TOWARDS AN ALTERNATIVE WAY OF VERIFYING PROXY OBJECTS IN JINI

Nikolaos Papamichail, Luminita Vasiu
School of Computer Science, Middlesex University, London, UK

Keywords: Jini Security, Proxy Trust Verification

Abstract: Jini networking technology represents an exciting paradigm in distributed systems. Its elegant approach in computer networking possesses immense advantages, but also generates security problems. Extensive research has been undertaken and existing security methodologies have been applied to provide a safe execution environment. However the unique nature of Jini has made it hard for traditional security mechanisms to be applied effectively. Part of the problem lies within the downloaded code and in the lack of centralised control. Current solutions are based on assumptions; therefore they are inadequate for enforcing the security requirements of the system. The goal of our research is to increase the security of the Jini model without altering its initial characteristics. We present our preliminary research efforts in providing an alternative, fault tolerant security architecture that uses a trusted local verifier in order to evaluate and certify the correctness of remote calls.

1 INTRODUCTION

Jini networking technology (Sun Microsystems Inc.2003a; http://www.jini.org/) presents an exciting paradigm in distributed computing. Based on the Java programming language, it allows the development of spontaneous networked systems. Users and applications are able to dynamically locate one another and form on-the-fly communities. Unlike traditional systems that rely on a fixed protocol and central administration, Jini requires no further human intervention once being set up. It employs strong fault-tolerance mechanisms that do not attempt to eliminate or hide the fact that network failures may happen. On the contrary it provides a programming model and an infrastructure that allow developers to recognise and isolate any faults that might occur.
When Jini was made publicly available, no security has been taken into consideration. The Java language alone was not adequate to cope with the security required in a distributed setting. Although some solutions have been proposed, Jini lacked a generic security model that could be applied to counter any threats that might arise. The Davis project (http://davis.jini.org/) presents such a security model that has been recently incorporated into the latest Jini release. The security model is based on well known and proven techniques to enforce the basic requirements for network security. However, some of the mechanisms that Jini employs are unique in distributed computing. Additionally, neither any real world applications that make use of the model nor a formal evaluation of it have appeared yet. Thus any assumptions about the correctness of the design and the degree of security provided might prove to be mistaken. The purpose of our research is to examine the security model employed by Jini technology for any potential security faults and propose appropriate modifications. In this paper we focus in the algorithm responsible for verifying trust in Jini proxy objects.
The rest of the paper is organised as follows. Section 2 presents an overview of the Jini programming model and infrastructure, particularly the components that constitute a Jini system and other mechanisms relevant to Jini operation. Section 3 presents some security problems related to proxy objects, Lookup Services and Jini Services while Section 4 presents an overview of the current Jini security model, the Davis Project, and a critical approach to its proxy verification algorithm. Section 5 presents an outline of two proposed solutions to the issues related with proxy object verification and the advantages that they may possess. Section 6 presents related work and some concluding thoughts are drawn in Section 7.

Papamichail N. and Vasiu L. (2004).
TOWARDS AN ALTERNATIVE WAY OF VERIFYING PROXY OBJECTS IN JINI.
In *Proceedings of the First International Conference on E-Business and Telecommunication Networks*, pages 68-75
DOI: 10.5220/0001402000680075

2 BACKGROUND

Jini (Sun Microsystems Inc., 2003a; http://www.jini.org/) is a distributed system based in Java that allows the establishment of spontaneous network communities or federations. To make that possible, Jini provides the following:
An infrastructure that enables devices, human users and applications to dynamically discover one another without any prior knowledge of their location or of the network's topology and form dynamic distributed systems. The infrastructure is composed of a set of components based on Jini's programming model. Parts of the infrastructure are the discovery join and lookup protocols and the Lookup Service (Sun Microsystems Inc., 2003a). A programming model that is used by the infrastructure as well as by services. Besides service construction, the programming model provides interfaces for performing leasing as well as event and transaction handling.
Services that are employed inside a federation and provide some functionality. Services exploit the underlying infrastructure and are implemented using the programming model.

