Making Jini Secure*

Thomas Schoch
Computer Science Department
ETH Zurich
CH-8092 Zurich
schoch@inf.ethz.ch

Oliver Krone
Corporate Technology
Swisscom AG
CH-3050 Bern
Oliver.Krone@swisscom.com

Hannes Federrath
Department of Mathematics and Computer Science
Freie Universität Berlin
D-14195 Berlin
feder@inf.fu-berlin.de

Abstract

In this paper we describe a security architecture for Jini services. The current Jini API and its implementations do not handle any security aspects. In an open and distributed environment like the Internet, many attacks can be started or Jini services can be abused. On the other hand, Java – the programming language which Jini is made of – provides some security concepts. We show how a strong authentication and authorization architecture for Jini with a secure communication channel can be realized by using existing Java technology and without any modifications of the Jini API.

*The main part of the work on this research topic has been done during a research stay of the authors at the International Computer Science Institute (ICSI) in Berkeley, California.

1 Introduction

It is likely that we will see another paradigm shift in the field of computer science in the near future. Starting from a centralized mainframe oriented approach, the main computing paradigm has shifted to a Client/Server approach some 15 years ago. The next logical step will consist of billions of small, autonomous devices implementing a rich set of services that will be equipped with spontaneous networking capabilities and have access to any information and provide access to any service “on the net”. Business processes within a company or between a supplier and its customers can also be modeled as cooperating services on the net.

These services require a distributed communication infrastructure. A communication infrastructure acts as a middleware layer between the operating system and the application. Its task is to provide functions like service discovery and brokerage, very similar to a marketplace in a classic commerce scenario, where marketers and customers can meet and trade. Among other duties, middleware can leverage these concepts of classic commerce to e-commerce.

Jini [2, 12, 9], a technology from Sun Microsystems, is one of several approaches to provide a middleware for service discovery and brokerage. A Jini service that wants to provide its funcionality has to register itself with a so-called lookup service. For that, it hands over a service proxy to the lookup service. A service proxy is a regular Java object that handles the communication between a client application and the Jini service. On the other hand, a Jini client that needs a Jini service, first queries the lookup service for an appropriate service. If an appriopiate Jini service is found by the lookup service, the corresponding service proxy is returned. From that moment, all communication between the Jini client and the Jini service is handled by the service proxy without further participation of the lookup service. Another key concept of Jini is the self-healing ability that fits very good into spontaneous networks, where

appliances appear and disappear without notification, so that the self healing ability does implicitly the clean-up of no longer used resources.

However, Jini does not provide any security mechanisms that are crucial to e-commerce applications which rely on confidentiality and integrity. This paper presents a solution to the following three problems:

- Jini's lookup service lacks a sophisticated security model. As a consequence, each Jini client can get a service proxy and each Jini service can register and even re-register on the lookup service without restrictions.

  The re-registration opens the door for "man in the middle attacks". The "man-in-the-middle-service" has only to re-register its proxy with the same Service-ID as the original service, hence replacing the well-behaving service by a malicious one. A client is unaware of that change, because the man-in-the-middle-service provides the same API like the original service. For example, malicious competitors could replace your service proxy with their own one and all your clients will be redirected to them, so that your revenues will shrink.

- Jini lacks service authentication and authorization mechanisms: suppose a bank provides an e-banking service that offers you access to your account. Nothing prevents your colleague to call the creditTransfer-method on your account.

- Jini also has to cope with a general security problem, namely sniffing on the net. Every network package in the Internet can potentially be read, altered, or replayed. For example, a malicious competitor could get confidential information or even worser could manipulate information like prices.

To solve these problems, we present an authentication and authorization architecture that uses secure communication channels and digital signatures. Other security issues such as of hiding services or preventing denial of service attacks are not subject to this paper.

The rest of this paper is organized as follows: In Section 2 we give an overview of related work on Jini security. Section 3 gives an overview of security in distributed systems in general, followed by Section 4 where we look at security issues in Java that are used in Section 5 which also introduces our approach. Section 6 concludes this paper and provides an outlook on future work.

