Trade-offs in a Secure Jini Service Architecture

Peer Hasselmeyer, Roger Kehr, and Marco Voß

Department of Computer Science,
Darmstadt University of Technology

{peer,mavoss}@ito.tu-darmstadt.de
kehr@informatik.tu-darmstadt.de

Abstract. Jini is an infrastructure built on top of the mobile code facilities of the Java programming language enabling clients and services to spontaneously engage in arbitrary usage scenarios. For a small home or office environment the currently available infrastructure might be adequate, but for mission-critical applications it lacks essential security properties. In the sequel we identify weak points in the Jini architecture and its protocols and propose an extension to the architecture that provides a solution to the identified security problems. We describe the design choices underlying our implementation which aims at maximum compatibility with the existing Jini specifications.

1 Introduction

The Jini connection technology [Wal99,Sun99b] is an innovative and usable technology for building reliable, fault-tolerant distributed applications. It provides an infrastructure that allows clients to find services independent of both party's location. The dynamic nature of locating and using services is one of Jini's major strengths. It is the base for the creation of plug and play devices and services. This works well in one's own home, but already in a small workgroup some problems can arise. While it is usually alright for everybody to access your printer, most people do not want everybody that can access their wireless LAN to take a peek at their latest project data.

This problem becomes even more serious if one wants to use services via an open network like the Internet. Suffice it to say that you want to be sure to give your credit card number only to your favorite online store and not somebody else. Unfortunately, this area is currently untouched by Jini. There are no provisions for data encryption or authentication beyond the abilities of Java 2 and RMI.

The research described in this paper identifies the weak points in the Jini architecture and proposes an extension to the architecture which enables secure lookup of services and trust establishment. The main security concern within the Jini architecture is the use of dynamically downloaded proxies. These provide great flexibility but present a security risk as the client does not know what the code of the proxy is doing. The client can safeguard itself against security breaches with regard to local resources like hard drives or even network connections by supplying a strict security policy. But it has no way of determining what a proxy is doing with supplied data like a credit card number.

©Springer-Verlag

Copyright Springer Verlag, USM'2000, September 12-14, Munich, Germany

The paper describes how this problem can be addressed by requiring all parties involved in a Jini federation (services and clients) to mutually authenticate themselves. Furthermore, we introduce the notion of *secure groups* to restrict the visibility of services registered at lookup services and to ease administration of access rights.

Section 2 briefly introduces the Jini connection technology and describes how clients find services and how they interact. Section 3 describes the security properties that we believe to be required in typical scenarios of a future Jini services world. In Section 4 we introduce our extension to the Jini architecture. Our implementation of this extension is described in Section 5. Section 6 discusses an example flow of communication. Relevant work is evaluated in Section 7 and we finally give an outlook in Section 8 on what else has to be done to enable fully secure Jini federations.

2 Component Interaction in Jini

Jini is a Java Application Programming Interface (API) that implements protocols and mechanisms for service registrations and service lookups centered around the so-called Jini *lookup service* [Sun99d]. Jini services are comprised of two components: the Jini *service provider* running on the network node or device offering a particular service, and the *service proxy*, a Java object fetched by clients from a lookup service and executed in the Java virtual machine (JVM) of a client. Both jointly implement the actual service provided. In the sequel we describe the core interactions between components in a Jini service scenario.

Service Registration. Figure 1a shows the relevant protocols for Jini service registrations. Service providers willing to offer their service to potential clients must first find nearby lookup services by means of multicast request messages [Sun99c] sent to the network. Lookup services are required to answer to these requests by opening a TCP-stream to the port and IP-address contained in the original request.

Via this callback the lookup service sends a serialized Java object that implements the well-defined Java interface *ServiceRegistrar*. This serialized object contains the state of the *lookup service proxy* and the so-called *codebase* which is essentially a URL pointing to a Web-server from where the implementation of the proxy in the form of Java bytecode can be downloaded. This bytecode is loaded into the JVM of the service provider and the serialized proxy object is instantiated. Eventually, the service provider uses the *register*-method of the lookup service proxy API to upload its own service proxy augmented with additional service description information to the lookup service.

Service Lookup. Clients obtain service proxies from the lookup service as depicted in Figure 1b. A client first performs the same steps as a service provider to obtain a lookup service proxy from a lookup service. Next, a client invokes the proxy's *lookup*-method to query the lookup service for services it is interested in.

