Decentralized Jini Security

Pasi Eronen and Pekka Nikander
Helsinki University of Technology
{pasi.eronen, pekka.nikander}@hut.fi

Abstract

Among the different approaches to distributed computing, the Jini technology provides a number of very promising methods for attacking the fundamental problems involved. Programs built according to the Jini principles will be able to function and survive in highly dynamic network environments, allowing applications to adapt their behaviour to the requirements of the current context. Unfortunately, the security problems that are bound to be present in any large scale deployment of Jini are not adequately addressed by either the current revisions of Jini technology or the underlying Java security solutions. In particular, the solutions proposed so far are either bound to a specific communication protocol—thereby hampering the protocol independence of Jini—or rely on centralized security servers, thereby losing the benefits of the ad hoc nature of Jini.

In this paper, we present results of our research that act as stepping stones towards a fully decentralized, complete Jini security architecture. In particular, we describe our experimental implementation that separates the Java 2 access permissions of Jini clients, service proxies, and services, while allowing natural delegation of Java 2 permissions between Jini enabled devices. Our solution integrates seamlessly to the underlying Java 2 security, and allows all of Jini's benefits to be utilized in a secure way.

1 Introduction

Distributed computing is fundamentally different from centralized computing. The usually mentioned four major differences include latency, memory access, partial failures, and concurrency (e.g. [35]). Security should definitely be added to this list, since a distributed system requires cryptography to be used while a centralized system may survive without it. While many of the approaches to distributed computing attempt to mask out some of these problems, the Jini approach mostly does not. Instead, it aims at providing tools and methods for effectively building software that adequately addresses these differences and is able to survive in the face of problems caused by distribution. [1]

The fundamental differences become especially apparent when considering the future ad hoc networks and other loosely coupled systems. These systems, by nature, are not only distributed but also *decentralized*. That is, a genuine ad hoc network does not have any centralized services but all the network services are configured and created on the fly. In the security area, the underlying mechanisms stay more or less the same to what is used in centralized systems, but the trust and infrastructure assumptions change altogether, requiring different kinds of solutions (cf. [29]).

In this paper, we present a fully decentralized network security architecture for Jini, and describe a prototype implementation of the architecture. The architecture builds upon our earlier work on trust management [25] and distributed Jini security [12]. The implementation is integrated to the Java 2 security model, augmenting and utilizing the security services provided by the Java 2 security architecture [15] and the Java Socket Security Extension (JSSE) [33]. In the future, we plan to look at how to integrate our extensions to the Java Authentication and Authorization Service (JAAS) [21] and to the RMI Security Extension [32]. Our solution is fully compatible with the Jini architecture and assumptions, and does not require any centralized security services. It makes a clean distinction between the access rights of client applications and service proxies, and provides a means of delegating Java 2 security permissions between Jini clients, services proxies, and services. These and other aspects of our solution are discussed in detail later in this paper.

The rest of this paper is organized as follows. First, in the rest of this section, we briefly describe the central concepts of trust management and the Jini architecture. After that, in Section 2, we provide a brief taxonomy of Jini related security requirements and a number of related design aspects. Section 3 discusses our design choices and outlines the architecture of our solution, and Section 4 describes the implementation, including some performance measurements. Related work is briefly discussed in Section 5, and Section 6 evaluates our approach in the light of the alternative solutions. We also give some ideas for future work. Finally, Section 7 contains our conclusions from this research.

1.1 Decentralized trust management

Traditionally, security has been based on identity authentication and locally stored access control lists (ACLs). This has been the case even in distributed systems. However, that approach has a number of drawbacks, including, for example, the problem of protecting the operations that are needed for managing access control lists remotely. In [5] Blaze et al. argue that "the use of identity-based public-key systems in conjuction with ACLs are inadequate solutions to distributed (and programmable) system-security problems."

An alternative solution, termed *trust management*, uses a set of unified mechanisms for specifying both security policies and security credentials. Basically, trust management usually involves signed statements (certificates) about what principals (users) are allowed to do, instead of traditional name certificates which just bind a public key to a name. Examples of trust management systems include the PolicyMaker, which originally introduced the term trust management [6], its continuations KeyNote and KeyNote2 [4], and in some respects, SPKI [11] and its applications, including TeSSA [23].

1.2 Introduction to Jini

The Jini programming model provides a set basic building blocks for distributed applications: distributed events, transactions, leases, and downloadable proxies. These don't try to hide the fact that networks are unreliable, and the approach, in general, encourages building more fault-tolerant applications [26, 35].

The building blocks are used in the centerpiece of Jini, the lookup service, which is a directory where service providers register themselves and clients search for what they need. For example, when a service registers itself with a lookup service, it receives a *lease* on the registration, with an expiration date. If the service doesn't renew the lease before it expires—for example, the service is disconnected from the network— the registration is automatically cleaned from the lookup service.

The lookup service is somewhat similar to other service location protocols, such as Salutation [28], Service Location Protocol [16], and Universal Plug and Play [34], except that matching is based in Java interface types. The central difference between Jini and the other service location protocols is protocol independence: that is, Jini does not mandate any specific communication protocol between the clients and the services (except for bootstrapping the system), but relies on dynamic Java class loading instead. Since the proxies are written in Java, the system also claims operating system independence; this in contrast with the other service location protocols which usually use non-portable device drivers.