2 Related Work

We know of three other projects which also provide a secure Jini infrastructure. Pasi Eronen showed in his master thesis [3] how to incorporate Simple Public Key Infrastructure (SPKI) into a Jini security solution. Hasselmeyer, Kehr and Voß put their focus on securing the lookup service [5]. The third solution was a presentation at the Java One 2000 conference by Christopher Steel, Senior Java Architect at Sun Microsystems. All of them use signing of code as we propose. However, they do not provide transparency to the client and they do not provide a multifunctional RemoteCallbackHandler. In our architecture, a RemoteCallbackHandler is a remote Java object at client side that can invoke different authentication technologies like the use of the so-called Java Ring. There also exists a Jini security homepage [10], where Jini security relevant problems are discussed.

Besides Jini, some other similiar systems exist that provide service discovery, for example Universal Plug and Play, Ninja, eSpeak. These technologies are evaluated and compared in the project "Jini and Friends at Work: Towards secured service access", which has been carried out by the Eurescom organization [6].

3 Security in Distributed Computing

Security can be seen as the fact that protection goals are fulfilled in spite of *intelligent attackers*. Security goals refer to the typical threads in information systems: [14]

- Unauthorized acquisition of information. (*Security goal:* protection of confidentiality of content data, traffic data, anonymity and unobservability and protection

of locations of mobile users.)

- Unauthorized modification of information. (*Security goal:* protection of integrity of content data, accountability of sending and/or receiving data by means of digital signatures and certificates, and protection of billing data.)

- Unauthorized impairment of functionality; (*Security goal:* protection of availablity of contents and services within well-defined time constraints for all authorized entities.)

Jini services provide functionality in an open environment and interact spontaneously with possibly untrusted components or "visitors". Consequently, serious attacks by malicious clients have to be blocked.

In a Jini-controlled home environment for example, the owner of the house may be allowed to control the temperature in the house from anywhere (e.g. Internet Cafe) by calling the corresponding Jini service of the house. On the other hand it has to be prevented that an unauthorized person can switch on and off heaters, lights, or even the alarm system of the house. Therefore, an identification and authentication mechanism is required. Moreover, the children of the family may control the temperature of the animal's terrarium but not of the whole house. Therefore, an authorization and rights managements system is necessary. Both, authentication and authorization can be implemented for Jini services and are based on the standard Java security classes.

Confidentiality of content can be achieved by encryption. Techniques to protect the confidentiality of traffic data as well as locations, anonymity and unobservability are subsumed under so-called privacy-enhancing-technologies [4].

The major technology that realizes integrity and accountablity are digital signatures, where a person generates a key pair (a private and a corresponding public key), and publishes the public key. Now, everybody is able to verify whether the person has signed a document by applying the public key and the document to a verification algorithm. The authenticity of the public key, i.e. the association of the person and the individual public key, is given by a digital certificate issued by a certification authority (CA).

4 Java Security

A short overview of Java Security is given in this Section, since Jini is based on Java. For further information about Java Security, we refer to the Java Security Site [11].

4.1 Security in Older JDK Versions

With the first version of Java (JDK1.0), Sun introduced the sandbox model in which downloaded code has restricted access to any resource outside the sandbox. JDK1.1 extended the sandbox model with the signing of code. A class file and its digital signature are checked at client side. If the signer of the code is trusted and the signature could be verified, the Java applet gets the same permissions as a local class. If the signer of the code is not trusted or the signature is not valid, the applet will be executed in the sandbox only.

4.2 Security in JDK1.2

JDK1.2 introduced fine grained access control to all security relevant resources. It uses `Permission` classes and subtypes of it to represent the right to access a resource, e.g. `new FilePermission ("c:\config.sys", "read")` represents the permission to access the file `c:\config.sys` for reading only. These permissions can be granted on the basis of the codebase, by which the code is signed or without any restrictions. That means the sandbox model and the explicit distinction between local and downloaded code is no longer needed, because the functionality of the older models is a subset of the new one. A so-called key store is used to store the private signing keys and the certificated public verification keys for digital signatures. To specify where the key store is located and which permissions are granted, a policy file is used which can be an argument to the JVM or referenced by the default security settings.

4.3 Security in JDK1.3 Based on JAAS

Granting permissions in JDK1.2 is based on where the code is from and who signed the code. Java Authentication and Authorization Service (JAAS) is a standard extension to the JDK1.3. JAAS extends the JDK1.2 concept by granting permissions based on which behalf the code is executed. It also contains modules to log on the user. JAAS is based on the Pluggable Authentication Modules (PAM) framework which defines an API for applications to log on users. Thus, every application can use the PAM framework which can be configured by an administrator. Consequently, PAM is independent of which application uses the authentication mechanisms and configuring the authentication policy can be done separately.