2.1 Services

Every entity that participates in a Jini system and provides some functionality is perceived as a service. No separation is made regarding the type or the characteristics of the service. A service could be either a hardware device, a piece of software or a human user. Jini provides the means for services to form interconnected systems, and each one separately to offer its resources to interested parties or clients. The separation between a service and a client, however, is sometimes blurred, as sometimes a Jini service may act both as a service and a client.
A word process application, for example, is perceived as a service by any human user that writes a document, although the same application acts as a client whenever it uses a device such as a printer. The latter is again a Jini service, thus for the infrastructure the word application is now its client.

2.2 Proxy objects

In order for services to participate in a Jini system they must create an object that provides the code by which they can be exploited by potential clients, the proxy object. The proxy object contains the knowledge of the service's location and the protocol that the service implements. It also exposes an interface that defines the functions that can be invoked. A client is able to make use of a service only after the correspondent service's proxy object is downloaded to the client's local space. By invoking functions defined in the proxy interface, clients are able to contact and control services. Clients need only to be aware of the interface that the proxy implements and not of any details of the proxy implementation.

2.3 Lookup Service

The Lookup Service (LUS) is a special kind of service that is part of the Jini infrastructure. It provides a mechanism for services to participate in a Jini system and for clients to find and employ these services. The Lookup Service may be perceived as a directory that lists all the available services at any given time inside a Jini community. Rather than listing String based entries that point back to the location of a service, the Lookup Service stores proxy objects registered by Jini services.

2.4 Discovery Join and Lookup

Relevant to the use of Lookup services are three protocols called discovery, join and lookup (Sun Microsystems Inc. 2003a). Discovery is the process where an entity, whether it would be a service or a client, is trying to obtain references to a lookup service. After a reference has been successfully obtained, the entity might register a proxy object with the Lookup service (join), or search the Lookup Service for a specific type of service (lookup). The discovery protocol provides the way for clients and services to find available Lookup Services in the network, and for Lookup Services to announce their presence.

3 JINI SECURITY ISSUES

Typically security is concerned with ensuring the properties of confidentiality, integrity, authentication and non-repudiation (Menezes et. al, 1996):

- Confidentiality ensures that information remains unseen by unauthorised entities
- Integrity addresses the unauthorised alteration of data
- Authentication is the verification of identity of entities and data
- Non-repudiation prevents an entity from denying previous commitments or actions

These properties are generic and apply to a wide variety of systems. Inside Jini, no prior knowledge of the network's infrastructure is assumed. For that reason, Jini is not only bound to security problems related to distributed systems, but also to any additional issues that the spontaneity of the environment invokes. The following components present different security requirements and they will be examined separately.

3.1 Proxy Object Issues

Nothing should be able to alter the state of the proxy object, either by intention or by fault. That means that the integrity of the proxy object must be ensured (Hasselmeyer et. al, 2000a). Since the proxy object is downloaded from an unknown location in the network, neither the source nor the intentions of the proxy object can be verified. Therefore, even the act of downloading the proxy of a service is considered by itself a security risk. Moreover, the proxy is responsible for performing the communication between the client, and the service that the proxy represents. Therefore the integrity and confidentiality of the communication has to be preserved, since the communication link might be intercepted, altered, or simulated by someone with malicious intentions. The privacy and anonymity of the client may be abused, because the client can not be ensured that the proxy does indeed provide the functionality it claims (Hasselmeyer et. al, 2000a). On the other hand it has to be verified that any data that needs to be supplied to the proxy object, for the interaction with the service to take place, reaches the appropriate service (JAAS).