All communication goes through proxies, which are local objects that implement some well-known interface (such as "Printer"). Proxies can be simple Remote Method Invocation (RMI) stubs which marshall method calls over the network, or they can implement part of the functionality in the proxy itself (for example, converting the data to the correct format for this printer). Also, some services don't necessarily require network communication at all, in which case the proxy alone implements the service.

Protocol independence and the ability to implement part of the intelligence on the client side give Jini tremendous flexibility. For example, proxies can communicate with devices which don't have a Java virtual machine; either legacy devices with proprietary protocols, or resource-stripped embedded devices. On the other hand, Jini requires that the clients have their own Java virtual machines.

1.3 Proxies and security

Protocol independence presents also some new security challenges. The Jini architecture doesn't include any security in addition to the normal Java security facilities (for protecting the client JVM from malicious proxy code), and the security aspects of RMI in their current state are insufficient for the task (see Section 4.7).

Since all communication goes through downloaded proxy objects, security methods used in environments with fixed protocols can't usually be used without some adaptation. For example, the Transport Layer Security (TLS) protocol supports authentication of both the client and the server using X.509 certificates [10]. This doesn't, however, help us in determining whether a specific proxy is trustworthy. The client certainly doesn't want to give its private key to the proxy (since it might use it to access a completely different service).

We feel the situation resembles the concept of delegation, and therefore a trust management system which supports delegation could be applied to the problem elegantly.

2 Requirements for Jini security

When talking about Jini security, we must first decide what security functionality is needed. This naturally depends on what we are using Jini for, and what trust relationships are involved. In this paper, we are focusing on the client accessing a server through a proxy, and leave the security aspects of distributed events, leases, and transactions for future work.

So far, we have identified the following requirements.

- *Principal authentication.* The client should be able to verify that it is actually talking to the right service and through the right proxy. Likewise, the service should be able to verify who is trying to access it.

It is important to notice that authentication is impossible in a number of situtations. For example, in a pure ad hoc network there may not be any prior information about the communicating peers.

- *Secure principal attributes.* In many circumstances, human readable and recognizable names are required for authentication. Services might also have other attributes such as security level (for example, a printer for printing classified documents) or the "owner" of the service (for example, "Alice's calendar"). Users might have other attributes such as memberships in groups or roles.

  Not all clients or services necessarily have names with any uniqueness beyond one client or server. For example, getting a CA-signed certificate for the doorbell ourside your door so that it can contact a server inside your house to play a tune doesn't seem very sensible.

- *Service access control.* Based on the result of principal authentication and/or capabilities presented by the client and/or other circumstances, the service should allow some operations and deny others.

- *Protection from applications.* The client Java Virtual Machine (JVM) might run multiple applications, some of which are not fully trusted, such as applets and games. Untrusted applications should not be able to access services with the user's privileges.

- *Protection from proxies.* The downloaded proxy code needs some special permissions (for example, to make network connections) when running inside the client JVM. Some proxies may need more permissions than others. These should be controlled somehow. Java, of course, provides some facilities for this, but they are somewhat insufficient for many applications.

On a lower level, protocol-related aspects such as message confidentiality and integrity, replay prevention, perfect forward secrecy of keying, and so forth, are also desirable. We do not consider these further, since appropriate solutions are widely known.

The actual requirements, of course, vary from case to case. For example, if the client runs only trusted applications, protection from applications might not be needed. Some of these, such as protection from proxies, could also be addressed separately from the rest.

2.1 Other design aspects

The requirements outlined above still leave a lot of freedom for the implementor. The design choices made will certainly affect the situations where the solution is applicable. In this section we identify some of the design aspects. In the next section we continue to set forth our choices, and the reasons behind them.

- *Centralization.* Does the architecture rely on some centralized servers or authorities? Are they required to be on-line during service access?

  Centralized security architecture probably makes administration in large networks easier. On the other hand, it doesn't work well for, e.g. mobile ad hoc networks. Furthermore, there are several somewhat independent features which could be centralized or decentralized. For example, we could have decentralized access control with either centralized naming (CA type) or decentralized naming (for example, PGP-style "web of trust").

- *Trusted components.* Does the system rely on the security of the lookup service, or some other on-line security server?

- *Protocol independence.* Is the solution tied to some transport protocol, such as the RMI wire protocol over TLS or IIOP? If the protocol is fixed, it can be implemented using trusted code, which simplifies the security situtation.

- *Service access control model.* How flexible and fine-grained is the access control mechanism? What kind of policies can it support? For example, applications which access medical data probably require more complicated policies than an office environment.

  This is influenced by other choices. For example, if the access control is managed by a "container" of some kind, the granularity is probably at most per-method.

- *Application protection model.* How flexibly can the user decide which client applications are allowed to do what? For example, if the system uses TLS client authentication and a server-side access control list (ACL), the restrictions can't probably be more specific than per service (i.e., application can use key X only to access service Y).

- *Delegation.* Does the system support delegation? Can the delegated rights be restricted somehow? How flexible are these restrictions?

- *Transparency.* How transparent the security system is for service or client software?

  For example, in Enterprise JavaBeans security is managed by the "container", so it is sort of transparent to the service software. It is probably a good idea to make the security as transparent as possible to client applications.