4.3.1 Subjects and Principals

`Subject`s and `Principal`s are used to represent the identity of an user and her login names. The distinction between `Subjects` and `Principal`s is very important because a user can have different login names for different services. That means that a user has a collection of different login names.

PAM demands that the login procedure is independent of the authentication technology, i.e. which authentication technology is used. For that purpose, JAAS defines a `LoginContext` and a `LoginModule`.

A `LoginModule` represents one authentication technology, e.g. user name password authentication (figure 1). Every authentication technology must implement the two phase protocol – similar to the two phase transaction protocol used by Jini's Transaction Manager – which will be invoked by the `LoginContext`, the central manager of the login process.

JAAS provides a stackable authentication framework. That means multiple `LoginModule`s can be specified in a login policy file (figure 1).

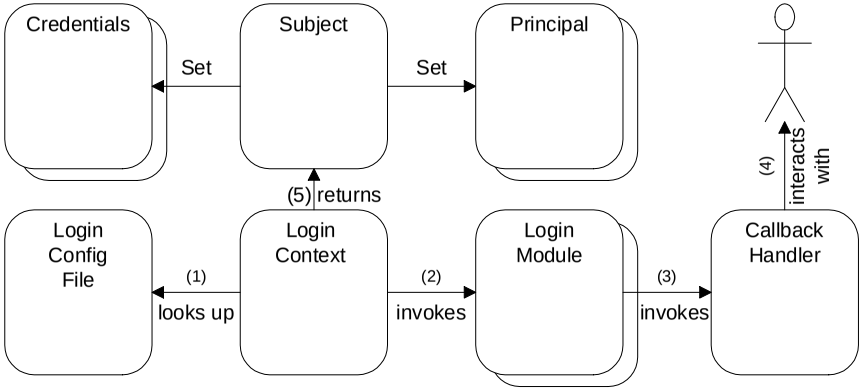

Figure 1: JAAS concepts and classes.

4.4 Signed Code in Java

A common problem of dynamically loaded code is the unawareness, whether the code does what it is expected to do, especially whether it does not perform any malicious actions. For that purpose Java uses digital signature of code. This ensures that an attacker cannot replace code in an unnoticed way.

We make two assumptions about signed code:

- the signer of the code keeps her private signing key secret;
- the signer guarantees that by using the signed code, the user does not suffer from any malicious actions, e.g. transmitting secret data without encryption over an insecure channel like the Internet.

These assumptions are also made by Java for downloaded code. This implies that

the user has to trust the signer and has to verify the validity or correctness of the certificate that the user utilizes to verify the digital signature of the code.

These assumptions do not overload the actual semantic of signed code. It is exactly the way Java intends signed code to be interpreted: the semantic only guarantees what the code is not doing, it says nothing about what the code is excatly doing. This fits perfectly in our model where the authentication procedure is transparent to the client. If authentication is used, and the user trusts the signer, the code is expected to do no malicious actions which could harm the user's interests.

It is up to both parties how the user gets the certificate from the owner. It is also up to the user if he trusts a self-signed certificate. In that case, the certificate is only used as a container for the public key. It is also possible to demand that the certificate has to be signed by a trusted CA.

4.5 Encryption in Java

We use the Java Cryptography Architecture (JCA) that provides several classes which make it possible to encrypt and decrypt Java objects. JCA provides cryptographic primitives independently of a certain implementation. It defines an Application Programming Interface of a cryptographic function (e.g. encryption, hash function, random generator) by means of an Engine class.

The actual implementation of the primitives are provided by a Cryptographic Service Provider (CSP). Thus, it is possible to work with an abstract encryption function on the application level. The programmer does not care about details such as the block size of a cipher, how key generation is realized, or even which algorithm is used.

5 An Architecure for Secured Service Access

As mentioned in Section 1, we present a solution to three security problems which Jini does not address. Firstly, Jini has no authentication and authorization mechanisms, so that untrusted parties can use any service on the net. Secondly, re-registration of untrusted service proxies is possible which opens the door for man-in-the-middle-attacks. And thirdly, the network packets that are required for the communication between Jini entities can be read, altered, replayed, or deleted.

