HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering

PASI ERONEN

SECURITY IN THE JINI NETWORKING TECHNOLOGY: A DECENTRALIZED TRUST MANAGEMENT APPROACH

Master's Thesis
March 6, 2001

Supervisor: Professor Arto Karila
Advisor: Pekka Nikander, Ph.D.

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF MASTER'S THESIS

Author:	Pasi Eronen
Title:	Security in the Jini Networking Technology: A Decentralized Trust Management Approach
Date:	March 6, 2001
Pages:	8 + 60
Department:	Department of Computer Science and Engineering
Chair:	Tik–110 Telecommunication Software and Applications (data networks)
Supervisor:	Professor Arto Karila
Advisor:	Pekka Nikander, Ph.D.

The Jini networking technology, developed by Sun Microsystems and based on the Java programming language, provides an elegant architecture for building distributed applications. However, 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.

This thesis analyzes the security requirements of Jini in different environments. Security threats and high level security goals are identified, and the implementation of these high level goals using lower level security mechanisms is described.

Based on the identified requirements, an architecture for Jini security is proposed. The architecture is based on the decentralized trust management approach, and uses Simple Public Key Infrastructure (SPKI) certificates for authorization. A prototype implementation of the architecture is also presented.

Keywords:	Java, Jini, RMI, SPKI
Language:	English

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Pasi Eronen
Työn nimi:	Turvallisuus Jini-verkkoteknologiassa ja hajautettu luottamuksenhallinta
Päivämäärä:	6. maaliskuuta 2001
Sivuja:	8 + 60
Osasto:	Tietotekniikan osasto
Professuuri:	Tik–110 Tietokoneverkot
Työn valvoja:	Professori Arto Karila
Työn ohjaaja:	TkT Pekka Nikander

Sun Microsystemsin kehittämä Jini-verkkoteknologia tarjoaa elegantin Java-ohjelmointikieleen perustuvan alustan hajautettujen sovellusten rakentamiseen. Se ei kuitenkaan ota riittävästi huomioon tietoturvaseikkoja, ja tämä haittaa sen soveltamista hajautetuissa järjestelmissä.

Tämä diplomityö tutkii sitä minkälaisia turvallisuusominaisuuksia Jinissä tarvitaan erilaisissa ympäristöissä, ja miten halutut korkean tason tietoturvatavoitteet voidaan toteuttaa alemman tason mekanismeilla.

Vaatimusten analysoinnin lisäksi diplomityössä rakennetaan tietoturva-arkkitehtuurin joka pohjautuu ns. hajautetun luottamuksenhallinnan käsitteisiin, ja käyttää Simple Public Key Infrastructuren (SPKI) mukaisia varmenteita tietoturva-asetusten määrittelyyn. Myös arkkitehtuurin prototyyppitoteutus esitellään lyhyesti.

Avainsanat:	Java, Jini, RMI, SPKI
Kieli:	Englanti

Preface

This work began in the SIESTA project organized under the HUT software project course in term 1999–2000. I would like to thank the the rest of the SIESTA team—Johannes Lehtinen, Antti Mannisto, Petra Pietiläinen, Satu Virtanen, and Jukka Zitting—for a great project and a memorable learning experience.

Pekka Nikander, the customer in the SIESTA project and later my advisor on this thesis, always found time to meet me and read my drafts. I am grateful for his encouragement and valuable comments which helped to shape this thesis.

Dieter Gollmann, Yki Kortesniemi, Helger Lipmaa, and Jonna Särs provided helpful comments, and Kristiina Volmari-Mäkinen helped with parentheses and other aspects of my English.

This thesis was written in the TeSSA 3 project at the Telecommunications software and multimedia laboratory of Helsinki University of Technology. The TeSSA 3 project was funded by Tekes and several industry partners.

Helsinki, March 6, 2001

Pasi Eronen
email: pe@iki.fi

Contents

Terms and abbreviations

ACL Access Control List.

CA Certificate Authority.

CORBA Common Object Request Broker Architecture, a distributed object architecture advanced by the Object Management Group.

CRL Certificate Revocation List.

DNS Domain Name System, a hierarchical distributed database for mapping domain names to IP addresses and other information.

DSA Digital Signature Algorithm, a public-key signature algorithm.

IDL Interface Description Language, a language used for describe remote object interfaces in, e.g., CORBA.

IEEE Institute for Electrical and Electronics Engineers, a professional association and technical publisher.

IETF Internet Engineering Task Force.

IIOP Internet Inter-ORB Protocol, the most widely used wire protocol in CORBA.

IP Internet Protocol, the basic network layer protocol of the Internet.

IPSEC Internet Protocol security architecture, an architecture for providing security services at the network layer. See RFC 2401.

JAAS Java Authentication and Authorization Service, an extension from Sun Microsystems which adds user-based authentication and access control to the Java 2 security architecture.

JAR Java Archive, a file format for Java applications. A JAR file can contain Java classes, digital signatures, and other files.

JCE Java Cryptography Extension, a part of Java libraries providing cryptographic primitives.

JDK Java Development Kit, a programming environment for the Java language developed and distributed by Sun Microsystems.

JSSE Java Secure Socket Extension, an optional Java package from Sun Microsystems which provides SSL/TLS sockets.

JVM Java Virtual Machine, an execution environment which interpretes Java bytecode.

LAN Local Area Network.

LDAP Lightweight Directory Access Protocol, a protocol for accessing directory services.

MAC Message Authentication Code, a cryptographic hash function involving a secret key.

PDA Personal Digital Assistant, a small handheld computer such as the Palm Pilot.

PKI Public Key Infrastructure.

PGP Pretty Good Privacy, a public key encryption application.

RFC Request for comments, a document series published by the IETF.

RMI (Java) Remote Method Invocation, a middleware component for distributed Java objects. See Section 2.4.

RSA Rivest-Shamir-Adleman, a public key cryptosystem which can be used for both encryption and digital signatures.

SPKI Simple Public Key Infrastructure, see Section 2.2.2.

SSL Secure Sockets Layer, a widely used protocol for providing secure connections over TCP.

TCP Transmission Control Protocol, a transport layer protocol providing reliable data streams over IP.

TeSSA Telecommunications Software Security Architecture project at Helsinki University of Technology.

TLS Transport Layer Security, the successor of the SSL protocol.

TTL Time-to-live, a mechanism used to limit the lifetime of IP packets.

UDP User Datagram Protocol, a transport layer protocol for providing unrealiable datagrams over IP.

UI User interface.

URL Uniform Resource Locator, e.g., “http://www.hut.fi/”.

WLAN Wireless Local Area Network, a LAN implemented using wireless technologies, such as radio or infrared.

X.509 ITU-T recommendation X.509. Specifies a widely used certificate format.

Chapter 1

Introduction

1.1 Background

Distributed computing is fundamentally different from centralized computing. The usually mentioned four major differences include latency, memory access, partial failures, and concurrency (e.g., [76]). 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.

Many approaches to distributed programming, such as CORBA or Java RMI, try to hide the differences between centralized and distributed programming from the programmer. Usually this means that local method calls, for example, look the same as remote calls. This has the benefit of making network programming easier, but has also some drawbacks. For example, dealing with network failures is harder.

The Jini networking technology, developed by Sun Microsystems, takes a different approach [1, 76]. Jini enables devices to form ad hoc communities without manual installation or intervention. Each device (or node) can provide services for other nodes to use. Since such ad hoc networks can be rapidly changing, the fact that networks are unreliable is not hidden from the programmer, and in fact, it is assumed that network failures do actually happen. Different applications require different ways of dealing with these problems, and making the failures visible to the programmers allows more fault-tolerant applications to be built [58].

Jini is based on the Java facilities for secure, downloadable code. It provides a programming model for building distributed applications which are resilient to network failures. In general, Jini looks like a promising technology for service discovery and communication in wireless ad hoc networks and group aware applications.

However, the Jini architecture does not currently include any security features in addition to the standard Java 2 facilities for protecting a Java Virtual Machine (JVM) from malicious code.

1.2 Research problem

In reality, computer networks are insecure, and some security features are desired. For example, the service may wish to authenticate clients, and based on who the client is, allow some operations and deny others. In distributed systems, this functionality is achieved using cryptographic protocols. For example, the Transport Layer Security (TLS) protocol supports authentication of both the client and the server using public keys and X.509 certificates [23].

In Jini, however, all communication (except for bootstrapping the system) goes through *proxies*, which are objects downloaded from the network. Proxies are described in detail in Section 2.5. Since the communication protocol is implemented by untrusted code—untrusted from the client's viewpoint, at least—security methods used in environments with fixed protocols cannot be used without some adaptation. For example, to authenticate the client to the server the TLS protocol needs access to the user's private key. The client certainly does not want to give the key to a piece of untrusted code since the code might use it to access a completely different service, or reveal the private key to a third party.

There is a small number of existing solutions for Jini security [18, 39, 68], but none of them adequately solves the problem. The proposed solutions are based on a centralized security architecture, which limits their usability in environments where centralized authorities do not exist naturally, e.g. ad hoc networks. The new RMI security API [71] promises more flexibility, but it is not yet available. The existing solutions are presented and compared with this work in Section 6.5.

This thesis has two major goals. First, my aim is to analyze what security features are needed in Jini. Second, I propose a solution architecture which implements a subset of the possible requirements identified. The solution architecture is evaluated using criteria presented in Section 1.3, and is verified to be feasible with an implementation.

The work was started in the SIESTA project, at the HUT Software project course in term 1999–2000. The SIESTA project designed and built a security library for Jini, a framework for managing calendar information in PDA devices, and a demonstration application. I was responsible for the design of the security architecture, and most of its implementation. Jukka Zitting wrote the RMI over TLS part (described in Section 5.10), and both he and Johannes Lehtinen provided some help with other security aspects as well.

The results of the SIESTA project were presented in a conference paper written together with Pekka Nikander [28]. The ideas were developed further in two conference papers, written together with Pekka Nikander and Christian Gehrmann [27, 29]. This thesis presents further work in the area, and collects the results into one easily approachable package.

1.3 Evaluation criteria

To evaluate the proposed architecture, and to compare it with the other proposals, I use the following criteria:

- *Security functionality*—What security features does the solution provide? Does it have convenient points of control for implementing additional features later?
- *Minimized trust relationships*—What kind of trust relationships does the solution assume? Are the assumptions flexible enough so that the solution can be applied in various environments?
- *Protocol independence*—How well does the solution preserve Jini's protocol independence, and the flexibility it brings?
- *Elegance*—How elegant is the solution? That is, is it easy to understand, technologically justified, state of the art, etc.?
- *Simplicity*—How transparent the solution is to applications, and how easy is it to use?

These criteria are described in more detail in Chapter 6 where they are used to evaluate the proposed architecture and to compare it with related work.

1.4 Limitations

In this thesis, I concentrate on providing security for a client accessing a service. The requirements of any specific Jini service, such as the lookup service or JavaSpaces, are not considered. The security aspects of Jini's distributed events, leases, and transactions are also left for future work. Modifications to the Java 2 virtual machine (JVM) or the standard libraries are also beyond the scope of this thesis.

1.5 Organization of this thesis

Chapter 2 presents a short introduction to decentralized trust management and SPKI, and to Java 2 security and Jini.

Chapter 3 analyzes the security requirements of Jini in different environments, and some related design aspects. Chapter 4 presents the proposed solution architecture, and Chapter 5 describes a prototype implementation of it. The architecture and implementation are evaluated in Chapter 6. Finally, Section 7 contains a summary and the conclusions from this research.

Chapter 2

Background

In this chapter, the fundamental technologies that form the basis for this thesis are described. First, Section 2.1 presents *wireless ad hoc networks*. Section 2.2 gives an overview of relevant concepts and technologies from the field of computer security. The rest of the chapter describes existing Java technologies, including the Java 2 security architecture, Java Remote Method Invocation (RMI), and Jini.

2.1 Wireless ad hoc networks

Wireless ad hoc networks are small networks of nodes connected using a short range wireless technology, such as radio or infrared. They are *ad hoc*, which means that they are established for a short duration, and do not require any fixed infrastructure or administration.

Jini was designed to handle changes in the network environment without manual interventions. In *wireless ad hoc networks* such changes occur often, and thus Jini seems a promising technology for application level communication in such networks.