3 Solution architecture

Our goal was to provide access control for Jini clients, services, and proxies without sacrificing any essential Jini features, such as protocol independence. In our architecture trust assumptions are made visible by using authorization certificates instead of traditional name certificates. This allows us to identify what level of trust is really required by the application, which is important in, e.g., ad hoc environments where a fully trusted third party can't be assumed to exist.

3.1 Our design choices

When designing the system, our target environment was ad hoc mobile networks. For example, such a network might consists of PDAs communicating with a short range radio network such as Bluetooth [24]. Such an environment requires that the solution is decentralized, or at least does not rely on any on-line third party, since such a party might not be available at all times in an ad hoc network. Therefore, we also assume that the lookup service isn't secure. We also early rejected a centralized off-line trusted third party for signing the code as "trustworthy", since we believe that that kind of solutions are better for assigning blame afterwards than preventing wrong things from happening in the first place. Considering the communication, reliance on a fixed communications protocol was also deemed an unsatisfactory solution from the start.

Since we had prior experience in using SPKI certificates, we decided to use them for decentralizing trust. In our system, clients and services are identified by public keys. Unlike in some other architectures, names bound to these keys are not important. That is, when performing the access control decision, names are not used—names and other security attributes may be, of course, used by the application.

3.2 Using authorization

We next describe a typical authorization scenario in our solution. The service key typically delegates full permissions for the service to the administrator's key, who can then authorize ordinary users to use the service. The authorization is expressed as a SPKI certificate, where the administrator delegates the access right to user's public key. The certificates are stored by client.

The user can delegate a subset of her authorizations to local applications. The authorizations delegated to the application depend on how much the user trusts the application. For example, the user might trust a word processor to print correctly, and she would delegate the corresponding permission it. However, the user probably won't give the word processor the permission to access personal calendar files, because the application does not really need it, and it just might contain code that misbehaves.

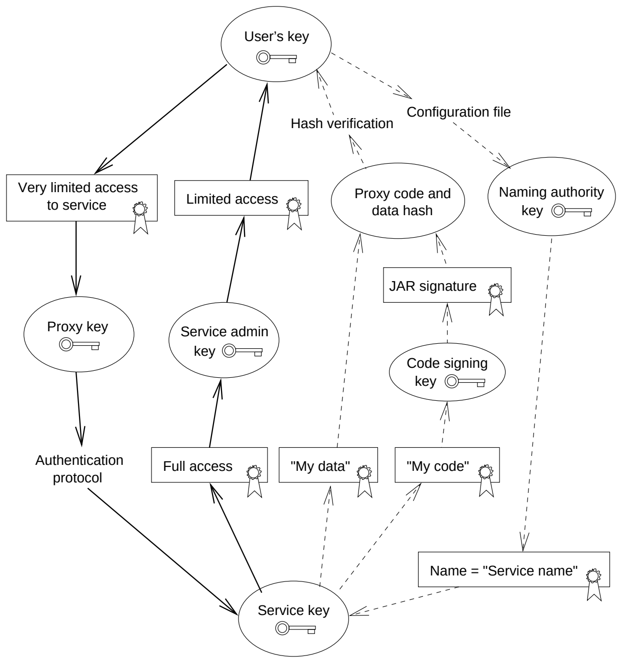

Figure 1: Typical certificate chains; the authorization chain (verified by the service) is shown in bold, the authentication chain (verified by the user) in dashed line.

Our solution provides a way for the application to use these authorizations with a service in Jini environment. One of the problems to be solved is how to prove these authorizations through the Jini proxy which is loaded from the network and can not be fully trusted by the user. The user's secret key is required to prove the user's authorizations but it must not be given to the proxy.

Typical certificate chains are shown in Figure 1. The details of proxy verifications are explained below.

4 Implementation

Our prototype implementation is responsible for proving user authorizations to services, authenticating proxies, and verifying authorizations. It is implemented completely in Java, and consists of about 10 000 lines of code. The implementation consists of the following components:

- SPKI certificate library (siesta.security.spki) is used for encoding and decoding SPKI certificates.

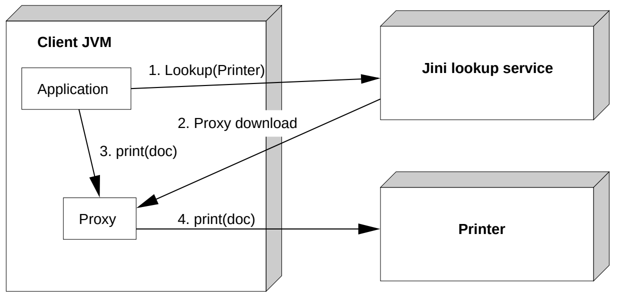

Figure 2: Accessing a Jini service, without any security features.

- Server-side module (siesta.security.authorization) verifies SPKI certificate chains. It could be considered a sort of "trust management engine", like KeyNote, but it is somewhat simpler. This package and its connection with the standard Java 2 security architecture are described in Section 4.6.
- Certificate repository (siesta.security.repository) provides a simple local certificate repository where authorization certificates are stored, and a certificate gatherer which tries to find a complete certificate chain. This could be extended to support of retrieval of certificates from the network using DNS [18], LDAP, or some other directory access protocol.
- Client-side security module is responsible for controlling access to user's private keys, authenticating proxies, and enforcing application access control.
- Utilities to simplify writing services and clients, such as signing proxies, verifying name certificates, etc.
- RMI over TLS [10] module supports using client authentication over TLS sockets, as described in Section 4.7.