This section provides a high level overview of our architecture that shows a solution to these problems. A first prototype is up and running, and includes smartcard authentication (Java Ring). The system is built on standard off the shelf Java technology and has the following characteristics:

- Uses JAAS 1.0, JDK 1.3, JINI 1.0.1, JCE 1.2.1, JavaCard 2.0,
- Transparent to the client: existing clients do not have to be modified,
- Introduces minimal overhead at server side,
- Provides powerful login policies that allow flexible authentication schemes,
- All security infrastructure components are built as Jini services,
- Provides authorization mechanisms (JAAS 1.0 + Java Security Architecture).

5.1 Service Authentication and Authorization

Every service has to authenticate and authorize each client request. The client has to make sure that it is running only signed code from a trusted source.

5.2 Login Policies

A user should interact as little as possible with the login infrastructure, therefore a single sign-on approach is used where appropriate. Especially in the home environment, users typically do not want to authenticate themselves, when they turn on the light.

The login policy should be as flexible as possible. Different services in different environments can specify different policies to log on the user. For example, a user at

home can turn on the lights without authentication, in her office she has to enter a user name and password to turn on the lights at home and on the Internet she may have to authenticate herself using a Java Ring.

5.3 Client Transparency

The security infrastructure should be transparent to the client implementation. Existing clients do not have to be changed in order to use the security infrastructure. The user does not have to specify the authentication method. This will be determined through the login policy at runtime by the system.

5.4 Architecture

Next, we describe how these goals are achieved. The architecture consists of different Jini services that implement the security infrastructure.

5.4.1 Client

In order to achieve client transparency, all security-related functionality is packed into the service proxy. The client uses the published interface of the service as if it were an insecure service. To ensure that the client only uses trusted service proxies, the security settings at client side have to be changed, so that the clients can only run code from trusted sources. These settings are transparent to Jini, since the evaluation of the security settings is done by the core Java classes in the background. This approach solves the problem of man-in-the-middle-attacks, since a re-registered service proxy could not be digitally signed with the orginal service's signing key and therefore the signature check at client side that is done by the core Java classes will realize that the re-registered service proxy is not digitally signed or that it is digitally signed by an unknown party.

5.4.2 Service Proxy and Service Backend

Upon service discovery, the client loads the appropriate service proxy from the Jini Lookup service that in turn communicates with the service backend. To provide privacy and to prevent replays, we use symmetric encryption of the data, a message authentication code and add a transaction number. The symmetric encryption is achieved using the Diffie Hellman key agreement scheme: the backend service generates its public and private keys, the private key remains at the backend service, the public key will be transferred to the service proxy. After a proxy has been downloaded by the client, the proxy generates its own private and public keys and with its own private key and the backend's public key, it generates the symmetric secret session key and sends its public key back to the backend, so that the backend can generate the symmetric secret key which will be stored in a session database.

Since the client runs only signed and trusted code, and the communication between service proxy and service backend is secure using the mechanism described above, the second of the three problems is solved, namely privacy is achieved and replaying and changing of data between the client and the server is prevented.

5.4.3 Interaction Scheme of the Infrastructure

In the following, it is described how authentication and authorization can be achieved using the above infrastructure, see figure 2. This is also the solution to the third and last problem. We put our main focus on that problem, since it is the most complex one.

- Upon service invocation, the service backend contacts a Jini service called `SubjectAuthenticatorService` which returns a `Subject` object introduced in JDK 1.3 / JAAS 1.0. It is up to the `SubjectAuthenticatorService` how to generate the `Subject`. In the reference implementation, we used the JAAS technology, because it