The most widespread wireless ad hoc networking technology today is the IEEE 802.11 wireless LAN [43]. When a group of people with WLAN equipped laptops gather in a room, an ad hoc network is automatically formed. It must be noted, though, that not all wireless LANs are capable of ad hoc operation; some require special nodes called access points which are typically fixed equipment.

Another new technology which promises ad hoc networking capabilities on an even smaller scale is Bluetooth, originally intended as a replacement for cables for mobile phones, digital cameras, MP3 players, PDAs, etc. [38, 56]

2.2 Computer security

According to Dieter Gollmann [35], "computer security deals with the prevention and detection of unauthorized actions by users of a computer system." Usually, this includes at least the protection of *confidentiality*, *integrity*, and *availability*, as defined in e.g. [35]. Sometimes *accountability*, and even *dependability*, are also added to this list.

In actual systems, the protection of these properties are achieved through various security services and mechanisms. From the point of view of this thesis, the most important services are *authentication*, *authorization*, and *access control*. In this work, authentication means verifying a claimed identity. Authorization means granting access to a restricted resource to someone, and access control mechanisms enforce these restrictions. In distributed systems, these functions are usually supported by various cryptographic primitives and protocols.

2.2.1 Decentralized trust management

Traditionally, access control has been based on identity authentication and locally stored access control lists (ACLs). The most popular method for identity authentication is probably user names and passwords. Another widely used method is to rely on public keys with identity certificates. Basically, identity certificates, such as X.509 [44], bind a human-readable name to a public key. It is important to notice that these certificates are fundamentally different from authorization certificates, described below.

Access control lists describe what access rights a user has for a resource. For instance, an entry in a list can grant Alice a read permission to some file. However, when applied to a distributed system, the ACL approach has a number of drawbacks. For instance, operations which modify the access control list need to be protected somehow. To illustrate this issue, the following example is given by Ellison et al. [26].

> Imagine a firewall proxy permitting telnet and ftp access from the Internet into a network of US DoD machines. Because of the sensitivity of that destination network, strong access control would be desired. One could use public key authentication and public key certificates to establish who the individual keyholder was. Both the private key and the keyholder's certificates could be kept on a Fortezza card. That card holds X.509v1 certificates, so all that can be established is the name of the keyholder. It is then the job of the firewall to keep an ACL, listing named keyholders and the forms of access they are each permitted.
>
> Consider the ACL itself. Not only would it be potentially huge, demanding far more storage than the firewall would otherwise require, but it would also need its own ACL. One could not, for example, have someone in the Army have the power to decide whether someone in the Navy got access. In fact,

> the ACL would probably need not one level of its own ACL, but a nested set of ACLs, eventually reflecting the organization structure of the entire Defense Department.

Indeed, in [15] Blaze et al. argue that "the use of identity-based public-key systems in conjunction with ACLs are inadequate solutions to distributed (and programmable) system-security problems."

Trust management, introduced by Blaze et al. [16], proposes an alternative solution. Basically, trust management uses a set of unified mechanisms for specifying both security policies and security credentials. The credentials are signed statements (certificates) about what principals (users) are allowed to do. Thus, even though they are commonly called certificates, they are fundamentally different from traditional name certificates. Usually the access rights are granted directly to the public keys of users, and therefore trust management systems are sometimes called key-oriented PKIs [6].

The unified mechanisms are also designed to separate the mechanism from the policy. Thus, the same mechanisms for verifying credentials and policies (a trust management engine) can be used by many different applications. This is unlike access control lists whose structure usually reflects the needs of one particular application.

Examples of trust management systems include the PolicyMaker, which originally introduced the term trust management [16], its successors KeyNote and KeyNote 2 [14], and in some respects, SPKI [26] and its applications, including TeSSA [54]. Blaze et al. compare PolicyMaker, KeyNote, and SPKI in [15]. SPKI is described in next in Section 2.2.2, and the TeSSA approach in Section 2.2.3.

2.2.2 SPKI

One particular example of a trust management system is SPKI, or Simple Public Key Infrastructure [25, 26]. In SPKI, principals are identified by their public keys. Access rights are granted to users using *authorization certificates*. If permitted, the access rights can be further delegated, forming a *certificate chain*, as described below.

An SPKI authorization certificate has five security-related attributes: *issuer, subject, delegation, tag,* and *validity*. *Issuer* is the public key of the principal who issued the certificate, and the whole certificate is signed with the corresponding private key. *Subject* is the public key of the recipient of the permissions. *Delegation* is a boolean flag telling whether the subject may authorize other users for the same actions. *Tag* is a service-specific field describing the permissions included in the certificate. *Validity* describes the conditions under which the certificate is valid—for example, the time of expiration, or reference to a certificate revocation list (CRL).

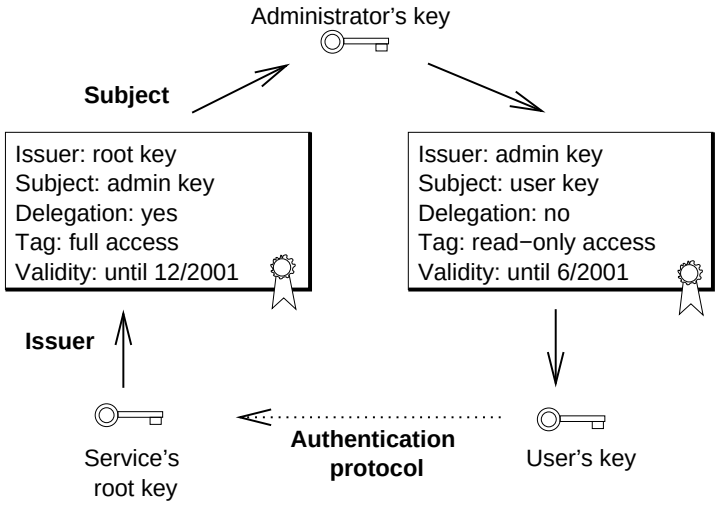

Figure 2.1: Simple SPKI certificate loop.

When using SPKI certificates, the service configuration usually specifies one "root key" which is implicitly allowed to perform any actions. Using this key, permissions are delegated to the administrators of the service, and the administrators then further delegate access rights to the users. When accessing the service, the user provides a certificate chain from the service's root key to the user's key. The chain is then completed into a loop, usually by signing the request with the user's key, and verifying the signature. The service then checks whether the chain is valid and actually implies the requested action. An example of a certificate loop is shown in Figure 2.1.

In addition to authorization certificates, SPKI also has name certificates and attribute certificates. Name certificates bind a public key to a name (or a group). The SPKI name certificates are adopted from SDSI [64] which introduced the concept of using local names globally. That is, instead of trying to create a globally unique name space, such as the X.500 Distinguished Names, SDSI names are always local to the issuer of a name certificate. The issuer is identified by a public key. Public keys are globally unique due to the way they are generated, and thus the combination is globally unique as well.

SPKI attribute certificates bind an authorization to a name or a group. Attribute certificates are usually used together with name certificates. The reason for separating them is usually that they come from different issuers.

2.2.3 TeSSA approach

The goal of the Telecommunications Software Security Architecture (TeSSA) project [54] at Helsinki University of Technology was to create a general purpose security architecture based on SPKI.

The results have given a much better picture of how SPKI could be deployed in the real world, and form much of the technical, conceptual, and intellectual background for this work. A large number of papers has been published. The results can be traced from an early paper on trust by Lehti and Nikander [52], to applications in Java 2 security [59, 62], and Nikander's dissertation [57], to give a few examples.

2.3 Java 2 security architecture

The Jini architecture relies heavily on the security features in the base Java technology. Security was one of the main goals in the design of the Java language and execution environment. The security features were originally designed for "applets"—that is, small applications embedded inside web pages—but have since then received numerous other uses. When running inside a web browser, an applet should not be allowed to access sensitive resources, such as the user's files, or open arbitrary network connections, for instance.

The security architecture of Java 2 is described in [36], and can be considered to consist of the following components.

- Java language and platform: type safety and isolation.
- Resource access control: policy and enforcement.
- Cryptography architecture.

The Java language and platform security are described in the next section, and resource access control in Section 2.3.2.

The third important component is the cryptography architecture. It provides access to cryptographic algorithms, such as message digests, digital signatures, symmetric and asymmetric ciphers and key agreement algorithms. It is used as a building block in the construction of the other security mechanisms. However, as cryptography itself is not an essential element of this thesis, but only a tool, a full description of the Java 2 cryptography architecture is beyond the scope of this thesis.

2.3.1 Java language and platform security

The Java language is designed to be *type safe*. This means, for instance, that no Java program can ever refer to an object using an incorrect type, refer to an unassigned memory location, or "forge" pointers from integer types. Also, access restrictions (private, public, package local) on classes, methods, and fields cannot be violated.

Some of these type checks are performed by the compiler, but Java is usually compiled into an intermediate platform-neutral form called *byte code*. This intermediate form is interpreted by a *Java Virtual Machine* (JVM). The actual checks must be performed on the byte code, since it is possible to bypass the compiler and write byte code by hand.

In the JVM, type safety is implemented using runtime checks (for example, type casts) and the *byte code verifier*. The byte code verifier checks the code when it is loaded, and ensures that it respects the Java language rules. The byte code verifier is a very complex piece of code, and most of the security bugs found in Java implementations so far have been in the byte code verifier [73].

In addition to type safety, untrusted code needs to be *isolated*. In Java, the isolation is provided by *class loaders*. Class loaders are responsible for mapping class names (e.g., "java.lang.String") to the corresponding byte code, and loading the byte code from a file or from the network. The mapping is context-dependent: there can be two classes with the same name running inside a single JVM, provided that they are loaded with different class loaders. The class loaders are themselves written in Java, and programmers can write new class loaders, if necessary.

Class loaders also interact with type safety. Because there can be more than one class with the same name, references to names must be resolved consistently, i.e., in a way which preserves type safety. The interaction of class loaders with typing is discussed in e.g. [22].

2.3.2 Resource access control

The resource access control framework is responsible for controlling access to valuable system resources, such as the file system. This part of the infrastructure has considerably evolved during the history of Java: both the enforcement and policy mechanisms are now more flexible and fine-grained than in the original Java 1.0.

In JDK 1.0 and 1.1, all code was either untrusted or completely trusted. Untrusted code was run side a *sandbox*, which limited its access to sensitive operations. In JDK 1.0, all code loaded from the local file system was considered trusted, and everything else (e.g. loaded from the network) untrusted.

However, sometimes applets have a legitimate need to access some protected resources. Thus, JDK 1.1 introduced the notion *signed applets*. In Java the byte code for an appli-

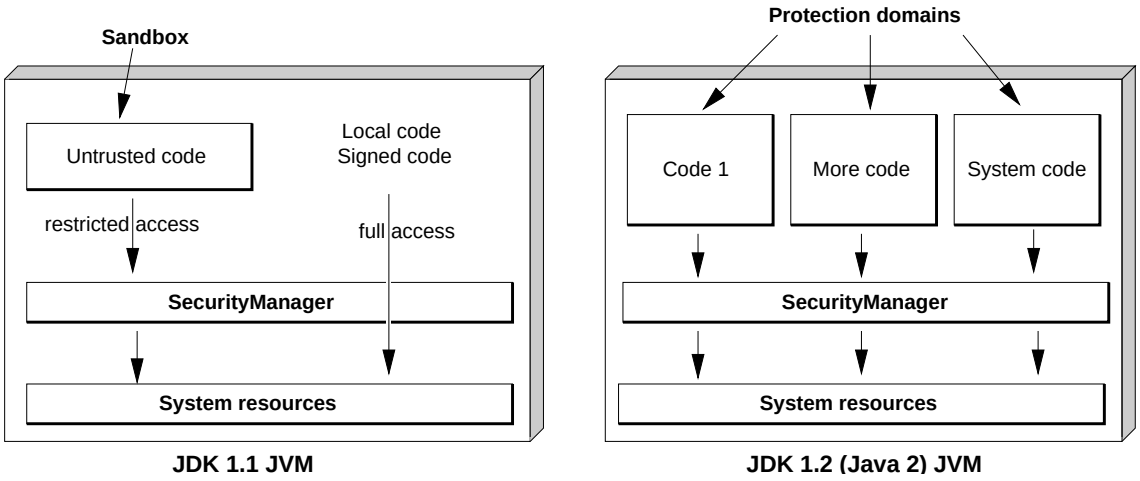

Figure 2.2: JDK 1.1 model (sandbox) and JDK 1.2 protection domains compared. In JDK 1.1 the code is either in the sandbox or trusted. In Java 2 the code is divided into different protection domains.

cation is usually stored in a Java archive (JAR) file. The JAR file can also include a digital signature. If the JAR file was signed by a trusted key, the code was considered trusted, even if the JAR file itself was loaded from the network.