4.1 An example scenario

The default behavior of a Jini client application and a service is shown in Figure 2, where an application prints a document.

1. An application, wishing to use a Jini service, contacts the lookup service, and performs an appropriate lookup (for example, searching for printer services). A list of available services is returned to the application.
2. The user (or the application itself) selects one of the listed services. A serialized proxy object is transported to the client, and the corresponding bytecode is downloaded.

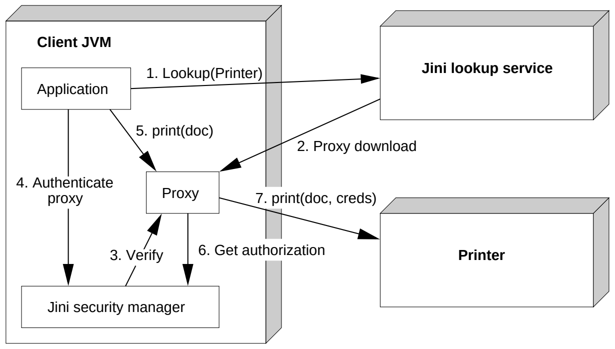

Figure 3: Accessing a Jini service with our security modifications.

3. The application calls some method on the proxy object, requesting it to do whatever the service does. In our example, it asks the proxy the print a document.
4. The proxy sends the request to the service, which prints the document.

In the next section we describe the modifications needed in our security solution.

4.2 Overview of the modified protocol

When security is applied to typical Jini scenario, a number of additional steps are needed. On the client side, we have a "Jini security manager" which is responsible for controlling access to the user's private keys. It also enforces application access control. The mechanisms needed at server side depend on the requirements of the service. The typical steps taken when accessing a service are described below, and are illustrated in Figure 3.

1. An application, wishing to use a Jini service, contacts the lookup service, and performs an appropriate lookup (for example, searching for printer services). A list of is returned to the application. No special security features are assumed here.
2. The user selects one of the listed services. A serialized proxy object is transported to the client, and the corresponding bytecode is downloaded (again, using standard Jini facilities).
3. The Jini security module asks the proxy for the service's public key, and checks that this proxy indeed represents that service. This is done by checking the signature of the code and data as described in Section 4.3.

4. Next we have an optional authentication step. If the application knows an identity of the desired service, it can now ask the Jini security service to authenticate the service key. Authenticating the actual identity (e.g. a human readable name) might involve certificates obtained from the lookup service or the proxy itself, and is described below in Section 4.4.

5. The application calls some method on the proxy object, requesting it to do whatever the service does. In our example, it asks the proxy the print a document.

6. The proxy then asks the Jini security library for authorization. The Jini security manager checks that (1) the proxy is trying to really access the service it represents and (2) that the application is allowed to perform this operation on behalf of the user. Application access control is described in Section 4.5.

   To enforce these local restrictions, we generate a temporary key for the proxy, and delegate the restricted rights to this temporary key[1]. A handle to this key is then given to the proxy. The proxy can't get the actual key material through this handle, but it can use it for signing data (this allows us to "revoke" the key immediately, if necessary).

   The certificate repositories are then searched for other certificates which might be relevant to the case, and the certificates are returned to the proxy.

7. Using the key handle, the proxy can open a secure connection to the server. The proxy can implement any protocol it chooses. Our library provides a module which uses RMI over TLS, as described in Section 4.7.

   After proving possession of the temporary key, the proxy sends the certificates and the service request to the server. The server checks the certificate chain, and then performs the operation. This aspect is discussed more in Section 4.6.

4.3 Proxy verification

Usually we wish to verify that the proxy really came from the service we want. Since services are identified by public keys, this can be arranged by having both the code and data signed by the service key. However, it should be noted that due to the ad hoc nature of the network, we do not necessarily know anything about the "authenticity" of the service key, at least not yet.

During the implementation we discovered what we consider a small deficiency in Java's facilities for signed code. It is not possible to give expiration dates for code signatures. The associated X.509 certificate has an expiration date, but it is not possible to produce signatures which have a shorter lifetime than the certificate. We wanted that possiblity (to make sure we are using the right version of the proxy code), so we had to make some modifications.

There are basically two ways of achieving the expiration. The JAR file signature could be modified to contain an expiration date. This would, however, require modifications to the JAR file loading code. This is by no means impossible; it has been done in the TeSSA project to allow delegation of code permissions with SPKI certificates [27].

Our approach splits the signature to two parts. We sign the JAR file (using standard Java facilities) using a newly generated key, whose private half is then destroyed. The service then supplies a SPKI certificate chain from the service key to this code signing key (usually just one certificate). This certificate chain is stored in the data part of the proxy. This approach has couple of advantages:

- We don't have to re-sign the JAR file if it hasn't changed. Since the JAR files are stored on a web server, the service might not be able to easily modify them.

- We don't have to modify the JAR file loading code.

- We can use existing JDK tools for signing the JAR.

The main drawback is that the signature expiry date isn't visible to the standard Java components.

In addition to verifying the authenticity of the proxy bytecode, we would like verify the proxy object as well. The straight-forward way would be to calculate the message digest of the serialized proxy object. However, this fails because we don't really know what part of the data is fixed state worth signing and which is just transient state. Also, the proxy might be composed of multiple objects.