3.2 Lookup Service Issues

The Lookup Service lacks any mechanism for authenticating services (Schoch et. al, 2001). That means every service can discover the Lookup Service and register its proxy. Malicious proxies may register and pretend they provide some functionality, while they don't. Moreover, every client can search the Lookup Service and find which services are provided. Some services may require only registered users to access them. Therefore access control mechanisms need to be imposed. Additionally, clients might encounter unfairness while searching the Lookup Service for available services (Hasselmeyer et. al, 2000a). There is no way a client of a service can be assured that he received the best available service from the Lookup Service. The fact that every service can register and even re-register with the Lookup Service can lead to "man-in-the-middle" attacks (Schoch et. al, 2001). A malicious service just has to re-register its proxy with the same service ID as the original one. Every time a client tries to access the required service, the Lookup Service may provide him with the new, malicious proxy. The client is unaware of the change, as the new proxy looks like it implements the same interface as the original one.

3.3 Service Interaction issues

In order for an interaction between two services to take place, the service acting as a client must first locate the provider of the desired service, via the process of discovery, and then download its corresponding proxy object. However, in a spontaneous environment like Jini, hundreds of services may be present at the same time and many of them may provide the same functionality. No standard names or address for recognising individual services exist, besides a unique service ID that is assigned by the Lookup Service. However, it is dependent upon the provider of each service to decide whether or not the assigned ID will be stored and used in any future transactions. Therefore clients have to be able to authenticate the services they access (Eronen et. al., 2000). Similarly, the service provider has to be able to authenticate clients that try to use its resources and call its provided functions.
Another aspect in the service interaction is different access levels (Kagal et. al, 2001). An obvious solution to this problem is the integration of access control lists. Every user could be identified by a unique username and password that would grant him or deny certain permissions. However, new problems arise, like the distribution of the appropriate keys and the way that the permissions are to be decided.

4 THE DAVIS PROJECT

The Davis project (http://davis.jini.org/) is an effort led by Sun Microsystems' project team responsible for the development of Jini. The purpose is to satisfy the basic Jini requirements for security, by providing a security programming model that would be tightly integrated with the original Jini programming model and infrastructure. Part of the requirements (Scheifler, 2002) has been to avoid changing any existing application code by defining security measures at deployment time. Also to extend the security mechanisms provided by the Java programming language, such as the Java Authentication and Authorisation Service (JAAS).
The Davis project has been integrated with the original release of Jini networking technology (Jini specifications archive – v 2.0) resulting in the

release of the Jini starter kit version 2 (Sun Microsystems Inc., 2003a).

4.1 Constraints

In order to support a broad variety of applications and requirements, the security model dictates that both service providers and their clients should specify the type of security they require before any interaction between them takes place. Decisions upon the type of the desirable security are expressed by a set of constraints that have the form of Java objects. Any service that wishes to incorporate security in its current implementation has to implement a proxy object that implements a well-known interface (Sun Microsystems Inc., 2003b). The interface defines a method for clients and services to set constraints to the proxy object. If the proxy implements that interface, all the imposed constraints apply to every single call through any method defined by the proxy. The basic constraints are the equivalent of Boolean constants that allow decisions upon the type of security required to be specified in proxy objects. Typically service providers specify the constraints during the proxy creation, while clients set the constraints after the proxy object has been downloaded. Constraints specify only what type of security is expected but not how this is implemented.
The security model dictates that constraints imposed by services and clients are combined to a single set of constraints. If any of them contradict with each other then no calls are performed. It is possible, however, that alternative constraints are defined. This provides an elegant way for all parties participating in a Jini interaction to have direct control over the security imposed.

4.2 Object Integrity

There are two mechanisms that the current security model employs to provide integrity for the code of proxy objects. Both assume that the http protocol is used. The first mechanism is http over SSL (https) (Rescorla, 2000), the standard protocol for providing web site security in terms of server authentication, confidentiality and integrity. The other is a custom defined protocol called HTTPMD (Scheifler, 2002; Sun Microsystems Inc., 2003b) The proxy object consists of code which is downloaded by clients, and data which is downloaded from the service. Therefore to ensure total integrity these mechanisms have to apply to both the location where the proxy object is downloaded from and the location of the object's codebase. Along with integrity, the https protocol provides confidentiality and encryption, resulting in additional overhead when these are not required. In these cases the HTTPMD protocol is used. The location of objects, including their code, is specified by a normal http URL. Attached to it is a cryptographic checksum of the contents of the code, a message digest (Rivest, 1992). By computing the checksum of the downloaded data and code and comparing it with the attached message digest, clients are ensured that integrity has been preserved, since any modification in the contents would result in a different message digest.

