.2html/jini-title.html, July 2002.

[31] The Java Community Process Program, *JSR 76 RMI Security for J2SETM Community Draft Ballot*, http://jcp.org/jsr/results/76-7-1.jsp, July 2002.

[32] Transport Layer Security Working Group, *Transport Layer Security*, http://www.ietf.org/html.charters/tls-charter.html, July 2002.

[33] Transport Layer Security Working Group, *SSL 3.0 Specification*, http://www.netscape.com/eng/ssl3, July 2002.

[34] Venners B., *Security and the Class Loader Architecture*, http://www.javaworld.com/javaworld/jw-09-1997/jw-09-hood.html, July 2002.

[35] Yellin F., "Low Level Security in Java", *in Proceedings of the 4th International World Wide Web Conference (WWW4'1995)*, Boston, pp. 369-380, 1995.

[36] Yoder J. and Barcalow J., "Architectural patterns for enabling application security," *in Proceedings of the 4th Pattern Languages of Programming*, Monticello, IL.

Ghita Mostéfaoui received her engineer's Diploma in electronics from the University of Blida in Algeria in 1996. She then received a fellowship from EPFL(Ecole Polytechnique Fédérale de Lausanne) Switzerland to attend a pre-doctoral school in computer science. Since 1999, she is a research and teaching assistant in the Software Engineering Group at the University of Fribourg and enrolled in both Fribourg and the University of Paris VI to prepare a PhD dissertation in computer science. Her main research interests include context-based security, context-aware computing and software frameworks for distributed systems.