We solved this by asking the proxy object to calculate its own message digest. The proxy bytecode has been verified in this point, so the proxy isn't completely untrusted, and it isn't in the service's interest to return a wrong message digest. On the other hand, a lazy service writer could defeat this check by always returning the same message digest (for example, zero).

The service then supplies a SPKI certificate chain from the service key to the message digest object-hash, and stores it in the proxy's data part. Usually this certificate chain is just one certificate.

1. We certainly wish to write a certificate that can only be used at the service the proxy represents. However, we noticed that there wasn't any elegant way to restrict the delegated certificate only to particular service in SPKI. Although we considered adding a new element to the SPKI certificate (say, named `(valid-at (public-key` *`service-key`*`))`), we decided to encode this information at a fixed position in the `tag` field.

4.4 Service/proxy authentication

After we have verified the signatures of proxy code and state, we know which service the proxy represents (as identified by the service's public key). Now the application may wish to verify other security attributes of the service, such as human-readable names.

We argue that this is a function best left to the application, since the trust models are very application specific. In some cases, a traditional solution based on a trusted third party is the most appropriate (e.g., naming printers on a corporate network). In many cases, a PGP-style web of trust may be more appropriate.

Stajano and Anderson [29] describe an example of an ad hoc networking situation where a completely different solutions are required. Consider a thermometer, having a very small display and communicating using a short-range radio. If we have a bowl of disinfectant containing many unused thermometers, it doesn't really matter which we choose; but we want to make sure we communicate with the one we have picked from the bowl. The thermometers could, of course, be given artificial names (such as serial numbers, which could be engraved on the case), but this solution isn't very user friendly. Instead, if we have a secure (free of active middle-men) communications channel, such as short-range infrared or physical contact, we can simply transmit the public key over this channel.

To help application developers in verifying human readable properties, our utility library provides support for two common cases: names signed by some central authority, and ownership of services (such as "John's siesta.pim.Calendar" service). The properties are represented by subclasses of java.security.Principal.

4.5 Application access control

When designing our system, the environment we had in mind was a PDA using a single keypair stored on a smartcard. This key would be used for accessing dozens of different services (using many different client applications). Therefore, we wanted to restrict what applications could do with the key. We call this feature "application access control".

Since we use SPKI certificate chains for authorizations, we can implement more complex restrictions than simply allowing or denying access to the key. Our implementation stores these restrictions as a subclass of Permission. The restrictions are associated with applications using the standard Java 2 policy mechanisms [15]. The only case requiring special treatment is the proxy class. Since the key used for signing the code isn't the same as the service key, this must be associated with the permission later (when the proxy state and code have been verified, as described in Section 4.3).

When the proxy requests some permissions to be delegated to its temporary key, the Jini security manager constructs the corresponding RemotePermission instance. It then uses the Java 2 stack inspection features (AccessController) [37] to verify that the application is authorized for this action.

4.6 Authorization checking

When the server has received a request from the proxy and verified the proxy's key, it gives the key and the certificates to a SPKI certificate chain verifier module. The verifier then verifies certificate signatures and validity and finds all certificate chains from the service key to the proxy key. These chains are then stored inside a PermissionCollection instance (PermissionCollections are used in the Java 2 security architecture to store a set of related permissions).

The service software can then call the implies() method of the collection, giving a parameter corresponding to the client request (the method then returns either true or false). Storing the authorizations inside a PermissionCollection gives the service software another possibility. It can use the AccessController.doPrivileged call to associate the permissions with the Java call stack. Permissions are then checked using normal System.getSecurityManager().checkPermission() call. In many cases, this is a cleaner solution than passing a PermissionCollection object through a long chain of method calls, or storing it in a visible variable. It also allows communicating these permissions to code which doesn't know the original call was a remote call.

4.7 RMI over TLS

Our architecture allows the proxy to implement any protocol for communicating with the service. In our tests, we have used RMI. The default RMI configuration uses normal TCP sockets, but it is possible to override this behaviour by supplying a pair of *socket factories* to be used on the server and client side of the communication. This is meant for plugging in Transport Layer Security (TLS) sockets.

Using these facilities, we implemented socket factories for TLS client authentication using the Java Security Socket Extension (JSSE) libraries [33]. During the implementation we found some slight deficiencies in the current RMI implementation. Hopefully, most of these will be fixed in the next release of RMI, and in the RMI security extension [32].

4.7.1 Problems with client authentication

Although the socket factories were originally intended for plugging in TLS sockets, the design supports cleanly only server authentication. The socket factories are given to the constructor of java.rmi.server.UnicastRemoteObject which is the base class of RMI server objects. The application has no further control of the remote method invocation process. The network connections are formed automatically whenever the client invokes a remote method and a server method is automatically executed with the arguments sent over the network. Neither the client nor the server has direct access to the underlying socket.

On the client side, it is difficult to actually verify that the stub is using the secure socket factory. Even more difficult is communicating the correct key to the socket factory, since the socket might be opened even before any methods are called (due to distributed garbage collection).

Similar problems appear also on the server side. Once a call is received, there it no way to get access to the socket instance it came from. In the case of TLS sockets, the socket would contain methods to get the client's key.

We worked around these problems (sort of) by communicating the keys using thread-local variables, and controlling the deserialization of the stub by wrapping it inside a MarshalledObject. We later found out that Balfanz et al. had independently discovered a similar workaround [2].