In response to this invocation the service proxy available at the lookup service is transferred to the client. Before the service proxy is de-serialized, its implementation is downloaded to the JVM of the client by means of the codebase attached to the serialized proxy. After the client has instantiated the service proxy in its JVM it uses the proxy's

©Springer-Verlag

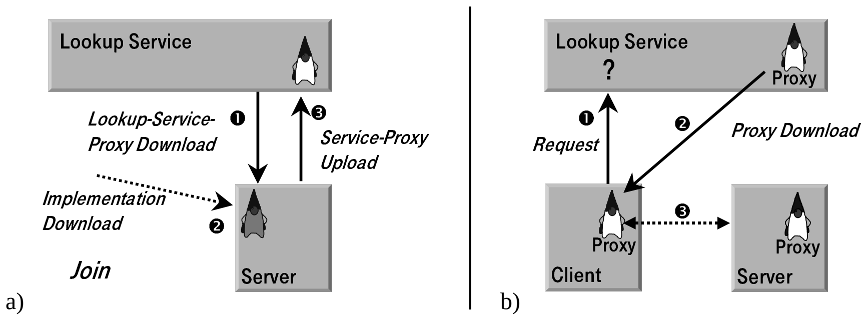

Fig. 1. Jini-Protocols: a) Discovery/Join and b) Lookup

API to invoke methods. It is entirely left open to the implementation of the proxy how it processes these invocations. Some invocations can be completely performed locally in the client's JVM. Others may result in a network communication to the service provider followed by remote computations. Part of the Jini philosophy is not requiring any particular form of communication between a service proxy and its provider. Developers are free to choose any suitable communication channel such as TCP-sockets, Java remote method invocations (RMI), CORBA object invocations, etc.

With the Jini approach, the implementation of a service can be partitioned arbitrarily between the service proxy and the service provider. This feature distinguishes Jini from other comparable service infrastructures. The mobility of Java bytecode is the enabling technology for this approach at the cost of requiring a JVM on both sides, the server and the client.

3 Requirements for Secure Component Interaction

In this section we identify several requirements for a secure component interaction in a Jini environment which have shaped the architecture of our implementation.

Mobile Code Security Issues. If we compare the Jini architecture to "traditional" client-server systems like CORBA or the world-wide Web, we can spot one major difference: in all these systems the client permanently contains the code for communicating with a server. The protocol code is part of the client and therefore part of the client's trusted computing base. If a client needs some kind of security (like authentication or integrity), it can choose to use any protocol that provides the required security properties (e.g. SSL). The Jini approach is fundamentally different. Jini clients do not implement any network protocol at all. They rather rely on the service's proxy object to perform the communication with the server. As mentioned before, proxy objects originate from some (usually untrusted) source on the network. This includes the download and execution of code from that source. Clients do not know what these objects are doing. Studies of the security risks of mobile code (e.g. [RG98]) usually focus on the protection of the execution platform against malicious actions of downloaded code. If we assume that effective protection of the platform can be achieved by the Java sandbox model and

©Springer-Verlag

appropriate security policies [Gon98], we still have a different concern here: a client does not and cannot know what a proxy object is doing with supplied data. A security approach that is different from those of traditional client/server systems is therefore required. Because the proxy is supplied by its associated service it should know which kind of security is appropriate for its application domain. We therefore trust the proxy to enforce the correct security constraints. By doing this we do not solve the problem of mobile proxies—we shift it to the problem of how to establish trust in proxy objects and, by implication, trust in the service provider that supplied the proxy. In the sequel we describe how this can be achieved.

Proxy Integrity. An obvious thing that is required to establish trust in a proxy object is to ensure its integrity. The object should not be changed on its way from the service (via the lookup service) to the client. As said before, an object consists of the two parts state and code. Both parts' integrity must be ensured. It is therefore necessary to digitally sign the code as well as the state. As we do not want anybody to observe the in-traffic service descriptions, the connections between the lookup service and its clients[1] should be encrypted.

Lookup Service Interaction. Even if we have encrypted communication and authentic objects, we still have to trust the lookup service. Even if a lookup service provides us with untampered objects, it might do so in an unfair manner. Instead of sending us the cheapest service (or whichever we are interested in), it might always only supply its preferred service provider. From a service provider's view even the knowledge of a service's existence might be considered a valuable asset that must be protected. For example, a network operator might have a Jini network management service. The knowledge of its existence might be interesting to competitors. A competitor could find out about that service by simply starting its own lookup service waiting for the service to register itself. It is therefore necessary for a client to trust the lookup service it is talking to. This can be achieved by requiring the lookup service to authenticate itself to its clients.