4.3 Proxy Trust Algorithm

In terms of deciding whether a client trusts a proxy object downloaded by an unknown source, the current model (http://davis.jini.org/) employs the procedure described below. It is assumed that the client has already downloaded a proxy object from somewhere but it can not yet trust neither the proxy object not its correspondent service. Initially the client performs an object graph analysis of the proxy object. By checking recursively all the classes that the object is composed of it can be determined whether these classes are local or not. If the classes are local, in perspective to the client, then the proxy object is considered trusted. This is accepted on the basis that all local code is considered trustworthy. In the case where the proxy object is not fully constructed of local classes, the following components take part in the proxy trust verification algorithm:
1. Proxy object
This is the object that implements the server's functionality. It is downloaded by the client, traditionally from the Lookup Service and it contains the knowledge of how to communicate back with the server. All remote calls to the server are passing through this object and this is the object that needs to be verified.
2. A 'bootstrap' proxy
If the object graph analysis proves that the classes used for the construction of the proxy object are not local relatively to the client, the client uses the initial proxy object to request another object called the 'bootstrap' proxy. The bootstrap proxy should be only consisted of local classes (relevant to the client). The purpose is that clients can trust an object that only uses local code to run. The 'bootstrap' proxy is also used to authenticate the server to the client, as well as to provide him with the verifier described next.
3. A proxy Verifier
The Verifier is an object sent to the client by the server, using the 'bootstrap' proxy. It checks the

downloaded proxy object in order to verify whether the server trusts the initial proxy object or not.
A client obtains a proxy object from the network using Jini discovery and lookup mechanisms. The client examines whether the proxy object is using local code (relative to the client). Since this is normally not the case the client has to verify whether the proxy object can be trusted. The way that this is performed by the current security model is illustrated in Figure 1.

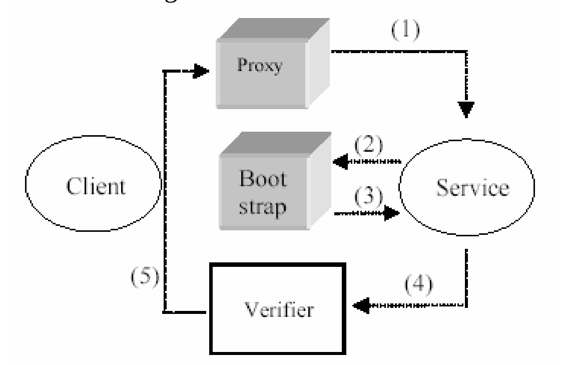

Figure 1: The proxy trust authentication employed by the current security model

In order to verify that the proxy originates from a legitimate service, the client has to contact the same service and ask the service whether the proxy should be trusted. Since there is no way to directly contact the service, the client places a call through the proxy it can not yet trust, asking for a 'bootstrap' proxy (1). The bootstrap proxy has to use only local classes, relative to the client, in order to be considered trustworthy. After the bootstrap proxy is downloaded to the client's local address space (2), and the locality of the classes that compose the bootstrap object is verified, the client performs a call through it (3). Part of the call is to request from the service to authenticate. After the service has authenticated successfully, it passes a verifier object to the client (4). Finally the verifier is used to verify the legitimacy of the initial proxy object (5).