In Java 2 (JDK 1.2) the security architecture was almost completely redesigned. Code is no longer treated simply untrusted or completely trusted, but is divided into protection domains. All the code running inside one protection domain share the same access permissions, but there can be as many protection domains as necessary. Classes are assigned to protection domains based on the URL and digital signatures of the JAR file. This is illustrated in Figure 2.2.

The permissions granted to protection domains are also more fine-grained than in JDK 1.1. For instance, it is possible to grant a permission to read only a particular file, or open a network connection to a single port on a single host. The set of permissions is not fixed but can be extended by programmers to protect application-specific resources.

When checking permissions, the access control mechanisms check the Java call stack. The effective permissions are the intersection of the permission of all the classes in the call stack, from the topmost to the first “privileged” stack frame, or the whole stack if none of the frames is marked as privileged. Privileged frames allow a piece of code to perform some operation with its own privileges, regardless of who originally called it. For a more detailed explanation, see e.g. [36].

2.3.3 Extensions

Since the introduction of Java, many researchers have modified its security mechanisms to provide additional functionality [42, 47, 77, 78]. The policy mechanisms have been extended with decentralized trust management in [59]. The concept of "who is running the code" has been implemented in the Java Authentication and Authorization Services (JAAS) [51], and has been extended with roles in [34]. Controlling the amount of resources, such as computational cycles, memory, etc., a program can use is discussed in [20].

While these extensions are interesting, they are beyond the scope of this thesis.

2.4 Java Remote Method Invocation

Java Remote Method Invocation (RMI) is a middleware building block for distributed Java applications. It provides object-oriented remote method calls for Java [69, 80, 81]. RMI is somewhat similar to the CORBA architecture, but being Java-specific gives it some unique features.

In RMI invocation of a method on a remote object—residing in a different JVM or on a different machine—looks the same to the caller as a local call. The RMI middleware takes care of converting parameters to a stream of bytes, opening the network connections, invoking the method on the remote object, and passing the return value back to the caller.

As in other similar architectures, remote objects are described by one or more interfaces. In RMI the interfaces are specified as normal Java interfaces, so no separate Interface Definition Language (IDL) is required. The RMI compiler, *rmic*, reads the Java class files and generates *stubs* which represent the remote object on the client and are responsible for e.g. converting the parameters to byte streams and return values back to Java objects.

In RMI the stubs can be downloaded from the network using Java's standard facilities for downloadable code—unlike in CORBA, for instance. This allows the stubs, in theory at least, to implement application-specific functionality on the client side, but this functionality is seldom used. In fact, the stubs implementing the standard functionality are no longer downloaded from the network in the next major Java release (JDK 1.4, codenamed "Merlin"), but are generated on the fly on the client side [66].

"Skeletons", which are present in other similar technologies, were used in earlier versions of Java—for converting the parameters to Java objects on the server side—but are no longer necessary. One interesting feature in RMI not present in most distributed object architectures is distributed garbage collection; the details of it are beyond the scope of this thesis.

2.5 Jini

The Java security architecture and RMI are used heavily in the Jini networking technology, also developed by Sun Microsystems. Jini enables devices to form ad hoc communities without manual installation or intervention. Each device (or node) can provide services for other nodes to use, and any necessary "device drivers", or other required pieces of software, are downloaded to the client automatically.

In some aspects, Jini is quite similar to other service location protocols, such as Salutation [65], Service Location Protocol [37], and Universal Plug and Play [75]. However, Jini is more than just a protocol for downloading device drivers. The Jini architecture provides basic building blocks for building any kind of distributed applications: distributed events, transactions, leases, and downloadable proxies. These do not try to hide the fact that networks are unreliable, and the approach, in general, encourages building more fault-tolerant applications [58, 76].

Perhaps the largest 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, as described below), 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 RMI stubs which marshal 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 do not 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 do not 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.

This protocol independence and the generic programming model are what really separates Jini from mere service location protocols. Although Jini was originally originally intended to provide similar functionality, its flexible and elegant distributed programming model has since received other uses as well (e.g., [2, 3, 40]).

2.5.1 Discovery

The only part of Jini relying on some fixed protocol, i.e., that needs to be implemented in non-downloaded code, is the discovery part. That is, obtaining the proxy object for a lookup service. This part of Jini is also specific to the underlying network protocols used. Currently, the Jini specification describes discovery protocols for TCP/IP networks [1].

There are actually three related discovery protocols: unicast discovery, multicast discovery, and multicast announcements.

- *The unicast discovery protocol* is used when a client knows the IP address and the port the lookup server is running on, and wishes to connect to this particular lookup service. Basically, the serialized proxy object for the lookup service is transmitted over a TCP connection.
- *The multicast discovery protocol* is used when a client wants to find all "near by" lookup services. It works by sending a multicast UDP packet with a small time-to-live value. All lookup servers which receive the multicast packet respond using server-initiated version of the unicast discovery protocol.
- *Multicast announcements* are used by lookup servers to periodically announce their presence to clients. This way, when a lookup service is temporarily down, services know when it is back on-line and ready to accept registrations again. The announcement contains the IP address and port number; interested clients can reply using the unicast discovery protocol.

While the current specification describes protocols for TCP/IP only, specifying protocols for other transports, such as Bluetooth, should be quite straightforward.

2.5.2 Lookup service and leases

The discovery protocols are used to download the proxy for the lookup service. The lookup service is a directory where service providers register themselves and clients search for what they need. The lookup service also provides a good example of the use of leases. When a service registers itself with a lookup service, it receives a *lease* on the registration, with an expiration time. If the service does not renew the lease before it expires—for example, the service is disconnected from the network—the registration is automatically cleaned from the lookup service. In general, leases are used in Jini whenever a server allocates some resources for a client.

When registering in the lookup service, a service provides a serialized proxy objects and a set of attributes. Clients can search for services based on the attributes and interfaces implemented by the proxy.

2.5.3 Other aspects of Jini

In addition to leases and the lookup service, Jini also provides distributed events, transactions, and a “tuple-space” distributed programming model called JavaSpaces. The Jini specification also defines a number of other standard services, such as event mailbox and lease renewal services. [1, 33]

Chapter 3

Requirements for Jini security

Before designing a security framework for Jini, it is necessary to decide what kind of security functionality is required. Naturally, this depends on the concrete applications written using Jini and on the trust relationships involved. As described in Section 1.4, in this thesis I concentrate on providing security for a client accessing a "generic" Jini service. The security requirements of any specific service, and Jini's events, leases, and transactions, are left for future work.

My goal in this chapter is to identify requirements for security functionality. Any actual application requires only a subset of these requirements, and identifying the relevant subsets is a difficult problem of its own. It is also important to keep in mind that some requirements may conflict, and not everything is technically feasible (cf. [32]).

This chapter is organized as follows. First, Section 3.1 presents a list of security services identified in the OSI and CORBA security frameworks. These lists are used to organize the discussion in the following sections and to identify the kind of functionality which is not covered by the requirements presented here.

Section 3.2 outlines a few examples where Jini might be used, and identifies some threats for each of them. Protection against these threats leads to a small number of high level goals. In actual systems these goals must be implemented using lower level mechanisms. The high level goals and their mapping to lower level requirements is presented in Section 3.3. Section 3.4 lists some security issues that are left for further work. Finally, Section 3.5 presents some related design aspects.

3.1 Security frameworks

As stated in Section 2.2, computer security deals with the prevention and detection of unauthorized actions by users of a computer system [35]. To accomplish this prevention and detection in a networked environment, the OSI security framework lists a number of security threats and defines the following security services [45, 46, 48].

- Authentication
- Access control
- Non-repudiation
- Confidentiality
- Integrity
- Security audits and alarms

In his dissertation Karila provides a good overview of the OSI security framework, and also criticizes it on some aspects [48]. For instance, the OSI framework does not address the dependencies between the security services very well. For example, message integrity and authentication are usually not completely separate mechanisms.

The OSI framework deals mostly with protocol level security services. It does not directly address the implementation aspects which are not visible on the network. Jini is somewhat different in this respect, because some of the implementation aspects are visible due to using downloaded code.

Although the OSI security framework is not perfect, the security services identified in it are used as a starting point for the discussion in this chapter.

An alternative starting point could be the CORBA security framework [61]. The framework defines the following security functionality.

- Identification and authentication
- Authorization and access control
- Security auditing
- Security of communications
- Non-repudiation
- Administration

The CORBA security takes a somewhat broader view than just network security, as can be seen from the inclusion of authorization and administration in the list. As a security framework for object-oriented middleware it is probably closer in spirit to Jini and RMI than the OSI framework. However, it still does not address the issues caused by downloaded code.

3.2 Examples and threats

The purpose of this section is to motivate the requirements listed in the next section by presenting a couple of examples where Jini might be used, and identifying some threats for each of them.

Example 1: An engineer uses a word processor to print a confidential document to a printer which is attached to the company network.

In this case, the main threats are the *disclosure* and *modification* of the document. An attacker might achieve this by intercepting or tampering with the network connection, or masquerading as the printer. In some cases denial of service might be an issue, too. For example, blocking a high security printer could cause the user to decide to use a low security one.

It would also be good if the security features would be transparent to the word processing application.

Example 2: A small handheld device with wireless networking is used to control heating, lighting, audio/video equipment, etc., at home.

In this case, the main threat is probably *unauthorized use* of the services. Again, this might be achieved by tampering with the network communications, or masquerading as the client device. Another threat to consider is *gaining access to information or service* on the handheld device, possibly by uploading a malicious proxy. Denial of service might be very annoying, but a properly designed system would still have manual controls.

Example 3: A cluster of machines is used to provide some kind of web services, and the requests and responses are communicated using Jini (similar to Concept Technologies' Hosta product [19]).

Disclosure and *modification* of the information are threats in this case as well, as are *unauthorized use of the services* provided, and *denial of service*.

3.3 Identified requirements

3.3.1 High level goals

The previous sections outlined some of the threats present when using Jini. Security functionality could be used to mitigate these threats. Loosely speaking, when using a Jini service, we would like to have security functionality to ensure the following.

- **Goal 1:** The user is talking to the right service (and the service to the right user), and the parties remain the same throughout the session (mutual authentication).

- **Goal 2:** Nobody else can listen on the communications (communication confidentiality).
- **Goal 3:** Only legitimate users can access the service (authorization and access control).
- **Goal 4:** Untrusted code, e.g. applications and proxies, cannot cause any harm (protection from untrusted code).

These goals do not cover all possible situations, and additional goals might be beneficial. For instance, the service should be available to legitimate users when they want to access it. Also, it should be possible, at some later point of time, to prove important actions to a third party, such as an impartial judge. More of these open questions are described in Section 3.4.

3.3.2 Mapping goals to requirements

The goals, as defined above, are quite high level. To realize the goals, they have to be mapped to some lower level security mechanisms or functions, which in turn can be mapped to simple, implementable primitives.

For goal 1, mutual authentication, the following components are likely to be needed. The numbers in parenthesis refer to a more complete discussion in the next section.

- *Authentication user interface (1)*—The user has to be able to distinguish the "right service" from all other services. This usually requires some kind of user interface to communicate the user's intent to the software.
- *Authentication of proxy identity/credentials (2)*—Services are represented by proxies; to ensure that the right service is being used, the client needs to verify that the downloaded proxy actually represents the service the user wants to use.
- *Authentication and key agreement protocol (3)*—The proxy has to know that it is communicating with the right service. Correspondingly, the service has to know that it is communicating with an authorized client[1]. To perform this over an insecure network, a cryptographic authentication and key agreement protocol is a required.
- *Message and connection integrity (4)*—After the proxy and the service have agreed on a session key, it must be possible to use the key to ensure the integrity of subsequent messages.