Now that we trust the infrastructure, it is still possible to have malicious services registered with secure lookup services. We therefore require services to authenticate themselves to the lookup service. Likewise, clients too are required to authenticate themselves to the lookup service. This is an obvious requirement as it is important to make sure that only authorized people access somebody's bank account.

An alternative for the indirect authentication (via the lookup service) would be to shift the authentication to a mutual authentication procedure between each service and client directly. Besides the disadvantage of needing authentication methods in every service interface, trust could only be established *after* the proxy has been downloaded to the client. This is too late as unknown code (e.g. the constructor or the method for performing authentication) is already executed at the client.

Administrative Issues. So far, the described requirements allow us to have trusted proxies. No distinction was made between different services: they all have the same security

[1] From the lookup service's point of view, any communication partner, whether it is an actual Jini service or client, is considered a client.

©Springer-Verlag

level. But usually different levels of security are desired. An example are administrators that have access rights for more services than ordinary users. We therefore partitioned the services by introducing *secure groups*. Services that have the same access restrictions are put together to form a secure group. Every service registration is associated with one secure group. Clients need appropriate access rights to view and access members of a group. The same holds for services: to prevent them from registering in arbitrary groups, they too must have the appropriate access rights. We therefore introduced two different access rights, namely *register* and *lookup*, which are currently sufficient to model access rights to groups.

Summary. To wrap up this section, we summarize the requirements that our secure Jini architecture has to fulfill:

- signed proxy objects (code and state),
- encrypted communication with lookup services,
- authentication of all participants (lookup services, services, and clients),
- access control to services, and
- limited visibility of service descriptions.

These requirements have guided the development of a secure Jini service architecture.

4 Architecture

Our design was influenced by two objectives. First of all we wanted to preserve compatibility with the existing Jini specifications: legacy clients and services should run without changes. Secondly, it was our aim to keep as much as possible of the dynamic behavior of a Jini federation, although this conflicts with security aspects as we will show later.

Figure 2 illustrates the Jini architecture with our security extensions: additional components are a *certification authority* (CA) and a *capability manager* (CM). Certificates provide for authentication of all participants. Capabilities are used for access control in the lookup service. The capability manager administers the rights for each user.

The only usable solution for the problem of opaque proxies is trust. In a dynamic environment with thousands of services it is impossible for an entity to make a decision about the trustworthiness of each service on its own. In our architecture this process is therefore delegated to the combination of lookup service and capability manager. They are part of the trusted computing base of our architecture.

Secure transfer of the proxy is guaranteed by adding a digital signature to the response message (callback) of the lookup service in the discovery protocol. Service descriptions are kept private by an encrypted connection between lookup service proxy and lookup service.

We have introduced the concept of secure groups in the lookup service. Every access to these groups is controlled by capabilities. A lookup service client must present appropriate capabilities for both registering and looking up services. Through this it is possible to restrict the registration in groups with high security to known services which meet the requirements and to control to whom the service descriptions are passed. Services in other groups are invisible.

©Springer-Verlag

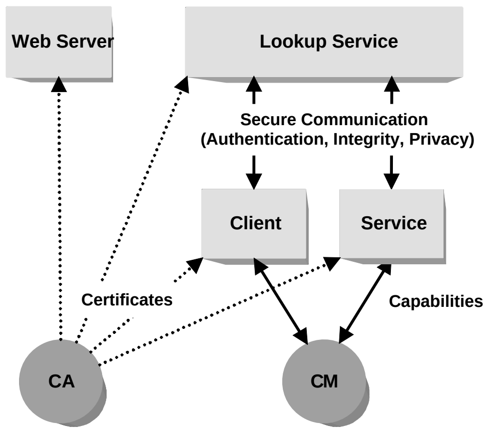

Fig. 2. Secure Jini Architecture

4.1 Certification Authority

Certificates provide for authentication of all participants. They are signed by a well-known certification authority, whose public key is assumed to be known by everyone. A certificate binds an entity's public key to its distinguished name. For authentication an entity proves that it possesses the corresponding private key using a challenge response protocol.

There are four categories of certificates and keys issued by the CA:

1. for signing system classes and LUS proxy code,
2. for the capability manager used to authenticate and sign capabilities,
3. for lookup services used to authenticate and sign LUS proxies, and
4. for clients and services used for authentication.