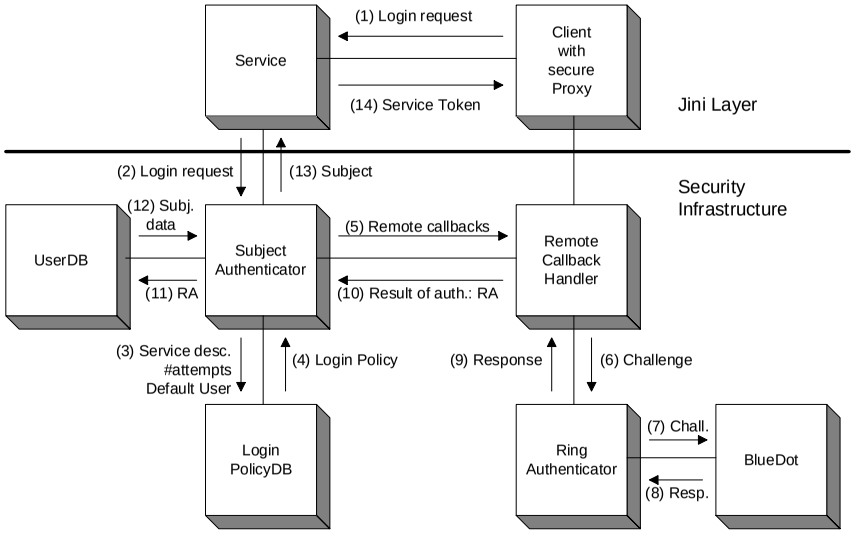

Figure 2: Components of the security architecture.

is already available and it is easily extendible to be used in a distributed environment.

- The `SubjectAuthenticatorService` gets the information to which service the user wants to log on including the number of attempts the user has performed already and from which device the user wants to log on. With this information the `SubjectAuthenticatorService` contacts the login policy database: `LoginPolicyDB`. This can be a separate JINI service or can be built in the ServiceAuthenticatorService.
- The `LoginPolicyDB` returns a `LoginPolicy` which specifies a policy of how to authenticate the user. Now the `SubjectAuthenticatorService` contacts a `RemoteCallbackHandler` which was instantiated by the service proxy at client side. Depending on the login policy, the `RemoteCallbackHandler` can prompt for a user name and password, or initiate even more complex authentication schemes.
- The `RemoteCallbackHandler` may contact a JINI `RingAuthenticationService` which in turn contacts a JINI `BlueDotService` to run a challenge response applet on a Java Ring, for example. A `BlueDotService` provides the functionality of a Java Ring reader that is also known as Blue Dot Receptor.
- After the `RemoteCallbackHandler` has received the result of the authentication procedure, the information is sent back to the `SubjectAuthenticatorService` which contacts a user database, `UserDB`, which again can be a JINI service or built in the `SubjectAuthenticatorService`.
- The `UserDB` checks if all the data is correct and returns the data needed to build a `Subject`. Based on this data the `SubjectAuthenticatorService` builds the `Subject` and returns it to the service.
- The service finally stores the `Subject` in its context information and returns a token to the service proxy, to be used by the client that is used transparently for every service invocation. This completes the authentication procedure.
- To perform actions that need authorization, the service runs the `Subject.doAs()` methods for the client. This method checks, if the appropriate permissions have been granted to the given Subject. If not, an exception will be thrown.

5.4.4 Anonymous and Single Sign-on

To faciliate the sign-on, another two concepts are used. At first glance, it may sound contradictory, but anonymous sign-on can be useful in many home environment scenarios. For example, a light switch inside a room should be usable by anyone who is in the room without authentication to turn the light on, but someone in an Internet Cafe must be authenticated if she wants to turn on the light. For this reason the light switch tries to log on as DefaultUser and a challenge response procedure verifies the identity of the light switch as DefaultUser.

Another simplification is the single sign-on approach. That means to log on one service and to use other services without further authentication. This works by giving out LoginTokens which are stored transparently at client side. To prevent log on services which require weak authentication and use their LoginTokens to log on services which require strong authentication, every LoginPolicy defines an `imply()` method, which defines what LoginPolicy are equal to or stronger than the current LoginPolicy.

The use of both concepts is optional. Only if they are explicitliy listed in a LoginPolicy configuration, they are used by the security infrastructure.

5.5 Comparision with the CORBA Security Service

CORBA is a middleware platform for remote invocation of objects. More information on CORBA are available at the OMG homepage [7]. CORBA uses Object Request Brokers (ORBs) on client and server side that handle the remote invocation transparently for the client and server object. Exactly the same security problems that Jini has to cope with its services, must be handled by CORBA on object level.