[1]Note that there is no way the server can ensure that the proxy is really the one provided by it. For example, even an authorized client can modify the proxy's code to perform actions not intended by the designer of the service.

For goal 2, communication confidentiality:

- *Message encryption (5)*—To protect the contents of the messages from eavesdropping, they have to be encrypted.
- *Authentication and key agreement protocol (3)*—The parties need to agree on a key to perform encryption. Authentication is required for ensuring that the session key is shared with the right party (and not a man-in-the-middle, for instance).

For goal 3, authorization and access control:

- *Message and connection integrity (4)*—If anyone can change messages on the network, we do not know enough to perform access control.
- *Authentication of client identity/credentials (through the proxy) (6)*—The set of allowed actions is usually based on the identity of the client and/or any credentials presented.
- *Local security policy (7)*—Security policy specifies what kind of authentication and credentials are required to perform different operations.

For goal 4, protection from untrusted code:

- *Access control mechanisms for local resources (8)*—Since there are resources which need to be protected, some kind of access control mechanisms are required.
- *Resource consumption control (9)*—Access control alone is not usually sufficient for protecting against denial of service. Mechanisms for controlling the amount of resources used are also required.
- *Local security policy (7)*—Local security policy specifies what kind of operations are allowed, based on the proxy's identity and credentials.
- *Authentication of proxy identity/credentials (2)*—Proxies have to be authenticated so access control can be performed. [2]

The next section provides an ordered list of the numbered requirements, together with discussion.

[2]The reasoning here goes as follows. The client must trust the server in some respect, since it is requesting services from it. The server signs the proxy to assure its authenticity. Due to the trust placed on the server, it is reasonable to trust the proxy, too, within the limits placed on the trust to the server.

3.3.3 Identified requirements

Requirement 1: Authentication user interface.

Authentication protocols often verify the possession of a private key corresponding to some public key. To ensure that the user is talking to the right service, the user has to be able to distinguish the "right service" from all the other services. Thus, the public key of the service is usually mapped to some user level concept, e.g., a name. This mapping might involve a name certificate, for instance. However, names are not useful in all circumstances, and cryptography is not necessarily involved.

Stajano and Anderson [67] 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 does not 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. However, this solution is not 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.

It is important to notice that authentication is impossible in a number of situations. For example, in a pure ad hoc network there may not be any prior information about the communicating peers. However, in such situations, physical or topological proximity and reachability can still be used to create some level of security.

Requirement 2: Authentication of proxy identity/credentials.

Services are represented, or sometimes completely implemented, by the proxy objects. To ensure that the right service is being used, we need to verify that the downloaded proxy actually represents the desired service. The proxy consists of Java byte code and the serialized state; both of them have to be authenticated.

Requirement 3: Authentication and key agreement protocol.

The proxies usually communicate with the actual service using some communications protocol. The parties need to agree on a key to perform encryption. Authentication is required for ensuring that the session key is shared with the right party, and not a man-in-the-middle, for instance.

There are a number of different approaches to this, such as authenticated Diffie-Hellmann for public keys, or the Kerberos protocol, which involves a central "ticket granting server".

Requirement 4: Message and connection integrity.

Naturally, nobody should be able to modify, insert, delete, or replay network messages undetected. Usually some cryptographic mechanisms, such as message authentication codes (MACs), are used to guarantee integrity, or more correctly, to detect modifications. The cryptographic mechanisms usually use the session key generated using an authentication and key agreement protocol.

In some special cases, message integrity can be achieved by other means. For example, if the network is assumed to be secure—such as a closed network, enforced by physical security—cryptography is not required.

Requirement 5: Message encryption.

To protect the contents of the messages from eavesdropping, they need to be encrypted. As in the case of integrity, communication confidentiality may be achieved by other means than encryption.

Requirement 6: Authentication of client identity/credentials (through the proxy).

To ensure that only legitimate users can access the service, we need to authenticate the identity and/or credentials presented by the client. Moreover, this authentication must be securely bound to the authentication of the server to the client, so that the server can be assured that the client indeed wanted to access it.

Requirement 7: Local security policy.

Based on the authentication of the client identity and credentials, and possibly other circumstances, the service should allow operations which are properly authorized, and deny others. The security policy can be implemented as a simple access control list (ACL) based on the client identity, or it can also consider other credentials, such as authorization certificates.

Similar authorization mechanisms are also needed for local resources. The standard Java 2 security architecture provides mechanisms for enforcing fine-grained access policies to sensitive resources, such as the file system. The standard mechanisms for specifying the security policy, however, are not very flexible. In some cases, limited access to sensitive resources is required, and in such cases, more flexible authorization mechanisms are probably needed.

Requirement 8: Access control mechanisms for local resources.

The standard Java 2 security architecture provides mechanisms for enforcing fine-grained access policies to sensitive resources, such as the file system. Private security credentials, such as keys, passwords, or Kerberos tickets, are certainly very sensitive, and mechanisms for protecting them are also required.

Access control is not limited to downloaded proxies. In most workstation operating systems, applications are run with user's privileges. The Java security architecture, however, was designed for environments where not all code is equally trusted, and allows implementation of more fine-grained control of access to sensitive resources. Untrusted applications should not be allowed to access Jini services with the user's privileges, and definitely should not be allowed to access the user's private credentials, such as keys, passwords, or Kerberos tickets.

This feature can be used, for example, to protect the system from Trojan horse applications. On workstations multiple JVMs with different security policies can be used to achieve similar results, but better solutions are needed in environments where a single JVM functions as the operating system of a small device (e.g., [10, 12]).

Requirement 9: Resource consumption control.

The standard Java platform does not have any mechanisms for limiting the amount of resources a program can use. Thus, a malicious proxy could simply allocate all the available memory in the system, and thus create a denial of service condition. This could be especially serious in an environment where a single JVM functions as the operating system.

Some researchers have modified the Java Virtual Machine to include such restrictions, e.g., [9, 13, 20].

3.4 Open questions and limitations

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. In many cases, the existing client side resource authorization mechanisms might be sufficient. There are, however, several possible security mechanisms not listed. Next, I discuss some of these, and describe why I feel that they do not belong to the list of requirements.

- *End system security*—Most computer security issues at the end systems are beyond the scope of this thesis. For instance, message encryption alone does not necessarily protect information from disclosure if, for example, the data can be observed before it

is encrypted or after it has been decrypted. Similarly, modifying important information by bypassing the access control mechanisms must be prevented.

It is the responsibility of the application to take appropriate measures to protect the data, and the underlying platform must provide support for this. The Java Virtual Machine provides basic mechanisms, such as type safety and isolation, but they, of course, must be used correctly.

Issues such as multi-level confidentiality and integrity models are also beyond the scope of this thesis.

- *Traffic flow confidentiality*—Protection against traffic analysis is a very difficult topic, depends heavily on the underlying protocols (such as TCP/IP), and is probably unnecessary in most environments.

- *Non-repudiation*—An underlying cryptographic protocol may provide some kind of proof of origin for messages. However, since non-repudiation usually involves legal aspects as well, it should be implemented at the application level—for instance, in an e-mail or EDI application.

- *Availability protection*—Protecting against denial of service (DoS) attacks, especially those that work by consuming some finite resource, is difficult. The attacks could target either the client or the service.

  The requirements in the previous section included resource consumption control against downloaded code, but other scenarios are possible. For example, a malicious client (or a number of clients) could simply send more service requests than a server can handle, and thus deny service to other, legitimate requests. There are some countermeasures, such as “stateless connections” and “client puzzles”, which could be used by the service (and the proxy) to offer some protection against such attacks [7, 8, 53].

  In addition, the current Jini discovery protocols do not offer much protection against denial of service attacks.

- *Audit, alarms, and management*—Requirements for auditing the use of a system, and other security management aspects, vary greatly from application to application.

3.5 Other design aspects

The requirements outlined in the previous section 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. The next chapter continues to set forth the choices made for this thesis and the reasons behind them.

The following aspects are related to the general software architecture and structure.

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

  A centralized security architecture probably makes administration easier in large networks. On the other hand, it does not work well for, e.g. 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*—What software components and nodes are assumed to be secure? For instance, does the system rely on the security of the lookup service, or some other on-line security server?

- *Placement of security mechanisms*—The security mechanisms could be placed at several different places. For example, message confidentiality and integrity could be protected at the link layer, the network layer using IPSEC, using TLS above the transport layer, or with some application-level protocol.

- *Point of proxy authentication*—Are proxies authenticated before or after the object is instantiated in the JVM? If the object is instantiated before authentication, untrusted code is run on the JVM. Access control mechanisms should prevent it from doing any damage, but at least denial of service attacks are possible.

The following aspects are related to security functionality.

- *Service authorization and 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 Enterprise JavaBeans style container, the granularity is at most per method.

- *Security mechanism authorization and access control model*—How flexibly can the user decide which client applications are allowed to do what? Also this is clearly influenced by other design choices. For example, if the system uses TLS client authentication and a server side access control list (ACL), the restrictions cannot 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?

Finally, there are several aspects which are not directly related to security, but nevertheless are important to consider.

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

  Sometimes using a specific protocol is necessary, for instance, to interoperate with existing services. On the other hand, fixed protocols limit the flexibility of the system.

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

  For example, in Enterprise JavaBeans [70] security is is managed by the middleware components, so it is quite transparent to the service software. It is probably a good idea to make the security as transparent as possible to client applications.

- *Interoperability*—Are the mechanisms interoperable with some other Java/Jini or non-Java security solution?

- *Extensibility*—Is the architecture closed, or does it have convenient points of control for implementing additional features? The additional features should not, of course, compromise the security of the existing functionality.

When applying security to a particular application, some of these design aspects could be taken as additional requirements. For example, ad hoc networks require solutions which do not rely on a centralized on-line party.

Chapter 4

Design

In this chapter, I propose a security architecture for Jini which implements some of the requirements identified in Chapter 3. I have also implemented this architecture; the implementation is described in Chapter 5.

The rest of this chapter is organized as follows. First, Section 4.1 gives an overview of the assumptions behind the design. Section 4.2 describes the parties and components involved, and Section 4.3 discusses the trust relationships involved. The components on the client and server sides are described in Sections 4.4 and 4.5, respectively. Finally, Section 4.6 gives a concrete example how the services provided by the architecture are used to construct a secure Jini service.

4.1 Design assumptions

When designing the system, the target environments in mind were wireless ad hoc networks. For example, such a network might consist of a number of PDAs communicating with Bluetooth. An ad hoc environment places many restrictions on the design, and some of these, as well as other goals in the design, can be described as follows.

- The Jini lookup server need not and is not secured in any way. In ad hoc networks, we cannot really assume any on-line trusted servers.
- The architecture should avoid trusting centralized authorities as much as possible.
- The solution should allow application-specific means of implementing the authentication user interface. That is, it should not be limited to, e.g., name certificates.
- Trust statements about code should not rely on claims about the author of the code. Many existing code signature schemes certify only that the author of particular piece

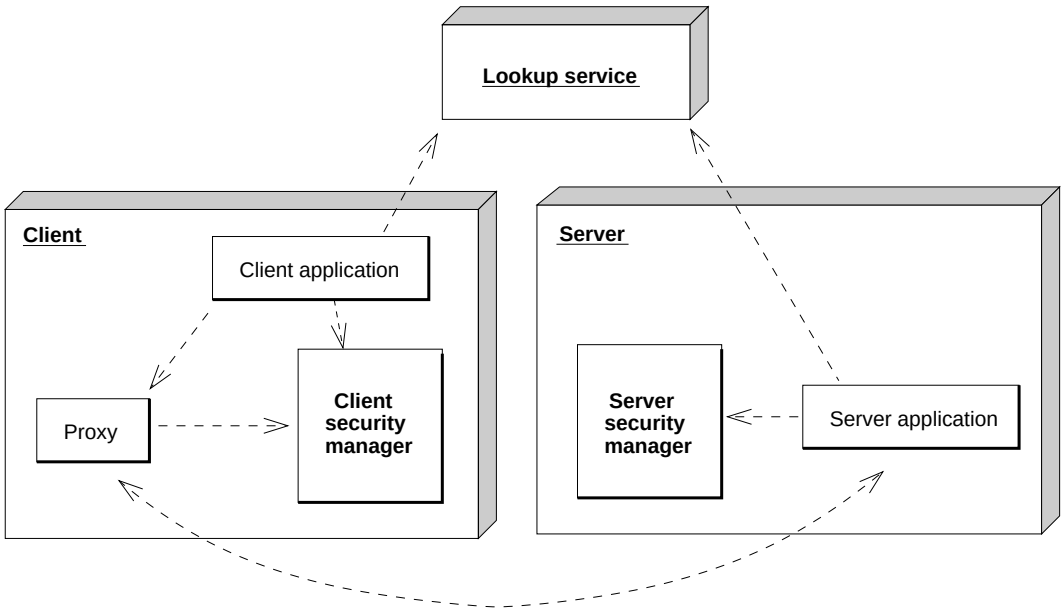

Figure 4.1: The parties and components involved in the architecture.

of software is thought to be “trustworthy”; unfortunately, that approach deals more with assigning blame afterwards than preventing things from going wrong in the first place.