4.7.2 Code bases

We also encountered a limitation in the way RMI serializes stubs. When sending a serialized object to a remote system, a codebase URL is included with it. The URL specifies the location where the bytecode can be downloaded. The current RMI implementation gets this codebase URL from a global system configuration property named "java.rmi.server.codebase". This makes running multiple services inside the same JVM more difficult.

Fortunately, if the proxy was originally loaded with a subclass of java.net.URLClassLoader, its getURLs method is called to get the codebase URL. If we load the proxy from the URL on the server side as well, and instantiate and initialize it using the reflection API, the codebase gets set to the correct value. We must, of course, verify the signature on the bytecode on the server side to make sure we got the right proxy.

4.8 Performance

Table 1 shows our initial performance figures. Basically, the measurement represents the time required to delegate a permission from the client to the server through the proxy. As the measurements show, currently the authorization requires quite a lot of time. Most of the time is spent in Java cryptographic primitives. However, our current implementation is quite unoptimized. In particular, the process requires that a separate public key pair is created on the fly; these keys can be generated beforehand, and taken from a pool of pre-generated keys during the protocol run. As the table shows, this cuts the time required to a more reasonable value.

Measurement	average (ms)	std dev
Standard Jini/RMI call	30	2
With SPKI and TLS applied	6180	80
With pre-generated keys	983	130

Table 1: The results of performance measurements, measuring the time required for the first remote method call through an already authenticated proxy. The second and subsequent calls take about 300 ms in the secure case.

We used Sun's JDK 1.2.2 under Red Hat Linux 6.2 to do our measurements. Both the client and the server were run on the same machine, which was equipped with a 750 MHz AMD Athlon CPU and 256 MB of RAM. The measurements were run ten times, and the average and standard deviation were calculated.

5 Related work

5.1 Java security

Most work in Java security has focused on protecting the host from malicious code. The original JDK 1.0 featured a *sandbox* which limited the operations untrusted code could invoke. Since then, the Java 2 security architecture [15] added more flexible and fine-grained access control. A number of other solutions have been proposed [36, 38, 19]. The Java 2 security architecture has been extended with decentralized trust management in [27]. The concept of "who is running the code" has been implemented in the Java Authentication and Authorization Services (JAAS) [21], and has been extended with roles in [14]. Controlling the amount of resources (computational cycles, memory, etc.) a program can use is discussed in [8].

5.2 Distributed object security, mobile agents

Most work related to remote object security has focused on CORBA (for example, [3, 22, 31]). Although the concepts in CORBA security are similar than in Jini, the problem of untrusted proxy code requires different solutions. This issue is discussed in Section 5.3 below.

The mobile agent research community has also produced a lot of results related to mobile code and security. Most of that work has focused on protecting sites from malicious agents, and also on protecting agents from malicious hosts. Often the situation is the reverse of that in Jini: a user sends an agent to the service site, where it performs some functions on behalf of the user. In Jini, the code moves from the service to the client.

5.3 Solutions for downloaded proxy code

As explained in Section 1.3, communicating securely through downloaded proxy code presents new security challenges. There are a couple of solutions for this problem, and the following are reported in the literature.

- *Fixed protocol.* If the communication protocol is fixed, proxies can be generated on the client side, either statically (using an IDL compiler), or dynamically on-the-fly (cf. RMI security extension draft [32]). Statically generated proxies are used by, e.g., Balfanz et al. [2].
- *Centrally signed proxies.* The proxy code is signed by some central authority, and if the signature is valid, the code is considered fully trusted. This is used by Hasselmeyer et al. [17] and in the Sun demonstration solution presented at JavaOne 2000 [30].
- *Mixed approach.* In the RMI security extension draft [32], it is also possible to combine dynamically generated proxies (implementing a fixed protocol) with signed hand-written proxies.

  The hand-written proxy code and data are signed by someone, usually the service. The "trust verifier" object is obtained from the service using the dynamically generated RMI stub.

All of these approaches have their own benefits and drawbacks. Fixing the protocol eliminates the need to download the proxies and allow communicating with any service, but also lose the ability to implement part of the proxy functionality on the client side. On the other hand, requiring that the proxies are signed by some central authority restricts spontaneous networking.

The mixed approach seems most promising of these three, and resembles our approach in that the data and code are effectively signed by the service (though the details are quite different). However, it loses some of the protocol independence.

5.4 Jini-specific security

The security of specifically Jini systems hasn't been studied much yet. Sun presented a demonstration solution which integrates Jini with JAAS at JavaOne 2000 [30]. It is based on a centralized security server, and a certificate authority (CA) signing all proxy code. It is somewhat similar to Geoffrey Clements's Usersecurity project [7].

Hasselmeyer et al. have developed a Jini security solution based on a centralized secure lookup server [17]. Similar secure service directory in non-Jini environment is described by Czerwinski et al. in [9].

Sun's future solution for Jini security is the RMI security extension, currently in draft stage [32], which provides at least some support for intelligent proxies. However, the initial implementation of it only supports TLS based authentication, which, in turn, relies on a centralized certificate architecture, and separates the mechanisms for authentication and access control. Furthermore, a trusted component (not downloaded from the network) is responsible for opening the network connections and implementing some authentication protocol, so this might limit the protocol independence offered by Jini.