A signature is rejected if the signer's certificate does not belong to the appropriate category. This ensures that an entity cannot simulate another component without proper authorization. The use of certificates and the administration of the capabilities require some initial configuration. The lookup service additionally uses the capability manager's key to verify the presented capabilities. These administrative requirements obviously reduce the dynamics of a Jini federation in which otherwise arbitrary services and clients can participate without any control restrictions.

4.2 Secure Lookup Service Discovery

Before any interaction occurs, both client and service have to locate a lookup service using the discovery protocols. As a result a participant gets a proxy for the lookup service which performs the actual communication. The proxy is transferred as a serialized object which contains its codebase and its state. It is important to assure that this proxy arrives unmodified and that it is authentic. The signature for the transferred serialized

©Springer-Verlag

object and the signer's certificate are therefore added at the end of the response message of the lookup service. This guarantees compatibility with the existing protocols because the additional data is ignored by standard clients. The signature can be used to verify the integrity and to identify the signer.

If the proxy classes are unknown, the code will be loaded from the web server indicated by the codebase in the serialized object. Again it has to be ensured that the classes arrive unchanged and are trustworthy. This can be achieved by signed classes using the standard mechanisms Java already provides. Before an entity uses the lookup service proxy it has to verify the signatures of both object and code and has to make sure that the signers are authorized by the CA.

4.3 Lookup Service

In our extended architecture the lookup service is not only the main component in the service discovery process but also the center of the trusted computing base. It has to enforce the capability-based access control mechanism and assure the privacy of the service descriptions. Therefore, the lookup service has to be authentic and the communication with it must be secure.

The lookup service permanently listens for request messages. It is therefore open for denial of service attacks. A message format which supports authentication may be a solution. In this case the lookup service would not even respond unless the request is from an authorized source. We did not evaluate this option because of our goal of compatibility. Practical tests have to show whether this is a real security problem.

4.4 Secure Groups

To simplify the administration of access control and to make a differentiation between secure and less secure services possible we have decided to organize service descriptions in groups. These groups are not to be confused with Jini's group concept for organizing lookup services. We therefore developed an orthogonal concept for the management of services. A client or service needs access rights for a secure group before it is allowed to perform any action on it. Every entity proves its authorization by an appropriate capability.

It is useful to arrange the groups in a hierarchy. An authorization for a group implies the same rights for all subgroups. By this, a number of groups can be united in a simple manner. A group is represented by its name which is denoted like a package name in Java: *group.subgroup.subsubgroup...* For example, we can have the two groups *ito.printers.deskjet* and *ito.printers.lj4000* which contain services for different printers. The right for *ito.printers* permits access to all available print services.

To maintain compatibility with the Jini specifications a special *public* group exists which can be accessed without any permission. Legacy services and clients use this group for registration or lookup.

4.5 Capabilities and the Capability Manager

A client or service proves its authorization to the lookup service through a capability object. A capability is similar to a certificate that contains an entity's name and its access

©Springer-Verlag

rights. It is signed by a central authority called capability manager. The CM administers a list of names and the associated access rights. Upon request the CM creates a capability object and signs it with its private key. Capabilities allow for offline verification, i.e., the verification can be done even if the CM is not accessible. Like a certificate, a capability is not delegateable and can only be used by an entity which can proof that it is the mentioned subject. Hence, there is no need to protect the communication with the capability manager, although it is definitely necessary to protect the capability manager itself from unauthorized access.

We have implemented capability managers as Jini services. For this purpose a special group has been introduced in which registration is restricted to authorized CM services only, but lookup is open to all users. This is necessary for the bootstrapping process, because an entity must get its capabilities before it can access restricted services.

5 Implementation

The implementation of our security architecture is based on the source code that comes with Sun's reference implementation of Jini (version 1.0). The parts we changed are the implementation of the lookup service and the classes which handle the discovery protocols. Additionally, we have implemented a capability manager as a separate Jini service.

Sun's implementation of the lookup service is called *Reggie* (package *com.sun. jini.reggie*). It consists of two parts: the actual directory service (*RegistrarImpl*) and a proxy object (*RegistrarProxy*). Both communicate via Java's RMI mechanism. We protect the RMI message exchange by tunneling RMI traffic through the SSL protocol. An SSL socket is therefore created instead of the standard socket. SSL has the advantage that besides encrypting the communication it can also be used for authentication of participants. The freely available ITISSL [Pop99] package has been used as implementation of the SSL-API. The lookup service authenticates itself by presenting its certificate in the SSL handshake.