- On the client side, the solution should have flexible mechanisms for authorizing and controlling access to the security mechanisms. That is, it should be possible to have partially trusted applications.
- The solution should allow the use of existing services without modification, at least when the applications do not use any security features.
- The implementation should not require modifications to the Java Virtual Machine, the standard Java class libraries, or the standard Jini code supplied by Sun.

I recognize that many of the goals are quite different from what would be natural to assume if designing a security architecture for, e.g., more traditional middleware applications. However, it is my understanding that these goals are “more difficult”, in a sense that a solution fulfilling these goals is flexible enough to be applied in many different environments.

4.2 Overall architecture

The parties and components involved in the architecture are shown in Figure 4.1. The "three-dimensional" boxes show boundaries between different nodes (hosts), and arrows indicate communication between the components.

On the client side, client applications search for services in the Jini lookup service, and use them through the downloaded proxies. The proxies communicate with the servers using some communications protocol, implemented by each proxy. Both the applications and the proxies can use the services provided by a trusted component, the client security manager. These services are described in Section 4.4.

On the server side, the server application registers its proxy in the lookup service. It uses the services of the server security manager, which are described in Section 4.5.

In addition to the essential security services described above, the actual implementation also provides some commonly needed utility components. These modules are not an essential part of the architecture, and could be easily implemented by each application. However, this would lead to duplication of code, and make the library harder to use. Since the utility components are not part of the architecture, they are described only in the next chapter, in Sections 5.6 and 5.10.

4.3 Trust relationships

Users and services are identified by public keys, as usually in trust management systems. These public keys do not have any centralized certification infrastructure; anyone can start a service and create a new key pair for it. Applications can use name certificates, for instance, to implement an authentication user interface, but these names are not used by the rest of the security system.

The proxies, i.e., both the code and the state of the proxies, are signed by the service the proxy represents. Since the service's public key is not necessarily certified by any trusted third party, this does not guarantee that the proxy is "well behaved", but only that the key used to sign the proxy trusts it. This is actually one of the key assumptions in the design: if a service signs a proxy, it is assumed that the proxy is not trying to subvert the security of that particular service. It might, of course, be quite hostile a client to some other service.

The clients use SPKI certificate chains to prove authorization to use a service. These certificates are generated and distributed using some application-specific means. When actually accessing a service, the chain is completed to a loop using some kind of an authentication protocol. The service then verifies the certificate chain before performing the requested operation. A concrete example of the process is presented in Section 4.6.

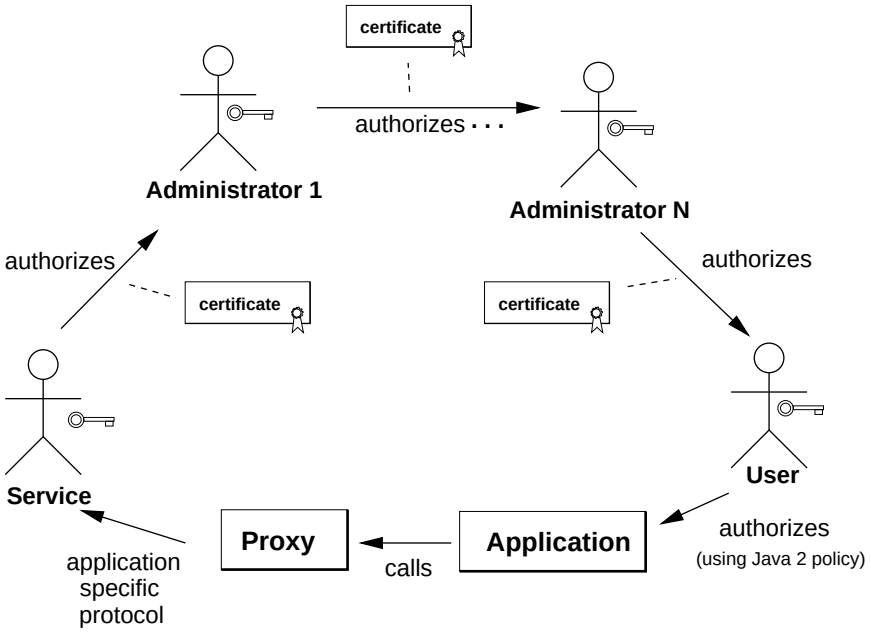

Figure 4.2: Authorization “loop”

On the client side, there can be multiple applications, none of which are fully trusted. The user can specify which permissions a particular application has, i.e., which actions the application itself and the proxies called by the application are allowed to take using the user’s credentials. For instance, a word processing application can be allowed to use the client’s credentials to access printers but not any other services. These restrictions are specified using the Java 2 security policy mechanisms, and they are enforced by a trusted piece of code called the client security manager.

The trust relationships are illustrated in Figures 4.2 and 4.3. The relationships form two loops. The service wants to verify that the user is authorized to use this particular service, and the user wishes to verify that the client is talking to the correct service.

4.4 Client side

When a proxy is downloaded to a client, the client security manager asks the proxy which service it represents, i.e., for the public key of the service, and then checks that the proxy was actually signed by this key. The signatures are verified after the proxy has been instantiated. After the verification, a new key pair is generated for the proxy.

The client security manager provides two services for the proxy. First, the proxy can ask the

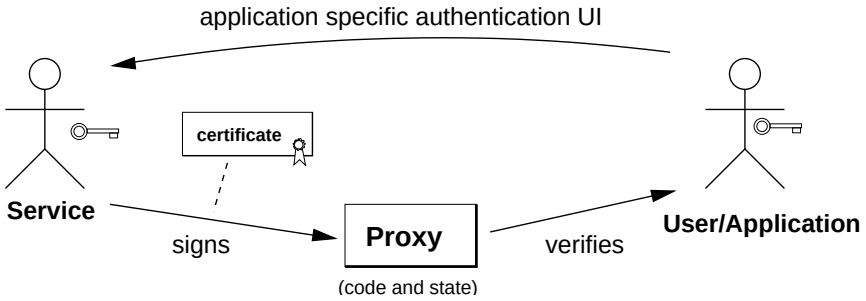

Figure 4.3: Authentication “loop”

security system to sign any piece of data using the proxy’s key (the private key is not given to the proxy). Second, the proxy can request some permissions to be delegated from the user to the proxy’s public key (remember that the security manager holds the user’s private key). The actual permission delegated is the intersection of the arbitrary permission the proxy requested, and the permissions the client application who called the proxy has. This delegation is expressed as a SPKI certificate and the certificate is given to the proxy. The certificate also contains the identity of the service which signed the proxy (a hash of the service’s public key) to ensure that the proxy cannot use the certificate to access some other service. If the security manager has some other certificates which might be relevant, they are also returned to the proxy.

The security manager also provides one service for the client applications. Given a proxy instance, a client application can ask for the public key of the corresponding service (i.e., the key which was used to sign the proxy). The application can then implement an authentication user interface, which could use, for instance, name certificates given by the proxy for authentication.

Figure 4.4 shows a summary of the services provided.

4.5 Server side

The proxy can use any protocol it wishes to contact the server. The only restriction is that it must be able to prove the possession of its private key using the interfaces provided by the client security manager, i.e., public key signatures. The proxy then passes the client’s request and certificates to the service using some application specific protocol. The service application then passes the public key, the certificates, and the request to the security library, which checks if the credentials actually permit the requested action.

The security library also provides a service for signing the proxy before it is registered in the

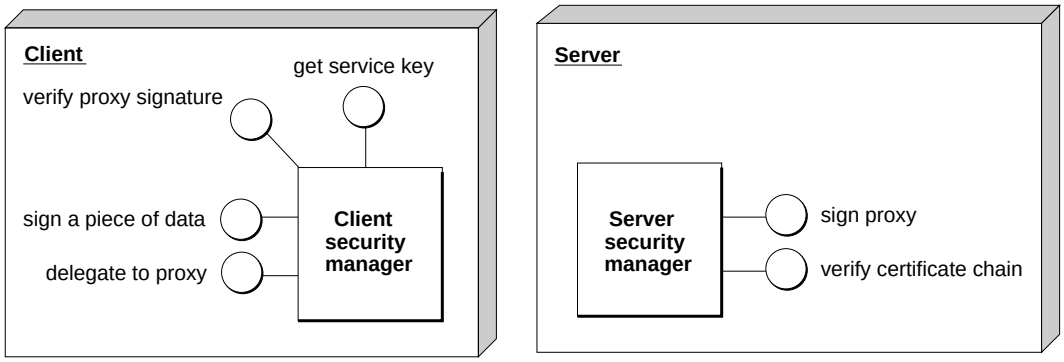

Figure 4.4: Interfaces

lookup service. The services provided by the server side security module are also shown in Figure 4.4.

4.6 An example

This section presents a concrete example to illustrate how the services provided by the security architecture are used to construct a secure Jini service. The normal behavior without any security features is described first, and then the additional steps taken in this architecture are presented.

4.6.1 Normal Jini behavior

The default behavior of a Jini client application and a service is shown in Figure 4.5. In the presented example an application prints a document.

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

1. The application, wishing to use a Jini printing service, contacts the lookup service, and performs an appropriate lookup (in this 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 byte code is downloaded.

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 to print a document.

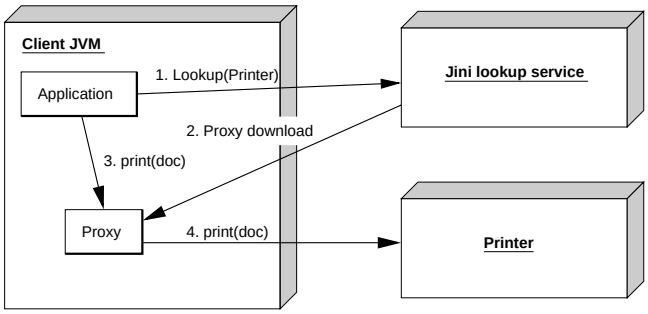

Figure 4.5: Accessing a Jini service, without any security features.

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.

4.6.2 Using the security architecture

When security is applied to a typical Jini scenario, a number of additional steps are needed. The steps taken when accessing a service in our example are described below, and are illustrated in Figure 4.6.

At some previous time, the service has delegated appropriate access rights to the user. The relevant delegation certificates are stored either at the server or at the client.

0. At some previous time, the service has signed the proxy using its private key, and registered the proxy to the lookup service.

1. The application, wishing to use a Jini printing service, contacts the lookup service, and performs an appropriate lookup (in this example, searching for printer services). A list of is returned to the application. No special security features are assumed here.

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 byte code is downloaded (again, using standard Jini facilities).

3. The client security manager asks the proxy for the public key of the service, and verifies that the proxy was actually signed by this key.

4. Next, the application asks the client security manager for the public key of the service. It might then use name certificates, for instance, to verify that the name of the printer shown to the user is correct.