4.4 Critical Review of the proxy trust algorithm

A number of potential problems might arise from the verification algorithm described above. The first is that clients have to rely on an object downloaded from an unknown source (the proxy object) to obtain the bootstrap proxy. In order for the latter to occur, clients have to place a remote call through an untrusted object. Since the functionality that the proxy object implements is unknown, clients may unintentionally execute an operation that presents a security risk in case the proxy is a malicious object. The second problem is that the service provider has to have some knowledge of the type of classes that are local to the user. If the bootstrap proxy is not consisted entirely by local classes, relevant to the client, the client would not utilize it to obtain the verifier.
A third type of problem is related to the way and type of checks that the verifier performs to the proxy object. There is no standardised set of tests that could be performed, since these are left for the service providers to implement. The method suggested is that the verifier carries the code of the proxy object. By checking the equality of the code that the verifier carries with the code of the proxy object, it is possible for a service to identify the correctness of the proxy object. However, there is no way to ensure whether the checks performed are adequate or if any checks are performed at all.
Therefore a 'lazy' verifier that just confirms the correctness of proxies without performing any checks might incorrectly identify a malicious object as a legitimate one.
Finally faults might occur if a service provider updates the implementation of the proxy object without updating the implementation of the verifier too. In that case legitimate proxy objects would not be able to be identified correctly, since the equality check would fail. Therefore the service provider might unintentionally cause a denial of service attack not initiated by a malicious client, but by himself.

5 AN ALTERNATIVE WAY OF VERIFYING PROXY TRUST

Instead of relying on the untrusted proxy object downloaded from an unknown source to obtain a proxy verifier, clients might be able to protect themselves from malicious proxy objects by using their own local verifier. The verifier is generated locally by clients before any participation in a Jini federation takes place. In order to create the verifier, clients specify their security requirements such as authentication, confidentiality and integrity. These requirements are injected to the verifier and might vary for different scenarios. Specification of the security requirements is similar to the concept of constraints specified by the current Jini security model (http://davis.jini.org/). This permits the specification of application independent security requirements and allows better interoperability with the current security model. The difference is that the client requirements are not injected into a

downloaded proxy, but into the locally generated verifier.
The notion of a locally generated verifier is central to all of the proposed solutions. The operations that the verifier performs, however, are different in every variation of the algorithm. The entities employed in all the solutions proposed here case are the following:

- Client: The entity that wishes to use a service. Clients need to be protected from any potential hazards.
- Proxy object: Typically the object that is downloaded by clients and used to access services. Presents the major source of incoming threats.
- Local Verifier: An entity generated by clients before any interaction with downloaded objects takes place. Used to either verify proxy objects or isolate clients from them.
- Service: The entity that lies somewhere in the network and provides some functionality. Services supply proxy objects and should be considered untrusted.

In every proposed solution it is assumed that a service has already discovered an available Lookup Service and registered its proxy object. The client is ready to perform discovery and lookup in order to obtain a proxy object from the same Lookup Service.

5.1 Proxy Verification Based on a Local Generated Verifier

In order to verify that the downloaded proxy object can be trusted, the following process is performed:
1. Before any discovery process takes place, the client generates a local verifier
2. Client's security requirements are injected to the verifier by the client
3. The client performs discovery of the Lookup Service and downloads a service proxy object
4. Before any interaction with the proxy takes place, the proxy object is passed to the verifier
5. The verifier performs a series of security checks according to the client requirements and makes a decision on behalf of the client about the trustworthiness of the proxy object
6. In case the verifier has decided that the security requirements are satisfied, the client interacts with the service through the proxy object as defined by Jini programming model.