The certificates used by SSL are issued by a certification authority. In an experimental setup the *ca*-tool that comes with ITISSL is sufficient. In a deployment environment a commercial variant should be used. The security of the architecture highly depends on the correct use of certificates.

The functionality of the lookup service is described by the *ServiceRegistrar* interface (package *net.jini.core.lookup*). We added new *lookup* and *register* methods which take the user's capability and a group name as additional parameters. The group name indicates the desired group for registration and lookup. Access rights for this group must be implied by the presented capability. The lookup service otherwise rejects the requested action.

A capability consists of a name and a list of permissions. We use signed objects (*java.security.SignedObject*) for capabilities. A special permission class describes an entity's rights. It is similar in structure to a file permission with the group being the target and *register* or *lookup* being the possible actions. The capability manager is implemented as a Jini service and communicates with its proxy via RMI over SSL.

©Springer-Verlag

6 Example Scenario

In this paragraph we describe an example scenario to clarify the communication flow in our extended architecture. Our scenario consists of a service which wants to register itself in the group ”secure services” and a client which performs a lookup in this group. We assume that all certificates and capabilities have already been set up and that the capability manager is registered at the LUS in the special group ”capability”.

Service registration:

1. **Lookup Service Discovery.** The service sends a conventional Unicast Discovery Request message and gets an extended response from the lookup service. This response contains the signature for the lookup service proxy and the signer's certificate. Before the service uses the proxy object, it checks the certificate and the signature. The proxy is rejected if the certificate is not of the appropriate category or the issuer is not a known CA. On deserialization, the code of the proxy object is loaded from a web server. It is only accepted if it has been signed by an entity that has obtained a ”class signing” certificate from the CA.
2. **Secure communication/Authentication.** The lookup service proxy establishes a secure communication session between the client and the lookup service with mutual authentication. Communication is stopped if the lookup service cannot present a certificate issued by a trusted CA. The service uses its own certificate for proving its identity.
3. **Capability Manager lookup.** The service calls the LUS proxy's *lookup* method to find an instance of the capability manager. It specifies the group ”capability” as parameter. This ensures that only trusted CM services, which are allowed to register in this special group, are returned.
4. **Obtaining Capabilities.** The service asks one of the CMs for its capabilities. The CM consults its database and creates an adequate capability object containing the permissions of this service. The capability is delivered inside a signed object using the CMs private key to guarantee its authenticity.
5. **Registering at the LUS.** The service calls the LUS proxy's *register* method with the desired group ”secure services” and its signed capability as additional parameters. The capability is only accepted if the contained name equals the distinguished name presented during the authentication phase (see step 2). The LUS verifies the signature of the capability using the CM's public key and checks if the permission for the specified group is implied. Upon success, the lookup service adds the service description to this group, otherwise it rejects the operation.

Client side service look up: Steps 1 to 4 are the same as above.

5. **Service lookup.** The client calls the LUS proxy's *lookup* method with the group ”secure services” and its signed capability as additional parameters. The LUS verifies the capability and checks if the permission for the specified group is implied. Upon success it returns all services of this group which match the given service template.
6. **Service use.** The client selects one service from the result and uses the service proxy for further interaction.

©Springer-Verlag

7 Related Work

There are a few research efforts that partly deal with the same area as the work described in this paper. A number of other technologies enabling dynamic service discovery exists. Among those we chose SLP and SDS and take a short look at their security features. Another effort promising to bring security properties to the Jini architecture is the RMI Security Extension.

ICEBERG Service Directory Service. The SDS [CZH+99] is the central service trading component of the ICEBERG project at UC Berkeley. Service providers use the SDS to advertise service descriptions, while clients use the SDS to query for services they are interested in. Services are described with XML [BPSM98] documents that encode different service properties, e.g., service location. SDS has been designed with security properties in mind. All security critical communication is either encrypted or authenticated. Similar to our approach capabilities are issued by a capability manager to allow service providers to register their services with an SDS server.

Service Location Protocol. SLP [VGPK97] is a service trading architecture that enables service providers to register service descriptions with a central component called *directory agent*. Although communication between SLP components is unprotected, SLP offers so-called *authentication blocks* to digitally sign messages to ensure integrity of the transmitted data. SLP does not specify how key distribution should be managed in an SLP environment.

RMI Security Extension. Sun Microsystems is currently working on an extension to RMI that is supposed to allow secure interaction with RMI-based servers including the establishment of trust in downloaded proxies. The specification [Sun99a] is currently in draft status. It allows fine-grained control of different security properties. While the extension is currently only aimed at RMI it is supposed to be possible to use the same methods and interfaces for other middleware architectures as well.