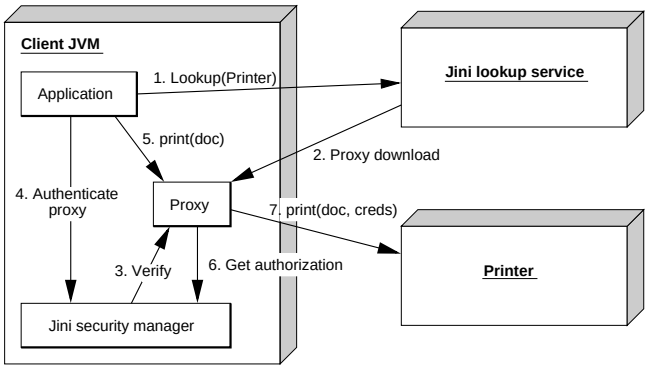

Figure 4.6: Protocol overview with our security modifications.

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

6. The proxy then asks the client security manager for authorization. The 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.

   The security manager then delegates the appropriate permissions to the public key of the proxy. 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. Next, the proxy opens a connection to the server. The proxy can implement any protocol it chooses, as long as it can authenticate itself using the interfaces provided by the client security manager (i.e., signing a piece of data using the proxy's key).

   After the authentication phase has been completed, the proxy sends the certificates and the service request to the server. The server gives the proxy's public key and the certificates to the server security manager, which then checks the certificate chains. If the authorization is valid, the service performs the requested operation.

The actual implementation of these steps is described in Chapter 5.

4.7 Design consequences

The proposed solution is quite flexible. It allows the proxy to implement any communication protocol it wants. It also allows the proxy to contact multiple hosts to provide the service. There is no requirement that the proxies would be signed by a central trusted party. Also,

the solution is not tied to some particular way of implementing the authentication user interface. Also, even if the authentication user interface fails, i.e., the an attacker successfully masquerades as the real service, the attacker's proxy still cannot access the real service using the user's credentials.

Chapter 5

Implementation

In this chapter, the prototype implementation of the architecture is described. First, Section 5.1 gives an overview of the different modules involved. Section 5.2 gives a brief introduction to the Java 2 protection domains and permission, which is needed for understanding some details of the implementation.

The individual components are described in more detail in Sections 5.3 through 5.10. Finally, Section 5.11 contains a summary of the implementation.

5.1 Overview

The prototype implementation of the architecture described in the previous chapter is written completely in Java, and consists of about 10 000 lines of code. The implementation consists of the following components; the components are also shown in Figure 5.1.

- SPKI certificate library (siesta.security.spki) and trust management engine (siesta.security.authorization) are responsible for encoding and decoding SPKI certificates, and verifying certificate chains. These packages are described in Section 5.3.
- Certificate repository (siesta.security.repository) provides a local repository where authorization certificates are stored, and a certificate gatherer which tries to find complete certificate chains. This package is described in Section 5.4.
- The client security manager (siesta.security.core) has many responsibilities. It verifies the proxy's signatures and provides access to the service's public key to client applications. These functions are described in Sections 5.5 and 5.6.

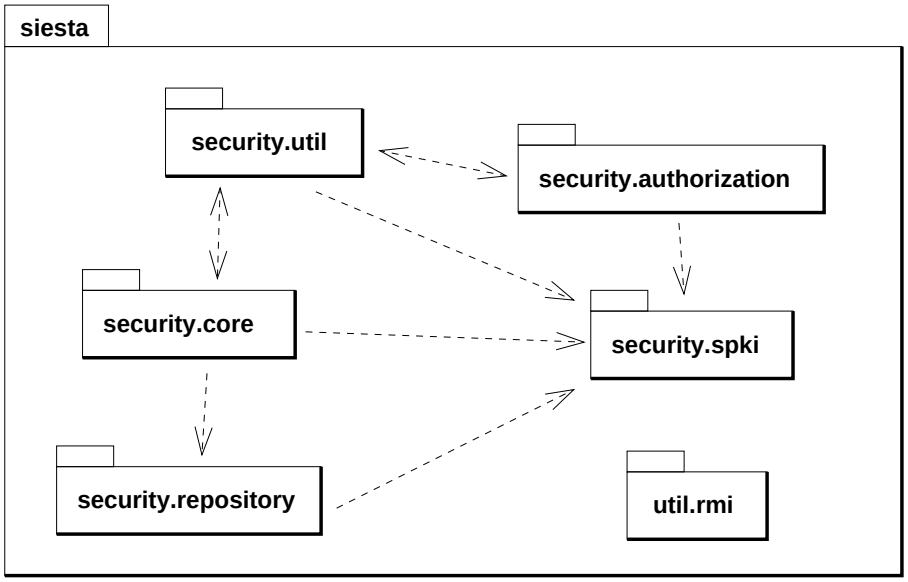

Figure 5.1: UML package diagram, also showing the dependencies between the packages. Some generic utility packages which are not essential for this discussion are not shown.

The client security manager is also responsible for delegating permissions to the proxy, enforcing access restrictions for different applications, and signing data using the proxy's private key, as described in Sections 5.7 through 5.9.

- The library also includes some utility routines which simply writing services and clients (siesta.security.util). Most of these are not described in detail, but the verification of name certificates is mentioned in Section 5.6.
- Another utility module (siesta.util.rmi) implements TLS sockets with mutual authentication for RMI. This component is described in Section 5.10.

A summary of the implementation is presented in Section 5.11.

5.2 Introduction to Java 2 protection domains

Before we can proceed to the implementation, a brief introduction to Java 2 protection domains and permissions is needed for understanding some details of the implementation.

As described in Section 2.3, each piece of code running inside a JVM belongs to exactly one *protection domain*. Each protection domain is associated with a set of permissions.

The permissions are represented as subclasses of the java.security.Permission class. Examples of predefined permissions include the FilePermission and the SocketPermission classes. The permission instances implement the *implies()* method. The semantics are defined so that *x.implies(y)* returns true whenever the set of actions represented by *y* is a subset of actions represented by *x*. For instance, if *x* is *FilePermision(/tmp/*, read)*, and *y* is *FilePermission(/tmp/foo.txt, read)*, *x.implies(y)* returns true.

The permissions associated with a protection domain are stored in a *permission collection*, a subclass of java.security.PermissionCollection. Permissions collections also have an implies() method, with semantics similar to java.security.Permission.

Typically, permissions are checked in some privileged code using the following pattern.

```
SecurityManager sm = System.getSecurityManager();
if (sm != null)
    sm.checkPermission(new SomePermission(...));
// do sensitive operation
```

The usual behavior of the SecurityManager is to call the AccessController.checkPermission() method, which checks the protection domains in the call stack. Eventually this results in calls to the implies() methods of some permission collections. If the action is not allowed, AccessController throws an AccessControlException. For a more detailed explanation, see [36].

5.3 SPKI certificate library and trust management engine

In the implementation there are two separate packages related to SPKI certificates. The first package is a Java Cryptographic Architecture (JCA) “provider” which provides standard Java CertificateFactory functionality, i.e., a factory for converting byte streams to Certificate instances. The certificate instances support verification of signatures and have methods for accessing data inside the certificates. The package also provides a custom interface for signing new certificates.

The SPKI encoding/decoding package is used by the SPKI trust management engine, which is responsible for verifying certificate chains. When a server has received a request from its proxy and verified the proxy’s key, it gives the key and the certificates to the certificate chain verifier module. The verifier then verifies certificate validity and signatures, and finds all certificate chains from the service key to the proxy key. The algorithm used for combining the certificates to chains is currently quite simple—essentially a depth-first search. It is efficient enough if the chains are not very long and if they do not include many unnecessary certificates. More efficient algorithms are described in [4, 5, 24, 60].

The certificate chains found are stored inside a java.security.PermissionCollection instance. 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.getSecurity-Manager().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.

One of the real virtues of this approach is that it also allows communicating these permissions to code which does not know the original call was a remote call, or does not know anything about Jini at all.

5.4 Certificate repository

The verification of certificate chain is closely related to retrieving the necessary certificates. The certificates can be stored on either the client or the server, they could be retrieved on-demand from some directory service, such as DNS or LDAP [41, 60].

The current implementation of the certificate repository supports storing of certificates in a file on the client. Retrieval is based on the client and service public keys, and the attempted action tag. However, the implementation has interfaces which could be used to provide on-demand certificate retrieval.

5.5 Proxy authentication

As described in Section 4.4, both the code and data of proxies are signed by the service. Java already provides facilities for code signing, but the signature itself does not have an expiration date. 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. To ensure that the correct version of the proxy code is always used, a small modification was necessary.

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 [59].

An alternative approach splits the signature to two parts. A new, temporary key pair is generated, and the JAR file is signed using this key and the standard Java code signing

facilities. The private half of the key 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:

- After the signature expires, the JAR file does not have to be signed again if it has not been modified. Since the JAR files are typically stored on a web server, the service might not be able to easily modify them.
- No modifications are needed to the JAR file loading code.
- Existing JDK tools can be used for signing.

The main drawback is that the signature expiration date is not visible to the standard Java components.

In addition to verifying the authenticity of the proxy code, it is necessary to verify the state of 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 do not 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, and getting hold of the serialized object before it is deserialized is tricky.

The problem was solved by asking the proxy object to calculate its own message digest, after it has been deserialized. The proxy byte code has been verified in this point, so the proxy is not completely untrusted, and it is not 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, such as zero.

5.6 Authentication user interface

After we have verified the signatures of the proxy's code and state, we know which service the proxy represents, as identified by the service's public key. We still do not know if this key actually belongs the service the user wants to access.

The application developer has to implement some kind of authentication user interface. The implementation provides supports for verifying two common cases of human readable properties: names signed by some central authority, and ownership of services (such as "John's calendar" service).

To give a simple example, an application could ask the proxy for a name certificate issued by some mutually trusted party. After verifying the certificate, the application could display

```
(sequence
 (cert
   (issuer (public-key (user's public key))
   (subject (public-key (proxy's public key)))
   (tag siesta.user foo.FooPermission #hash of service's public key# read)
   (not-after 2001-01-19_12:33:07))
 (signature ...)))
```

Figure 5.2: Example of a certificate written by the client security manager. The certificate delegates permissions from the user to the proxy.

the name on the screen and ask the user if this was indeed the service he or she intended to access. Naturally, more user friendly approaches are possible.

5.7 Proxy authorization

When accessing a service, the proxy asks the client security manager to delegate the required access permissions to the public key of the proxy. The proxy passes the requested "tag" field. As described in Section 4.4, the actual permission delegated is the intersection of the requested permission and the permissions of the calling application.

Before signing the certificate, the tag field is verified in two ways. First, to ensure that the certificate can only be used by the service that the proxy represents, the hash of the public key of the service is stored in a fixed position in the tag field.

Second, the local security policy is consulted to check that the application which is calling the proxy is authorized to act with the user's credentials. This is described in the next section.

An example of a certificate written by the client security manager is shown in Figure 5.2.

5.8 Protecting the security mechanisms

Since applications on the client side are not fully trusted, mechanisms are needed to specify local security policy, and enforce the restrictions.

The Java 2 security architecture offers such mechanisms, and indeed they work well on the server side. For instance, the service software can define a permission type of its own, say, PrintPermission, and delegate these permissions using SPKI certificates. These permission cannot, however, be used as such on the client side, because the client does not necessarily

have the implementation (i.e., the byte code) for the permission. Without the implementation, it is not possible to instantiate the permission. Furthermore, the implementation has to be trusted to work properly.

This is solved by specifying the security policy in the Java 2 security policy using a special permission class *RemotePermission*. The remote permission contains a regular expression which is used to match the tag requested by the proxy.

When the proxy calls the security manager and asks for delegation, the security manager constructs a RemotePermission instance corresponding to the tag given by the proxy, and uses the SecurityManager's checkPermission method to check authorization.

The only case requiring special treatment is the proxy itself. The proxy must have a permission to access the service it represents since otherwise the security check made by the SecurityManager.checkPermission method would fail. However, when the proxy's class is loaded and the permissions granted, the proxy is not yet fully authenticated, and the public key of the service is not known (as explained in Section 5.5, a different key is used for signing the JAR file). This problem is solved by having RemotePermission instances contact the client security manager when the permission is checked, since the client security manager knows which proxies are properly authenticated.

5.9 Implementing authentication protocols

After some permissions have been delegated to the proxy, it usually contacts the service to actually perform the requested action. To implement an authentication and key exchange protocol, the proxy can ask the client security manager to sign arbitrary pieces of data, using the proxy's private key.