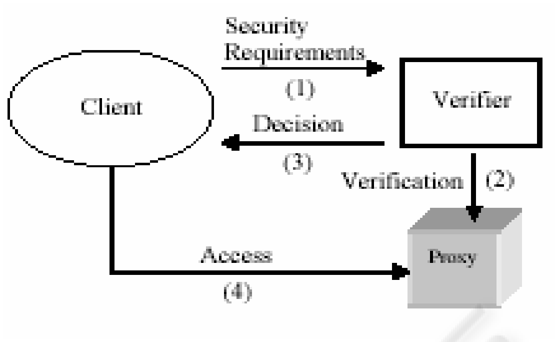

Figure 2: Proxy trust verification by a local verifier

The described process is illustrated in Figure 2. Initially the client generates the verifier and specifies the security requirements (1). The verifier performs a series of tests to verify trust in the proxy object (2). The result of the verification procedure is expressed as a decision and the client gets notified (3). If the proxy has been considered to be trustworthy, the client is allowed to contact the proxy object (4) and access the related service. Comparing this solution with the default proxy verification algorithm, in both algorithms the client is responsible for specifying the type of security required. However, the entity that is responsible for enforcing these requirements is not an untrusted proxy object anymore, but a locally generated verifier. The type of checks performed and the way these are carried out is much more transparent from the client's point of view. Moreover, clients do not have to rely on a verifier object downloaded from a service since the process of such object verifying the initial proxy object is not clear to the client.
Therefore the problem of a service generated verifier that performs no actual check to the proxy object, resulting in the verification of a faulty proxy, is eliminated.
Service providers also do not need to worry about having to provide a bootstrap proxy and a verifier. The only entity that services need to expose is the default proxy object. Absence of a bootstrap proxy eliminates the need for services to implement an object based on the assumption that it would consist of classes that the client already has. Moreover, the current algorithm dictates that every time the implementation of a proxy object changes, the verifier object has to change as well, since proxy verification is based on equality checking. Finally by eliminating the need for services to produce two additional objects (the bootstrap proxy and the verifier), administration burden is removed from the service provider.

5.2 Restricting Proxy Object in a Controlled Environment

1. Before any discovery process takes place, the client generates a local verifier
2. Client injects to the verifier the security requirements and the maximum amount of local resources permitted for use by proxy objects
3. The client performs discovery of the Lookup Service and downloads a service proxy object
4. The verifier provides a controlled environment for the proxy object to run. Besides performing security checks to the proxy object, the verifier ensures that the proxy does not use more resources than specified. All requests to and from the proxy object pass through the verifier.

Figure 3 illustrates the followed process. The client generates a local verifier and assigns the security requirements as well as any resources that proxy objects are permitted to use (1). After the proxy object has been downloaded, it is passed to the verifier. The verifier performs similar type of security checks as in proposed solution 1, and additionally provides a controlled environment where proxy objects run. Any client requests and any responses from the proxy object pass through the verifier (2). The same is true for any communication held between the proxy object and its corresponding service.

The advantages of this solution are similar to those of the solution proposed in Section 5.1. The need for service providers to produce additional objects besides the default proxy object is eliminated and so are the assumptions relevant to the locality of classes in the bootstrap proxy and the checks performed by the service's verifier. Moreover, by restricting execution of the proxy object into a set of finite resources, a protected environment safeguarded by the verifier is created. Verification does not occur only once, but the verifier is monitoring the proxy object continuously. Therefore any potential hazards that might take place during the execution of the proxy are more likely to be identified and get dealt with.

6 RELATED WORK

In (Eronen et. al, 2000) certificates are used to establish trust between services and users. Secure interaction is assumed, by allowing users and services to interact only if they carry the appropriate credentials, supplied by a security library. However, these credentials must be assigned to every service of the Jini community before any interaction could be realised. That reduces the spontaneity that Jini provides, and requires prior knowledge of the services' properties to exist, in order for the appropriate permissions to be assigned correctly. Trust establishment is also the purpose of (Hasselmeyer et. al, 2000a). Trust establishment is attempted between the Lookup Service, the service provider and the client. The authors propose an extension to the Jini architecture with a certification authority, which provides certifications for the authentication of components. Capability managers are responsible for administering the rights for each user. In that way, different access levels for each client can be easily implemented. Their solution, however, assumes that one central certification authority exists, in order for the appropriate certificates and capabilities to be distributed to every Jini component that exists in the system. Thus, a prior knowledge of every service's characteristics should exist something that is not usually the case in Jini. Moreover, the existence of a centralised authority is opposed to the decentralised nature of the Jini technology. The integration of authorisation and authentication techniques in Jini is also examined in (Schoch et. al, 2001). The authors try to achieve that without introducing any additional components, besides the facilities that Jini and Java already provide. They try to prevent man-in-the-middle attacks, by signing the proxy object with a digital signature. This allows the clients to authenticate the source of the provided service, although it still can not be verified how the service users the provided by the service data.