CORBA's security service [8] is transparent to the client and the service, since it is invoked by the ORB, that in turn performs the invocation transparently for the client and the service. In our architecture, we can achieve transparency for the client only, since one of our goals is not to change the underlaying infrastructure. But the overhead at server side is minimal. Like in CORBA, the security policies can be managed by an administrator independently of the application.

Authentication of users or system entities is done at client side and it is up to the actual implementation of the security service how to achieve this. For this reason the remote ORB must establish trust between itself and the client ORB that again needs extra administration. In our approach the authentication of users is driven by the service side. User interaction schemes are performed at client side, but the data is checked at service side. We think our approach is more useful, since a bank service for example does not want to trust all the client ORBs on all the home computers.

Authorization in CORBA is also based on an authenticated user. Its authorization model is more complex than the one used in our architecture, but it has the same expressive power. Access checks are performed by the infrastructure in both architectures.

CORBA's security service implementations may also provide delegation so that the server object can act as the calling client object. We also integrated the basics for delegation and a finer model is currently under development. In general, delegation can be of use, but an administrator has to do extra work to configure delegation rights and it is possible for delegates to misuse their delegated rights.

Two other concepts of CORBA are auditing and non-repudation. Both concepts refer to accountability and the latter is optional in CORBA. We have not focused on accountability, since this can be easly added to our architecture.

5.5.1 Summary of Used Java Security Features

To implement the presented architecture, we used some classes of the JCA. The following summary gives an overview of all keys and classes used in our implementation. The following keys are used:

- private signing keys, stored in the key store at service backend side, used to sign static code and dynamic data, created together with the certificate with the command line program `keytool` at service backend side;

- self signed certificate containing the public verifying key, stored in the key store at client side, used to verify signed code and dynamically signed data, created together with the command line program `keytool` at service backend side;
- private Diffie Hellman (DH) key, created dynamically together with the public DH key on both sides, stored locally, needed for creation of a secret key;
- public DH key, created dynamically together with the private DH key on both sides, the service backend signs it and puts it in the service proxy, the service proxy sends it unsigned to the service backend;
- secret key, created dynamically on both sides with the own private DH key and the public DH key from the other side, used to encrypt and decrypt data symmetrically and to compute a Message Authentication Code (MAC).

The following classes of the `java.security` and `javax.crypto packages` are used:

- `Cipher`, used to encrypt and decrypt data, needs the secret key for initialization; input and output are byte arrays,
- `SealedObject`, used to encrypt and decrypt Java objects, needs a serializable object and a `Cipher` object for encryption, needs the secret key for decryption,
- `KeyAgreement`, used to compute a secret key that is only known by two parties over an insecure channel, requires the own private DH key and the other public DH key,
- `KeyPairGenerator`, used to generate a pair of public and private keys, used here to generate the private and the public DH key,
- `Signature`, used to compute and to verify a digital signature, needs the private signing key or the public verifying key for initialization, input and output are byte arrays,
- `SignedObject`, used to sign and verify signatures on Java objects, needs a serializable object, the private signing key and a `Signature` object for signing, needs the public verification key and a `Signature` object for verification,
- `Keystore`, used to store and retrieve private keys and public certificates persistently, needs an `InputStream` and optional a key store password for initialization, needs an alias to retrieve public certificates and private keys, whereas latter needs a password obligatory to access, used here to store the private signing key and the public verifying key inside a certificate,
- `Mac`, used to guarantee the integrity of a message, needs the secret key for initialization, used here to guarantee the integrity of the encrypted data.

6 Conclusion

The security enhancements described in this paper provide the following advantages. The solution builds on top of the Jini API, so that modifications to the Jini API are not necessary. Only standard Java packages and no proprietary software is used. The overhead to implement a secure Jini service compared to the implementation of a normal insecure Jini service is minimal. That means rapid creation of new secure Jini services and an abstract view of the underlaying security infrastructure. At client side, the security infrastructure is transparent to any Jini client. An already existing Jini client does not need to be modified if the security of a service has to be improved.

The authors think that the transparency for the Jini client is an important design decision, since the Jini client could not weaken the security policy at service side. Another benefit is the faster creation of new Jini clients, since no security issues have to be handled by the programmer. But there also exists another view. The Java RMI

Security Specification provides – building on JAAS – authentication, integrity, confidentiality and delegation for Java Remote Method Invocation in general. Like Jini, it also uses a service proxy client concept. The main difference is that all security concepts can be negotiated between the service and the client. Accordingly, the client is aware of security and the programmer must deal with it. That means the development of client software is more complex.