There is also a "Jini and Friends at Work" project going on at Eurescom, but no results have been published so far [13].

6 Evaluation and future work

In this project, our goal was to provide a Jini security solution that does not unnecessarily restrict the possibilities for creating secure Jini services. In particular, we wanted our solution to be protocol independent, to rely on existing Java security mechanisms to the greatest extent possible, to be reasonably transparent to the clients, service proxies, and services, and not to require any centralized servers.

We feel that we have mostly reached these goals. The implementation allows Jini services to implement any protocol between the proxy and the service. The solution utilizes the possibilities created by the underlying Java 2 security architecture and, if used, the existing Java Secure Socket Extension (JSSE). Furthermore, there is no need for any centralized security or other servers. On the other hand, in order to support permission delegation, the proxy and the server must take a number of additional steps, and they must transport SPKI certificates in the whatever protocol they use. Thus, our solution is not fully transparent to the proxy or the service implementations. But, given the requirement of protocol independence, our solution seems to be reasonably easy to utilize, and it is in full conformance with the central toolbox approach of Jini.

Compared to the other proposed solutions, our solution is similar to the other Jini security in the sense that it mainly addresses the proxy security problem, focusing on the authentication of the clients and the services, and on authorization at the method call level. On the other hand, our solution has a number of additional benefits. The main benefits can be enumerated as follows.

- The solution does not place any unnecessary restrictions on the implementation of Jini services. In particular, the solution is protocol independent, basically allowing any protocol to be used between a Jini proxy and the corresponding server.

- The solution does not require any centralized security services, and therefore it can be easily utilized in ad hoc networks.

- The solution allows fine grained Java 2 permissions to be separately applied in the client, proxy, and server. In particular, a client application that requests a Jini service runs in a separate protection domain than the proxy, thereby restricting the permissions the proxy has access to. Furthermore, permissions are explicitly delegated from the application to the proxy. Thus, in addition to restricting the permission of the proxy itself, this allows the proxy to provide the service with a proof that the application does have the permissions required by the service and that the application really wants to use the permissions to access the service.

  Since the permissions can be delegated to the service (if desired), and since they can be presented as genuine Java permissions at the service JVM, the service can further use them when calling unrelated Java code. In other words, we have basically extended the Java 2 access control mechanisms to distributed environments, allowing an application to pass any Java 2 permissions to a Jini service through the service proxy.[2]

Considering our future directions, there are two major branches. First, we want to test the applicability of our solution to the security of other Jini mechanisms, including leases, distributed events, and transactions. In particular, we would like to provide a kind of toolbox that would allow Jini service implementors to easily add strong, cryptographical security to these mechanisms without adversely affecting the environmental requirements of the services. Second, we plan to study how to integrate our approach with additional Java security mechanisms, including the forthcoming Java Authentication and Authorization Service (JAAS) [21] and the planned RMI Security Extension [32].

Another important aspect which requires further work is the integration of a certificate revocation or validation mechanism. The revocation and validation of SPKI certificates are discussed in e.g. [20].

7 Conclusions

The Jini approach provides a number of methods and tools for building distributed applications for decentralized, ad hoc network environments. However, the current state of the technology does not adequately address the security requirements present in many of such environments.

In this paper, we have briefly analyzed the client–service related security requirements relevant to typical Jini environments, and described a software architecture, backed up by an implementation, that provides decentralized solutions to these requirements. Our approach is based on applying SPKI based trust management to controlling Jini proxies and delegating Java 2 permissions between Jini clients, proxies, and services. Our experience indicates that it is indeed possible to build a completely decentralized security solution, and apply it to Jini without losing any of the intrinsic Jini properties. In particular, our implementation does not restrict the methods and protocols that can be used to implement communication between a service and its proxies. Our initial performance measurements indicate that, once optimized, the approach should provide adequate peformance in practice.

Acknowledgements

This work would not have been possible without the rest of the SIESTA team: Johannes Lehtinen, Antti Mannisto, Petra Pietiläinen, Satu Virtanen, and Jukka Zitting. We would also like to thank Yki Kortesniemi, Jonna Särs, and the anonymous reviewers for their valuable comments and suggestions, and Pekka Kanerva for helping in the preparation of this manuscript.

References

[1] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, June 1999.

[2] Dirk Balfanz, Drew Dean, and Mike Spreitzer. A security infrastructure for distributed Java applications. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 15–26, Oakland, California, May 2000.

[3] Konstantin Beznosov, Yi Deng, Bob Blakley, Carol Burt, and John Barkley. A resource access decision service for CORBA-based distributed systems. In *Proceedings of the*

2. As the astute reader quickly understands, the actual solution is not quite that easy due to the requirement of keeping the relevant certificates along all the time. That is, in addition to representing the permissions as Java objects, they must also be present in the form of properly authorized certificates. Only that creates a proper certificate chain that the server can use to verify the access permissions of the client application.

15th Annual Computer Security Applications Conference (ACSAC '99), pages 310–319, Phoenix, Arizona, December 1999.

[4] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system version 2. RFC 2704, IETF, September 1999.

[5] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. In Jan Bosch, Jan Vitek, and Christian D. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, Lecture Notes in Computer Science volume 1603, pages 185–210. Springer, 1999.

[6] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, California, May 1996.

[7] Geoffrey Clements. Jini Usersecurity project home page. http://www.bald-mountain.com/jini.html, 2000.