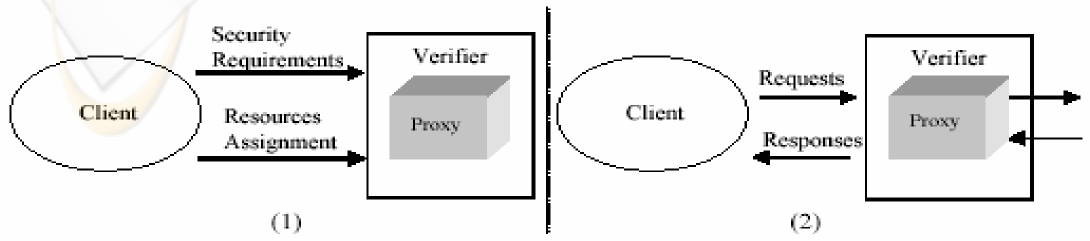

Figure 3: Verifier creation and interaction with the proxy object

7 CONCLUSION

We presented some security problems related with Jini and how they are countered by the current Jini security model. Our focus is placed in the proxy trust verification algorithm since we believe that an alternative way of verifying proxy object trust might encounter some of the existing problems. We presented our initial ideas in providing an alternative way of ensuring that hostile proxy objects would not impose any risk to clients of the system. We sketched two different approaches in solving the problem. Both involve the concept of a local generated verifier that either verify a downloaded proxy object or impose restrictions to that object's functionality. We also pointed out the advantages of these solutions. Future work includes further rectifying the presented concepts and come up with a viable solution that would integrate with the existing model. Also implement a working prototype and test it in a real world environment.

REFERENCES

Eronen, P., Lehtinen, J., Zitting, J., and Nikander, P., 2000. 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.

Hasselmeyer, P., Kehr, R., and Voß M. 2000a.Trade-offs in a Secure Jini Service Architecture. In 3rd IFIP/GI International Conference on Trends towards a Universal Service Market (USM 2000), Munich, Germany. Springer Verlag, ISBN 3-540-41024-4, pp. 190-201.

Java Authentication and Authorisation Service (JAAS) http://java.sun.com/products/jaas/ [Accessed 10 Feb. 2004]

Jini specifications archive – v 2.0 http://java.sun.com/products/jini/1_2index.html [Accessed 10 Feb. 2004]

Kagal, L., Finin T. and Peng, Y. 2001. A Delegation Based Model for Distributed Trust. In Proceedings of the IJCAI-01 Workshop on Autonomy, Delegation, and Control: Interacting with Autonomous Agents, pp 73-80, Seattle.

Menezes, A., van Oorschot, P., and Vanstone S. 1996. Handbook of Applied Cryptography. CRC Press. ISBN: 0849385237

Rescorla, E. 2000. HTTP Over TLS, the IETF Network Working Group http://www.ietf.org/rfc/rfc2818.txt [Accessed 09 Feb. 2004]

Rivest, R. 1992. RFC 1321 - The MD5 Message-Digest Algorithm, the IETF Network Working Group, http://www.ietf.org/rfc/rfc1321.txt [Accessed 09 Feb. 2004]

Scheifler, Bob 2002. Comprehensive Network Security for Jini Network Technology Java One Conference Presentation , San Francisco, March 2002 http://servlet.java.sun.com/javaone/sf2002/conf/sessions/display-1171.en.jsp [Accessed 15 Dec. 2003]

Schoch, T., Krone, O., and Federrath, H. 2001. Making Jini Secure. In Proc. 4th International Conference on Electronic Commerce Research, pp. 276-286.

Sun Microsystems Inc. 2003a. Jini architecture specification. http://www..sun.com/software/jini/specs/jini2_0.pdf [Accessed 15 Dec. 2003]

Sun Microsystems Inc. 2003b. Jini architecture specification. http://wwws.sun.com/software/jini/specs/jini2_0.pdf [Accessed 15 Dec. 2003]

http://www.jini.org/ [Accessed 11 Feb. 2004] The Davis project http://davis.jini.org/ [Accessed 11 Feb. 2004