The private key itself is not given to the proxy, so it cannot send the key material to some third party, for instance. Additionally, this allows the use of the proxy key to be securely logged. To accommodate interfaces which require a java.security.PrivateKey, such as JSSE's X509KeyManager, the proxy is given a "guarded object" which implements the PrivateKey interface. The guarded key provides the same methods as the original key. Before a method call is forwarded to the actual private key, the guarded key checks that the call is coming from a trusted signature implementation. This is implemented using Java stack inspection mechanisms. However, the proxy can, of course, act as a signing oracle to a third party.

5.10 Optional transport protocol (RMI over TLS)

Many proxies probably will not want to implement a custom protocol, and therefore the implementation provides utilities for using RMI. The default RMI configuration uses normal TCP sockets, but it is possible to override this behavior by supplying a pair of *socket factories* to be used on the server and client sides of the communication. This facility is meant for plugging in Transport Layer Security (TLS) sockets [23].

The implementation has socket factories for TLS client authentication using the Java Security Socket Extension (JSSE) libraries [72]. During the implementation some slight deficiencies were found in the current RMI implementation. Hopefully most of these will be fixed in the next release of RMI and in the RMI security API [71].

5.10.1 Problems with client authentication

Although the socket factory interfaces were originally intended for plugging in TLS sockets, the design supports only server authentication cleanly. 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 have 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. The socket, in the case of TLS, would contain methods to get the client's key.

The current implementation works (sort of) around these problems 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 [11].

5.10.2 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 byte code 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.

However, if a class is loaded using a java.net.URLClassLoader (or some subclass of it), the URL it was loaded from is used instead of the default codebase. Thus, if the server loads the proxy class using URLClassLoader, and instantiates it using the reflection API, the codebase gets set to the correct value. The use of the reflection API is needed to avoid loading multiple copies of the same class.

5.11 Summary of implementation

The key services offered by the architecture were described in Chapter 4. The client side security manager provides the following services.

- *Verify the signatures of a proxy*—Both the code and data (state) of the proxy are protected using digital signatures, which are verified when the proxy is loaded, as described in Section 5.5.
- *Given a proxy instance, get the corresponding service key*—The application must implement an authentication user interface, which maps the public key of the service to a name, for instance, as outlined in Section 5.6.
- *Delegate permissions to the proxy*—When a properly authorized application calls a proxy, the proxy can ask for credentials which allow it to access the service it came from using the user's permissions. This functionality was described in Sections 5.7 and 5.8.
- *Sign a piece of data using the proxy's key*—To securely use the credentials the proxy has received, it can implement an authentication and key agreement protocol using digital signatures, as described in Section 5.9. A sample implementation which uses the JSSE libraries was described in Section 5.10.

On the server side, the server security manager offers the the following services.

- *Sign proxies*—Proxies must be instantiated on the server and signed before they are registered in the lookup service.
- *Verify certificate chains*—When a client attempts to access the service, the service verifies that the credentials provided indeed allow the requested action, as described in Section 5.3.

Chapter 6

Evaluation

This chapter evaluates the architecture and the implementation based on the evaluation criteria first presented in Section 1.3. For easy access, the evaluation criteria are restated in Section 6.1, and the architecture is evaluated based on these criteria in Section 6.2.

The focus is on the evaluation of the proposed architecture; however, the prototype implementation and its performance are also discussed briefly in Sections 6.3 and 6.4. Finally, the architecture is compared with related work in Section 6.5.

6.1 Evaluation criteria

In Chapter 1 the following criteria were chosen for evaluating the proposed architecture.

- *Security functionality*—What security features does the solution provide? Does it have convenient points of control for implementing additional features later? The required features were further elaborated on in Chapter 3.
- *Minimized trust relationships*—What kind of trust relationships does the solution assume? Are the assumptions flexible enough so that the solution can be applied in various environments?
- *Protocol independence*—How well does the solution preserve Jini's protocol independence, and the flexibility it brings? In particular, does the solution restrict the ability to implement part of the functionality in the proxies, or does it require some specific wire protocol or authentication mechanism?
- *Elegance*—How elegant is the solution? That is, is it easy to understand, technologically justified, state of the art, etc.?

- *Simplicity*—How transparent the solution is to applications, and how easy it is to use? Does the solution allow separation of security related code from the application code?

In the next section the architecture is evaluated based on these criteria.

6.2 Architecture

6.2.1 Security functionality

Chapter 3 identified a number of functional security requirements. These requirements are included in the architecture as follows.

1. *Authentication user interface*—The architecture itself does not use any properties associated with the public keys, and does not impose entities, such as names, which might not be relevant in all situations. This leaves applications free to implement the authentication user interface.

   The authentication user interface is consulted only after the proxy has been instantiated. In some environments, authenticating the proxy before instantiation might be a better solution if, for instance, denial of service attacks are a concern. However, this would make it more difficult to use credentials provided by the proxy in the authentication decision.

2. *Authentication of proxy identity/credentials*—Both the code and data of a proxy are authenticated. The authentication of data is done only after instantiation, mostly to avoid modifications to the standard libraries.

3. *Authentication and key agreement protocol*—The architecture supports any authentication and key agreement protocol which is based on public key signatures. The proxy can implement the protocol itself or use some existing libraries, such as JSSE.

4–5. *Message and connection integrity; message encryption*—The architecture does not mandate any specific protocol; the proxy can either implement the necessary components itself, or use existing third party libraries.

6. *Authentication of client identity/credentials (through the proxy)*—The client is authenticated indirectly through certificates which delegate permissions to the proxy's key. To guarantee freshness the proxy can include into the certificates nonces given by the server.

7. *Local security policy*—The architecture uses the standard Java 2 mechanisms for specifying access policies to the client JVM resources. To allow more flexible policies the architecture could be combined with something like [59].

   The server side policy for controlling remote access to a service is essentially only an ACL with one line, specifying the public key of the service. The rest of the security policy is specified using SPKI certificates.

   Restrictions for what kinds of remote calls client applications are allowed to make using the user's credentials are also specified using the Java 2 policy mechanisms.

8. *Access control mechanisms for local resources*—Existing Java 2 mechanisms are used to provide access control for security mechanisms, i.e., the client security manager.

   The private key of the proxy is guarded using a custom stack inspection mechanism, as described in Section 5.9.

9. *Resource consumption control*—The architecture does not implement any resource consumption control mechanisms.

6.2.2 Minimized trust relationships

The key assumption behind the architecture is the following: if a service signs a proxy, it is assumed that the proxy is not trying to subvert the security of that particular service. It might, of course, be quite hostile a client to some other service.

The architecture does not introduce any centralized trusted third parties. In particular, proxies are signed directly by the service, and no certificate hierarchies are required. Also, the lookup service does not have to be trusted to contain only "friendly" services.

The problem of verifying that the public key of the service actually is the right service is left to the application. Thus the application is free to use the most appropriate method for the particular situation. In some cases, this might involve, for instance, a hierarchical PKI, but other solutions are possible.

6.2.3 Protocol independence

The architecture does not mandate any specific communication protocol between the proxy and the service. There are two limitations, however, related to the authentication of the client. First, the authentication protocol must be based on public key signatures. Many existing protocols, such as TLS and ISAKMP [55], support this, but some protocols are based on public key encryption or symmetric key cryptography instead.

Furthermore, since the client is authenticated only indirectly through the proxy, the server has to understand the SPKI certificates which delegate authority from the client to the proxy. Thus the solution is not, unfortunately, interoperable with existing systems.

6.2.4 Elegance

I feel that some features of the architecture are quite elegant. For instance, since the architecture does not include any concept of names, it can be used in many circumstances where names do not naturally occur. Yet, the application can easily use name certificates if the situation so requires.

Another aspect worth mentioning is independence from the actual wire protocol. From an architectural point of view, this also means a large reduction in the amount of code that absolutely has to be trusted.

On the server side, integration of the permission checking with the standard Java 2 security architecture allows the use of same mechanisms for checking permissions as for non-remote calls. On the other hand, the service application is responsible for checking these permissions. In some environments, separations of the application and service code, as done in, for instance, Enterprise JavaBeans, might be a better approach.

6.2.5 Simplicity

Even though the number of interfaces provided for proxies and client applications is quite small, some aspects of the architecture cannot be considered to be simple.

In my experience, many people without previous experience in SPKI certificates, or trust management systems in general, find the concept of delegation chains difficult to understand. Thus, the architecture might to considered too complex to use, or used incorrectly due to some misunderstanding.

The Java 2 security architecture, with many components such class loaders, protection domains, code sources, and the access controller, is not very simple to understand either. Thus, the aspects of the solution that rely on it are also quite complex.

However, other proposals, such as the forthcoming RMI security API [71], seem even more complex and harder to understand.

The proposed architecture also leaves a lot of freedom to the service developer. This naturally gives flexibility, but in cases where such flexibility is not required, it certainly introduces unnecessary complexity. For instance, leaving the implementation of the wire protocol to the developer might cause an unnecessary complexity in many cases. Furthermore, as discussed above, the architecture does not support the separation of the application and the security related code very well.

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

Table 6.1: 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.

6.3 Implementation

The implementation is intended to be a research prototype, and therefore is not fully polished. Some aspects which would require further work are mentioned next.

First of all, the client side architecture does not integrate with the Java 2 security model as well as the server side components. As described in Section 5.8, regular expressions are used for matching permissions given to applications, instead of the normal implies() methods.

Another inelegant aspect is that the delegation bit of SPKI certificates is not used strictly according to the SPKI specification. The user must be able to delegate to the proxy's key even if delegation was not allowed in the certificate given to the user. This might be worked around by including a "nonce" in the tag field, proving that the certificate was written recently.

The SPKI trust management components is also missing some other features. The implementation does not support fetching certificates from remote repositories. In many real applications, at least some mechanism for distributed the necessary certificates is required. Another important aspect which requires further work is the integration of a certificate revocation or validation mechanism. Revocation and validation of SPKI certificates are discussed in [49], for instance.

On the positive side, the implementation contains no modifications to standard Java or Jini libraries. However, two implementation aspects are relied upon. First, the getCallContext method from java.lang.SecurityManager is used to inspect the call stack when guarding access to the proxy's private key. The same result could have been achieved by modifying the standard libraries. Second, the RMI over TLS implementation communicates keys using thread local variables, and this relies on certain assumptions about how threads are allocated and connections opened inside the RMI libraries.

6.4 Performance

Table 6.1 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.

The measurements were run using Sun's JDK 1.2.2 under Red Hat Linux 6.2. 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.

These performance figures are not meant to be comprehensive, but they illustrate that the performance would be acceptable for real world applications. Possible targets for optimization include both the security part itself and RMI in general [17, 50].

6.5 Related work and comparison

The security of Jini systems has not been studied much yet. Some of the results from other object-oriented middleware, such as the CORBA security framework, are useful, but Jini is fundamentally different due to the extensive use of downloaded code.

Sun presented a demonstration solution which integrates Jini with JAAS at JavaOne 2000 [68]. 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 [18].

Hasselmeyer et al. have developed a Jini security solution based on a centralized secure lookup server [39]. The lookup service authenticates all clients and services, and vice versa. Proxies returned by the lookup service are assumed to be "well behaved" and not misuse the client's private credentials when authenticating the connection between the client and the service. The authors have also built an infrastructure for associating payments with Jini leases based on this architecture. Their work on leases, which focuses on the non-repudiation aspects, is described in [40].

A commercial implementation of a secure lookup service is available from PersonalGenie Inc. [63]. The whitepaper does not give very much details, but the architecture seems quite similar to the one proposed by Hasselmeyer et al. Similar secure service directory in a non-Jini environment is described by Czerwinski et al. in [21, 79].

Balfanz et al. describe a system based on RMI, though not Java, in [11]. It uses a delegation system resembling SPKI, and also uses SSL for protecting the transmission. An important difference is, however, that their architecture does not allow dynamic downloading of code.

Eurescom's Jini and Friends at Work project is looking into Jini and ubiquitous computing issues from a telecom operator's point of view [31]. The project's first public deliverable [30] also gives an outline of the security issues involved. While the document does not specify any concrete architecture, it proposes a centralized certification infrastructure—to be operated by telecom operators, of course—as a solution to problems associated with downloaded code.

