Securing ad hoc Jini services

Pasi Eronen [1]
pasi.eronen@hut.fi

Christian Gehrmann [2]
christian.gehrmann@ecs.ericsson.se

Pekka Nikander [1,2]
pekka.nikander@hut.fi

[1] Helsinki University of Technology
[2] Ericsson Research

Abstract

In this paper, we look into the concept of trust in ad hoc wireless networks, and the role of PKIs in creating trust. By allowing some simplifying assumptions, we propose a solution for creating temporary trust relationships without central authorities. Our low complexity solution for ad hoc communication security can be elegantly implemented in an environment which supports downloadable code. We also give an overview of our prototype implementation, which uses Java and Jini.

1. Introduction

Short range radio technologies, such as Bluetooth [11], open up the possibility to offer cheap, fast, and convenient local communication services to people on the move. In order to be attractive for all kinds of applications, the ad hoc the connections used by these services must be authenticated and possibly encrypted. In this paper we show some solutions of how to secure ad hoc services.

Security in distributed systems is a large field with several different problems and aspects. The problem area spans from access control to key distribution and key management [23]. The importance of security functions in a service network depends on where the service network resides and how the users attach to the services in the network.

From a pure cryptographic point of view, ad hoc services do not imply many "new" problems. The requirements regarding authentication, confidentiality, and integrity or non-repudiation [17] are the same as for many other public communication networks. However, it is not always obvious how the well known cryptographic primitives should be used to create secure services in ad hoc networks. The lack of well pre-defined relations between the communication parties demand flexible mechanisms for setting up trust relations, distributing keys, providing access control, and creating security sessions.

To realize ad hoc services, some type of service discovery technique must be used. Examples of techniques dealing with these kinds of issues are the Bluetooth service discovery protocol [2], Jini [1], Salutation [22], Universal Plug and Play [18], and Service Location Protocol [10]. Some of these different techniques are used in some specific settings while others are rather generic technologies. Regarding security, as described above, they all have very similar problems to face.

In this paper we mainly discuss the role of trust in securing ad hoc services, and show how to secure Java Jini connections in a concrete way. Different from most of the other mentioned service discovery methods, Jini uses a distributed model that is built on allowing code to be moved between entities in the ad hoc network. Downloaded code is a security problem by it own. Hence, security is an important part of the system design of Jini ad hoc services, and we show how to address it by relying on decentralized authorization. We base our solution on the ready to use components of the Java 2 security architecture [9].

The rest of this paper is organized as follows. First, in the rest of this section, we briefly introduce Jini. Then, in Section 2, we discuss the role of trust in ad hoc networks and describe a trust distribution protocol. Section 3 explains our basic approach to securing Jini services, and Section 4 contains the details of the changes made to the Jini operation model. In Section 5 we compare our research to related work, and finally, Section 6 contains our conclusions from this research.

1.1 Introduction to Jini

The Jini [1] technology allows devices to easily form network communities without previous planning or ad-

ministration efforts. Jini devices can just be plugged in the network and they will automatically locate and join the network community, thus making their services available to anyone in the same community. No configurations need to be made and no drivers need to be installed to use such services.

The members of the Jini communities are called services. Each service offers some functionality that can be accessed through an interface defined by the service. When a service is plugged in the network, it uses a discovery protocol to locate a lookup service that manages the community. The service then joins the community by uploading its interfaces to the lookup service. The service interfaces are implemented by proxy objects that can be moved over the network to the lookup service and to any clients that want to access the service.

When a client wants to access a service, it first asks the lookup service for a proxy object for the service it wants. If such a service is available in the community, then the lookup service replies with the proxy object it received when he service joined the community. The client may then use the service through the interface implemented by the proxy.

The Jini technology also defines mechanisms for leasing and transactions that help creating resilient and reliable distributed systems [21, 27]. Leasing is used extensively by the Jini itself to recover from network failures and service crashes. A Jini community is very tolerant of partial failures and network partition. An important feature is that devices can easily be taken off the network for maintenance or upgrading and returned back without disrupting the community. Thus, Jini is especially suitable for a service layer in ad hoc networks.

2. Trust in an Ad Hoc Setting

In any security setting, trust is a fundamental property. Some of the components of the system *must* be trusted, or the system security would collapse, while some other components *may* be trusted, thereby simplifying the situation. In general, trust is one of the basic assumptions that allow us humans to behave reasonably efficiently in our everyday lives. The concept seems to imply lack of sufficient amount of knowledge [4], meaning that there is at least some amount of uncertainty involved [14, 15 19]. On the other hand, trusting reduces the complexity of a situation. When we decide to trust rather than suspect—this is what it means when we talk of a leap of trust—the number of issues we have to consider is reduced, thereby simplifying the process of making decisions. Trusting also describes an attitude towards future expectations, as well as introduces the presence of implied risk in a given situation [19].