Another important fact is the signing of code. In our architecture, the service signs the service proxy code and its data so that the JVM at client side can check whether the service is trustworthy. The RMI solution does not use this concept, since signing of code does not guarantee that the service really supports integrity or confidentiality, although signed applets which can be downloaded have the same semantic of signing. The semantic is that the integrity of the signed applet could be verified and the identity of the signer is known. So if the issuer is trustworthy, the signed code is trustworthy, too. The issuer guarantees that all Java code does not harm the users interests (e.g. does not delete files or send confidental data over an insecure channel). This cannot be technically achieved, but it could be a legal contract. Even Java applets in the Internet banking environment use this concept of signed code to establish a secure communication channel.

For these reasons, we think that our architecture is a suitable basis to secure Jini applications. The source code of our reference implementation can be downloaded at `http://www.inf.ethz.ch/~schoch/jaaa.zip`. One aspect of our future work is to put the cryptography work on socket level. Accordingly, the security architecture would also be transparent to the service, except the initilization of the service. Another aspect is to secure events and leases, which could be achieved in a similar way.

References

[1] Dallas Semiconductor. iButton Home Page. `http://www.ibutton.com/index.html`

[2] W. Keith Edwards. Core Jini. Prentice Hall, 1999.

[3] Pasi Eronen. Security in the Jini Networking Technology: A Decentralized Trust Management Approach. 2001-03-06. `http://www.cs.hut.fi/~peronen/publications/masters_thesis.pdf`

[4] Hannes Federrath (Ed). Designing Privacy Enhancing Technologies. Proc. Workshop on Design Issues in Anonymity and Unobservability, LNCS 2009, Springer-Verlag, 2001.

[5] Peer Hasselmeyer, Roger Kehr, Marco Voß. Trade-offs in a Secure Jini Service Architecture, 2000. `http://www.ito.tu-darmstadt.de/publs/papers/usm00.pdf`

[6] Eurescom. Jini & Friends @ Work. `http://www.eurescom.de/~public-webspace/P1000-series/P1005/`

[7] The Object Managment Group. `http://www.omg.org/`

[8] OMG Security. Security Service 1.7. `http://www.omg.org/technology/documents/formal/omg_security.htm#corba`

[9] The Community Resource for Jini Technolgy. `http://jini.org/`

[10] Jini Security Project hosted by Cees de Groot. `http://www.cdegroot.com/cgi-bin/jini?SecurityProject`

[11] Sun Microsystems. Java Security API. http://java.sun.com/security/

[12] Sun Microsystems. Jini Network Technology. http://www.sun.com/jini/

[13] Thomas M. Schoch. An Authentication and Authorization Architecture for Jini Services. October 2000. http://www.inf.ethz.ch/~schoch/da.ps

[14] Viktor L. Voydock, Stephen T. Kent. Security Mechanisms in High-Level Network Protocols. ACM Computing Surveys 15 (1983), No. 2, June 1983, 135-170.

Bio sketch of the Authors

Thomas Schoch studied computer science and business administration at the Darmstadt University of Technology, Germany from 1996 to 2000. He wrote his master thesis ”An Authentication and Authorization Architecture for Jini Services” during a research stay at the International Computer Science Institute in Berkeley, CA and received a Master degree in computer science. Currently, he is a PhD student at the ETH Zurich and works within the M-Lab project that investigates technical and business aspects of ubiquitous computing.

Oliver Krone studied computer science and electrical engineering at the Technical University of Munich and later received the doctoral degree from the University of Fribourg. After graduating from Munich, he worked as a Research Fellow at the IBM European Networking Center in Heidelberg, Germany, where he participated in the development of a multimedia communications system. He joined Swisscom Corporate Technology in 1998 and is currently Manager of the Open Communication Services Architecture Exploration Programme.

Hannes Federrath studied computer science and is working as a research assistant at Dresden University of Technology. He earned a Ph.D. degree in 1998. From 1999-2000 he was a guest researcher at the International Computer Science Institute Berkeley, and from 2000-2001 he was a guest professor at Freie Universität Berlin. His research interests include security and privacy in communication networks, mobile communication systems, cryptography, steganography and data security.