Sun's future solution for Jini security is the RMI security API [71], which is currently under development. It is expected to be included in the next major release of Java 2 Standard Edition (JDK 1.4, codenamed "Merlin"). I was a member of the expert group which reviewed the specification under the rules of Java Community Process [74].

In the RMI security API, a trusted component (not downloaded from the network) is responsible for opening the network connections and implementing some authentication protocol. The API is based on the "service provider interface" model, which means the providers for different authentication technologies can be plugged in. A provider for TLS sockets is expected to be included in JDK 1.4.

Although the RMI security API supports intelligent proxies, which implement a part of their functionality on the client, it could limit the benefits offered by Jini's protocol independence. However, it is important to remember the RMI security API is intended for a larger audience than just Jini, such as application developers using RMI as a middleware components, and also the Java 2 Enterprise Edition. Taking the interests of different audiences into account in the design of the API has been difficult. On one hand, programmers need a flexible API to control security functions. On the other hand, in the Java 2 Enterprise Edition security issues are typically handled by the Enterprise JavaBeans "container", and not the application directly.

Chapter 7

Conclusions

Jini provides an elegant architecture for building distributed applications, especially in decentralized ad hoc environments. Programs built according to the Jini principles will be able to function and survive in highly dynamic network environments, allowing applications to adapt their behavior to the requirements of the current network context. However, the current state of the technology does not adequately address the security requirements present in many of such environments. In particular, the existing solutions are either bound to a specific communication protocol, or rely on centralized security servers.

This thesis has two main components, a requirements analysis and a proposed architecture.

Chapter 3 analyzed the security requirements of Jini in different environments. Threat scenarios were used to motivate the choice of high level security goals, and these high level goals were mapped into lower level security mechanisms. Based on existing security frameworks, some features which were not included were identified. The chapter also presented some related design aspects and tradeoffs.

Chapter 4 proposed one possible security architecture for Jini, based on the decentralized trust management approach. The architecture uses SPKI certificates for authorization, and does not compromise the protocol independence of Jini. As it does not rely on centralized security servers, it is flexible enough to work in a wide range of environments.

The implementation described in Chapter 5 demonstrates that the architecture can be implemented, and does not have unreasonable performance impact. The evaluation of the architecture in Chapter 6 showed that the design of this type of a security architecture is highly non-trivial, and making it elegant and easy to understand is hard.

There are many possibilities for further work, dealing both with the the requirements analysis and the proposed architecture. Some unresolved questions about requirements were mentioned in Section 3.4. The requirements for core Jini services, such as the lookup ser-

vice, leases, events, and transactions, need to be clarified. Also, identifying relevant subsets of reasonable and non-conflicting requirements for different environments is necessary.

Furthermore, it would be beneficial to study how the proposed architecture could be either integrated or used together with other Java security mechanisms, including the forthcoming Java Authentication and Authorization Service (JAAS) [51] and the planned RMI Security Extension [71]. Interoperation with existing security services, such as Kerberos, could also be interesting.

To summarize, this thesis has shown that using centralized security servers or certificate authorities are not the only solutions to problems associated with downloaded code in Jini.

Bibliography

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

[2] Gerd Aschemann, Svetlana Domnitcheva, Peer Hasselmeyer, Roger Kehr, and Andreas Zeidler. A framework for the integration of legacy devices into a Jini management federation. In Rolf Stadler and Burkhard Stiller, editors, *Active Technologies for Network and Service Management: 10th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management proceedings (DSOM '99)*, Lecture Notes in Computer Science volume 1700, pages 257–268, Zürich, Switzerland, October 1999. Springer.

[3] Gerd Aschemann, Roger Kehr, and Andreas Zeidler. A Jini-based gateway architecture for mobile devices. In Clemens H. Cap, editor, *JIT '99: Java-Informations-Tage 1999*, Informatik aktuell series, pages 203–212, Düsseldorf, Germany, September 1999. Springer.

[4] Tuomas Aura. Comparison of graph-search algorithms for authorization verification in delegation networks. In *Proceedings of the 2nd Nordic Workshop on Secure Computer Systems (NordSec '97)*, Espoo, Finland, November 1997.

[5] Tuomas Aura. Fast access control decisions from delegation certificate databases. In Colin Boyd and Edward Dawson, editors, *Proceedings of the 3rd Australasian conference on information security and privacy (ACISP' 98)*, Lecture Notes in Computer Science volume 1438, pages 284–295, Brisbane, Australia, July 1998. Springer.

[6] Tuomas Aura. *Authorization and availability: Aspects of open network security*. Doctoral dissertation, Helsinki University of Technology, November 2000. Laboratory for Theoretical Computer Science research report HUT-TCS-A64.

[7] Tuomas Aura and Pekka Nikander. Stateless connections. In *Proceedings of International Conference on Information and Communications Security (ICICS '97)*, Lecture Notes in Computer Science volume 1334, pages 87–97, Beijing, China, November 1997. Springer.

[8] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In Bruce Christianson, Bruno Crispo, and Mike Roe, editors, *Proceedings of the 8th International Workshop on Security Protocols*, to appear in the Lecture Notes in Computer Science series, Cambridge, UK, April 2000. Springer.

[9] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000. USENIX Association.

[10] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the USENIX Annual 2000 Technical Conference, General Refereed Track*, San Diego, California, June 2000.

[11] 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.

[12] Dirk Balfanz and Li Gong. Experience with securie multi-processing in Java. In *Proceedings the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.

[13] Philippe Bernadat, Dan Lambright, and Franco Travostino. Towards a resource-safe Java for service guarantees in uncooperative environments. In *Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications (PLRTIA '98)*, Madrid, Spain, December 1998.

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

[15] 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.

[16] 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.

[17] Stefano Campadello, Oskari Koskimies, Kimmo Raatikainen, and Heikki Helin. Wireless Java RMI. In *Proceedings of the 4th International Enterprise Distributed Objects*

Computing Conference (EDOC 2000), pages 114–123, Makuhari, Japan, September 2000. IEEE Computer Society Press.

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

[19] Concept Technologies Ltd. Hosta: Java based web server cluster controller. White paper, http://www.concept-technologies.com/pages/Hosta/HostaWhitePaper.html, May 2000.

[20] 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.

[21] 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.

[22] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the 4th ACM Conference on Computer and Communications Security (CCS '97)*, pages 18–27, Zürich, Switzerland, April 1997.

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

[24] Jean-Emile Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, May 1998.

[25] Carl Ellison. SPKI requirements. RFC 2692, IETF, September 1999.

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

[27] Pasi Eronen, Christian Gehrmann, and Pekka Nikander. Securing ad hoc Jini services. In *Proceedings of the 5th Nordic Workshop on Secure IT Systems (NordSec 2000)*, pages 169–177, Reykjavik, Iceland, October 2000. Reykjavik University.

[28] 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.

[29] Pasi Eronen and Pekka Nikander. Decentralized Jini security. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2001)*, pages 161–172, San Diego, California, February 2001.

[30] Eurescom. Jini and friends at work project, deliverable 1: Jini state of the art: an operator's perspective. http://www.eurescom.de/Public/Projects/P1000-series/P1005/P1005.htm, July 2000.

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

[32] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th National Information Systems Security Conference (NISSC 96)*, pages 591–597, 1996.

[33] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, June 1999.

[34] 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.

[35] Dieter Gollmann. *Computer Security*. John Wiley & Sons, February 1999.

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

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

[38] Jaap Haartsen, Mahmoud Nagshineh, Jon Inouye, Olaf J. Joeressen, and Warren Allen. Bluetooth: Vision, goals, and architecture. *Mobile Computing and Communications Review*, 2(4):38–45, October 1998.

[39] 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.

[40] Peer Hasselmeyer, Markus Schumacher, and Marco Voß. Pay as you go — associating costs with Jini leases. In *Proceedings of the 4th International Enterprise Distributed Objects Computing Conference (EDOC 2000)*, pages 48–57, Makuhari, Japan, September 2000. IEEE Computer Society Press.

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

[42] Manfred Hauswirth, Clemens Kerer, and Roman Kurmanowytsch. A secure execution framework for Java. In *Proceedings of the 7th ACM conference on computer and communications security (CCS 2000)*, pages 43–52, Athens, Greece, November 2000.

[43] ISO/IEC 8802-11:1999: Information technology — Telecommunications and information exchange between systems — Local and metropolitan area networks — Specific requirements. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications, August 1999. Also ANSI/IEEE standard 802.11 (1999 edition).

[44] ITU-T Recommendation X.509 (1997): Information technology — Open systems interconnection — The directory: Authentication framework, June 1997. Also ISO/IEC 9594-8:1998.

[45] ITU-T Recommendation X.800 (1991): Security architecture for Open Systems Interconnection for CCITT applications, 1991.

[46] ITU-T Recommendation X.810 (1995): Information technology — Open systems Interconnection — Security frameworks for open systems: Overview, November 1995. Also ISO/IEC 10181-1:1996.

[47] 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.

[48] Arto Karila. *Open Systems Security: An Architectural Framework*. Doctoral dissertation, Helsinki University of Technology, March 1991.

[49] 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.

[50] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementation of Java remote method invocation (RMI). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 19–35, Santa Fe, New Mexico, April 1998.

[51] 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.

[52] Ilari Lehti and Pekka Nikander. Certifying trust. In *Public Key Cryptography: the 1st International Workshop on Practice and Theory in Public Key Cryptography, proceedings (PKC '98)*, Lecture Notes in Computer Science volume 1431, Yokohama, Japan, February 1998. Springer.

[53] Jussipekka Leiwo, Pekka Nikander, and Tuomas Aura. Towards network denial of service resistant protocols. In *Proceedings of the 15th International Information Security Conference (IFIP/SEC 2000)*, Beijing, China, August 2000. Kluwer.

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

[55] Douglas Maughan, Mark Schneider, Mark Schertler, and Jeff Turner. Internet security association and key management protocol (ISAKMP). RFC 2408, IETF, November 1998.

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

[57] Pekka Nikander. *An Architecture for Authorization and Delegation in Distributed Object-Oriented Agent Systems*. Doctoral dissertation, Helsinki University of Technology, March 1999.

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

[59] 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.

[60] Pekka Nikander and Lea Viljanen. Storing and retrieving Internet certificates. In *Proceedings of the 3rd Nordic Workshop on Secure IT Systems (NordSec '98)*, Trondheim, Norway, November 1998.

[61] Object Management Group. *CORBAservices Security Service Specification*. Version 1.5 (document 00-06-25), May 2000.

[62] Jonna Partanen. Using SPKI certificates for access control in Java 1.2. Master's thesis, Helsinki University of Technology, August 1998.

[63] Personal Genie Inc. GuardianGenie secure lookup service white paper. Available from http://www.personalgenie.com/, September 2000.

[64] Ronald L. Rivest and Butler Lampson. SDSI — a simple distributed security infrastructure, version 1.1. Manuscript, available from http://theory.lcs.mit.edu/~cis/sdsi.html, October 1996.

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

[66] Bob Scheifler. Personal communications, 2000.

[67] 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.

[68] 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.

[69] Sun Microsystems. Java remote method invocation specification. Revision 1.7 (Java 2 SDK Standard Edition 1.3.0), December 1999. http://java.sun.com/products/jdk/rmi/.

[70] Sun Microsystems. Enterprise JavaBeans Technology home page. http://java.sun.com/products/ejb/, 2000.

[71] Sun Microsystems. Java RMI security API. Technical specification, JSR 76 community draft version, December 2000.

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

[73] Sun Microsystems. Java security FAQ: Chronology of security-related bugs and issues. http://java.sun.com/sfaq/chronology.html, August 2000.

[74] Sun Microsystems. Java specification request 76: RMI security. http://java.sun.com/aboutJava/communityprocess/jsr/jsr_076_rmisecurity.html, 2000.

[75] Universal Plug and Play Forum. Home page. http://www.upnp.org/, 2000.

[76] 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. Reprinted in [1].

[77] 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.

[78] 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.

[79] Matt Welsh. Ninja RMI: A free Java RMI. http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html, September 1999.

[80] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *USENIX Computing Systems*, 9(4):265–290, 1996.

[81] Ann Wollrath, Jim Waldo, and Roger Riggs. Java-centric distributed computing. *IEEE Micro*, 17(3):44–53, May 1997.