In a wireless ad hoc network, trust is one of the central problems. Since we cannot trust the communication medium, our only choice is to use cryptography. That, however, forces us to rely on the cryptographic keys used. Thus, our basic problem to solve is the creation of initial trust relationships between keys. Using those initial trust relationships, others can be easily formed with standard cryptographic protocols.

2.1 The Role of PKIs in Creating Trust

In general, the goal with a PKI is to distribute public keys and determine whether these public keys can be trusted for certain usage or not. A piece of digitally signed information is often called a certificate; certificates are the basis upon which PKIs are built.

The most commonly used certificate format is the X.509 certificate format [13], which is an identity oriented certificate format. Basically, an X.509 certificate says that certain name, denoting a person or an organization, has a specific public key. From our point of view, that is necessarily not much, since names are inherently bound to some namespace. In the case of ad hoc networks, the parties do not necessarily share any such namespace. Another form of certification are presented by authorization certificates, including the SPKI certificates [7]. Different from identity certificates, in the SPKI framework rights are given to keys, not to specific users or organizations.

As still another example of certificates, we mention the Pretty Good Privacy (PGP) system [3]. PGP provides a way to encrypt, decrypt, and sign data and exchange keys. Thus, by providing actual services, it is more than just a PKI. However, the main idea with PGP is that no strict, central authority based PKI is needed. Instead, the PGP user themselves create and extend the PKI they need. This is done by certifying other users public keys, i.e., signing trusted public keys with their own secret key. In this way a “web of trust” is created.

One possibility to utilize decentralized authorization certificate, such as SPKI certificates, is to use them on the grass root level just as PGP certificates are used. The users can authorize other users to access services they “own”. For example, the user of a PDA can authorize other users within a group to read non-private entries of his or her calendar. Such authorizations can be bound with various validity conditions, e.g., by limiting the

access for the duration of the current meeting only. Furthermore, the users can even delegate access rights, if the original authorizer has permitted delegation. Thus, for example, if you have received from your bank a certificate that authorizes you to sign electronic cheques that will be cashed on your savings account, you can delegate that right to your legal proxy. It is also possible to make limitations while making delegation, i.e., when you sign a certificate delegating the cheque signing right to your proxy, you can also make conditions on when the cheques may be signed, and what is the maximum amount of a single cheque.

In addition to the different usage, as illustrated above, there are also big differences in the way different certificates, needed for different kind of purposes, are obtained. In the case of ordinary X.509 type of PKI with hierarchical Certificate Authority (CA) structures, finding the right certificate is done using some central online server or by direct transmission of the certificate at connection set up. When using PGP, the desired public key either is stored locally on a machine or the device has to make a connection to a central PGP server to be able to find the desired pubic key. Currently, SPKI does not define any definite means of obtaining certificates, but leaves it to the specific application.

Since ad hoc networks are created on the fly between entities that happen to be at the same physical location, there is no guarantee that all nodes hold trusted public keys of the other nodes or that they can present certificates that will be trusted by the other parties. However, if we allow delegation of trust between the nodes, it is possible for nodes that already has trust relations to extend this to other members in the group. This will make it possible to reduce the number of manually interactions needed to set up security connections.

2.2 Trust distribution protocol

Now, if we further consider the ad hoc settings, and consider trust solely from the point of view of creating initial security contexts between the involved nodes, the following observations may be made. First, we may reduce the number of trust types by assuming that if Alice trusts Bob in our ad hoc networking sense, it means that Alice trusts Bob in all of the following senses.

- Alice trusts that it is safe to create secure connections with Bob, i.e., that Bob will take care of the keys agreed to create such a connection, and that Bob will not unnecessarily leak any confidential information received over such a connection.
- Alice trusts that it is safe to believe in Bob's recommendation, i.e., Alice trusts that if Bob says that he trusts Carol, then it is also safe for Alice to trust Carol.
- Alice trusts that Bob is able to successfully act in the roles required by the trust creation protocol (described below), and will act according to the protocol whenever requested.

We make the assumption that trust is always symmetric. That is, if Alice trusts Bob, then Bob necessarily trusts Alice.

The simplifications allow us to reduce the initial step of creating mutual trust within an ad hoc collection of nodes into a simple trust distribution algorithm and a corresponding distributed protocol.