[8] Grzegorz Czajkowski and Thorsten von Eicken. JRes: a resource accounting interface for Java. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, pages 21–35, Vancouver, Canada, October 1998.

[9] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networks (MobiCom '99)*, pages 24–35, Seattle, Washington, August 1999.

[10] Tim Dierks and Christopher Allen. The TLS protocol, version 1.0. RFC 2246, IETF, January 1999.

[11] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylönen. SPKI certificate theory. RFC 2693, IETF, September 1999.

[12] Pasi Eronen, Johannes Lehtinen, Jukka Zitting, and Pekka Nikander. Extending Jini with decentralized trust management. In *Short paper proceedings of the 3rd IEEE Conference on Open Architectures and Network Programming (OPENARCH 2000)*, pages 25–29, Tel Aviv, Israel, March 2000.

[13] Eurescom. Jini and friends at work project home page. http://www.eurescom.de/Public/Projects/P1000-series/P1005/P1005.htm, 2000.

[14] Luigi Giuri. Role-based access control on the web using Java. In *Proceedings of the 4th ACM workshop on Role-based access control (RBAC '99)*, pages 11–18, Fairfax, Virginia, October 1999.

[15] Li Gong. *Inside Java 2 Platform Security: Architecture, API design, and implementation*. Addison-Wesley, June 1999.

[16] Erik Guttman, Charles Perkins, John Veizades, and Michael Day. Service location protocol, version 2. RFC 2608, IETF, June 1999.

[17] Peer Hasselmeyer, Roger Kehr, and Marco Voß. Trade-offs in a secure Jini service architecture. In Claudia Linnhoff-Popien and Heinz-Gerd Hegering, editors, *Trends in Distributed Systems: Towards a Universal Service Market. Third International IFIP/GI working conference proceedings (USM 2000)*, Lecture Notes in Computer Science volume 1890, pages 190–201, Munich, Germany, September 2000. Springer.

[18] Tero Hasu. Storage and retrieval of SPKI certificates using the DNS. Master's thesis, Helsinki University of Technology, April 1999.

[19] Trent Jaeger, Atul Prakash, Jochen Liedtke, and Nayeem Islam. Flexible control of downloaded executable content. *ACM Transactions on Information and System Security*, 2(2):177–228, May 1999.

[20] Yki Kortesniemi, Tero Hasu, and Jonna Särs. A revocation, validation and authentication protocol for SPKI based delegation systems. In *Proceedings of the 2000 Network and Distributed System Security Symposium (NDSS 2000)*, pages 85–101, San Diego, California, February 2000.

[21] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User authentication and authorization in the Java platform. In *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC '99)*, pages 285–290, Phoenix, Arizona, December 1999.

[22] Tuomo Lampinen. Using SPKI certificates for authorization in CORBA based distributed object-oriented systems. In *Proceedings of the 4th Nordic Workshop on Secure IT systems (NordSec '99)*, pages 61–81, Kista, Sweden, November 1999.

[23] Sanna Liimatainen et al. Tessa project home page. http://www.tml.hut.fi/Research/TeSSA/, 2000.

[24] Riku Mettälä. Bluetooth protocol architecture white paper, version 1.0. Bluetooth Special Interest Group, August 1999.

[25] Pekka Nikander. *An Architecture for Authorization and Delegation in Distributed Object-Oriented Agent Systems*. Ph.D. thesis, Helsinki University of Technology, March 1999.

[26] Pekka Nikander. Fault tolerance in decentralized and loosely coupled systems. In *Proceedings of Ericsson Conference on Software Engineering*, Stockholm, Sweden, September 2000.

[27] Pekka Nikander and Jonna Partanen. Distributed policy management for JDK 1.2. In *Proceedings of the 1999 Network and Distributed System Security Symposium (NDSS '99)*, pages 91–101, San Diego, California, February 1999.

[28] Salutation Consortium. Salutation home page. http://www.salutation.org/, 2000.

[29] Frank Stajano and Ross Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols, 7th International Workshop Proceedings*, Lecture Notes in Computer Science volume 1796, Cambridge, UK, April 1999. Springer.

[30] Christopher Steel. Securing Jini connection technology. Technical presentation 573 at the JavaOne 2000 conference, San Francisco, California. Slides available from http://java.sun.com/javaone/javaone00/, June 2000.

[31] Daniel F. Sterne, Gregg W. Tally, C. Durward McDonell, David L. Sherman, David L. Sames, Pierre X. Pasturel, and E. John Sebes. Scalable access control for distributed object systems. In *Proceedings of the 8th USENIX Security Symposium*, pages 201–214, Washington, D.C., August 1999.

[32] Sun Microsystems. Java remote method invocation security extension. Technical specification, early look draft 3, http://java.sun.com/products/jdk/rmi/rmisec-doc/, April 2000.

[33] Sun Microsystems. Java secure socket extension home page. http://java.sun.com/products/jsse/, 2000.

[34] Universal Plug and Play Forum. Universal plug and play home page. http://www.upnp.org/, 2000.

[35] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, November 1994.

[36] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 116–128, Saint-Malo, France, October 1997.

[37] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, California, May 1998.

[38] Ian Welch and Robert J. Stroud. Supporting real world security models in Java. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 155–159, Cape Town, South Africa, December 1999.