The most interesting part of the specification deals with the establishment of trust in downloaded proxies. The basic method used here is to allow only trusted code to be run. Further security properties (e.g. authentication and encryption) are then guaranteed by the trusted code. Trusted code includes dynamically generated RMI stubs. If a proxy is not an instance of a trusted class, it is asked to present another object which is trusted. The associated server is then asked if it trusts the original object. This method seems to restrict spontaneous networking to RMI-based services.

Furthermore, a few problems that we regard as essential are not addressed. First, objects are instantiated before establishing trust. Malicious code could therefore be executed in the constructor of the proxy. Secondly, the specification is aimed at RMI in general and does not address Jini in particular. Services are therefore still visible to everybody. Different security levels can only be enforced after downloading the service's proxies and depend on their enforcement by every client and server.

©Springer-Verlag

8 Conclusion and Future Work

With our approach we believe to have solved the most urging security problems in Jini environments. Clients can safely assume that the service proxies running in their JVM have been properly authenticated to the security infrastructure and have been shipped without loss of integrity. Service providers themselves trust the infrastructure that only clients with the correct capabilities are able to access them. This might be important in those cases where service providers are only interested in the fact that services are accessed by authorized clients only, without exactly knowing the identity of the client. We think that for many application areas the fact that the infrastructure guarantees certain security properties simplifies the development and shipment of services to a significant extent.

While we presented a solution to the problem of secure service registration and lookup, it is important to note that this covers only a part of the Jini architecture. The Jini specifications describe a number of further concepts that were not considered in our research. These concepts are leasing, distributed events, and transactions. We do not know yet what the security concerns are, not to mention how to solve possible risks.

But even in the presented architecture, a few questions are still open. We assume that there is one central CA. In a dynamic environment, a distributed architecture would probably be a more favorable solution. An overview of work in this direction can be found in [Per99].

Despite the obvious advantages of a secure service infrastructure we should not forget that it does not come for free. The drawback is the partial loss of "spontaneity" of client/service interactions which was said to be one of the main advantages of Jini. Plugging devices and services into the network, spontaneously finding these devices via the lookup service, and using them are easily done. Establishing trust relationships in such spontaneous environments seems to be a task that results in a decrease of spontaneity, since prior to actual use administrative processes (e.g. distributing keys) must take place first.

Open is the question whether the trade-off between trust and spontaneity can be avoided by additional means that take the mobility of users and devices into account. We think that mobility is likely to be a driving force for changing service environments. Further work aims at identifying properties and usage models that may facilitate key distribution and granting of capabilities in our architecture.

Acknowledgments

We would like to thank Andreas Zeidler and Prof. A. Buchmann for their valuable comments on an earlier version of this paper.

References

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language XML 1.0.* W3C, February 1998. Available at http://www.w3.org/TR/1998/REC-xml-19980210.

©Springer-Verlag

[CZH$^+$99] Steven Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual International Conference on Mobile Computing and Networks (MobiCOM '99), Seattle, WA*, August 1999.

[Gon98] Li Gong. Java Security Architecture (JDK 1.2). Technical report, Sun Microsystems Inc., October 1998.

[Per99] R. Perlman. An Overview of PKI Trust Models. *IEEE Network*, 13(6):38–43, November 1999.

[Pop99] A. Popovici. ITISSL - A Java 2 Implementation of the SSL API based on SSLeay/OpenSSL. http://www-sp.iti.informatik.tu-darmstadt.de/itissl/, 1999.

[RG98] A. D. Rubin and D. E. Geer. Mobile Code Security. *IEEE Internet Computing*, 2(6):30–34, November 1998.

[Sun99a] Sun Microsystems Inc. *Java Remote Method Invocation Security Extension (Early Look Draft 2)*, September 1999.

[Sun99b] Sun Microsystems Inc. *Jini Architecure Specification – Revision 1.0.1*, November 1999.

[Sun99c] Sun Microsystems Inc. *Jini Discovery and Join Specification – Revision 1.0.1*, November 1999.

[Sun99d] Sun Microsystems Inc. *Jini Lookup Service Specification – Revision 1.0.1*, November 1999.

[VGPK97] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service Location Protocol (SLP). Internet RFC 2165, June 1997.

[Wal99] Jim Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, July 1999.

©Springer-Verlag