Whenever a number of ad hoc nodes meet, an ad hoc network is created. Depending on the underlying link medium and routing protocol, the nodes sooner or later become aware of each other. However, initially the nodes are only aware of their direct, pre-configured trust relationships. Due to our simplifying assumptions, our trust relationships are essentially equivalence relations. Thus, in this setting creation of trust is a basic two step process.

- Given an arbitrary group of ad hoc nodes, first determine the equivalence classes created by existing trust relationships. Each of these sets are fully connected, i.e., all nodes within the set trust all others nodes (in that particular set).
- If there are more than one trust groups, manually create a new trust connection between arbitrary nodes in distinct groups. Each time such a new manual trust relationship is created, the groups merge due to our assumptions about the nature of trust relationships.

In many circumstances, the actual protocol can use multicast, but is still quite straightforward.

We acknowledge that these simplifying assumptions about the nature of trust involved are not valid in many circumstances. For example, symmetry is more believable in peer-to-peer communications (e.g., a network of wireless PDAs) than in a strictly client-server case. However, in an ad hoc setting, no better information may be available, and if we want to communicate, we will have to settle for something less than "trusted beyond all doubt". The user should, of course, be made aware of the level of security which is used, so he or she can make an informed decision wheter to use a service or not.

3. Securing ad hoc services

In order to illustrate how initial trust relationships together with the associated public keys can be used to secure ad hoc services in practise, we next describe a possible practical solution for securing Jini based ad hoc services. Even though we use Jini in our example, we do not restrict our description to any specific programming environment; however, we do assume that all the ad hoc nodes support a common computing platform, i.e., all nodes can download and execute programs written in a common language. In the example case, the code is Java code and the download mechanism is the standard Jini download mechanism, but it could as well be something else.

In general, whenever downloading code for execution, care must be taken. That is, the downloaded code must have restricted access rights the resources of the local machine. Similarly, if the code is downloaded on the demand of some other party but its executor, the requester must be able to trust that the executor faithfully executes the code. In the case of Java and Jini, the basic mechanisms for running the downloaded code inside a "sandbox" are already in place.

We next proceed to describe how trust relationships can be managed using downloaded code in a common computing platform. We first describe the basic arrangements, and then show how a key agreement protocol can be implemented with proxies. Section 4 describes our implementation in Jini.

3.1 Basic arrangements

Our security model is based on the use of public keys. We separate the problem of trusting public keys from the usage of trusted public keys. Typically, the user's machine has a file or a database where trusted certificates (or secure one way hashes of certificates) are stored. It is noteworthy that here we use certificates instead of plain public keys. The reason for this, as we later explain, is that we associate usage semantics with each public key.

The user can manage the database. Common access security techniques, e.g., passwords and local access control, can be used to protect the access to the database. New trusted certificates are added to the database either manually or after they are otherwise determined trustworthy.

The certificates in the database represent initial trust relationships. Any manually added certificates can include any semantics whatever. Certificates added as a result of running a trust distribution protocol depend on the exact nature of the keys distributed in the protocol. If plain keys are used, as we suggested for simplicity, the keys are considered to be implicitly certified good for creating secure connections within the current ad hoc network. That is, the keys may be used to secure communications within the ad hoc network with the provision that it is possible for some other node within the ad hoc group to eavesdrop and even modify the communications. On the other hand, the keys should not be kept good for any other purposes, due to the possibility of cheating nodes within the ad hoc network.

If the initial trust relationships do not suffice for a task in hand, any standard delegation mechanism can be used. For concrete approaches, see e.g. [7], [20] or [8]. However, care should be taken not to delegate permissions to any of the temporarily trusted keys, unless that is what is explicitly wanted. Even in such a case the time frame of the delegation should probably be quite short, or the delegation bound to the existence of the current ad hoc setting. On the other hand, if the user has already received other evidence on the validity of the desired subject key, the delegation may be more permanent.

3.2 Securing connections

Given that all of the nodes in the ad hoc network have public key pairs, and that all of the nodes consider the public keys of others' good for creating secure connections within the ad hoc network, any of public key based authentication protocols can be used. For example, it would be possible to use standard Internet key exchange [12] and TLS or IPSEC protection protocols [5, 16] to secure the actual service. However, we seek for a more flexible solution adapted to the ad hoc scenario. The extra freedom given in a Jini like environment facilitates this.

Unlike most standard approaches, we do not assume that the client and server necessarily share a large set of different symmetric key encryption or MAC algorithms. Instead, we assume that the client only has the following two pre-defined cryptographic capabilities.

1. By using a public key algorithm together with a one way hash function the client machine can digitally sign arbitrary data. The software program or hardware used for the signing is physically located in the client such that it can not be changed or manipulated by any hostile person. Any software used for the sign-

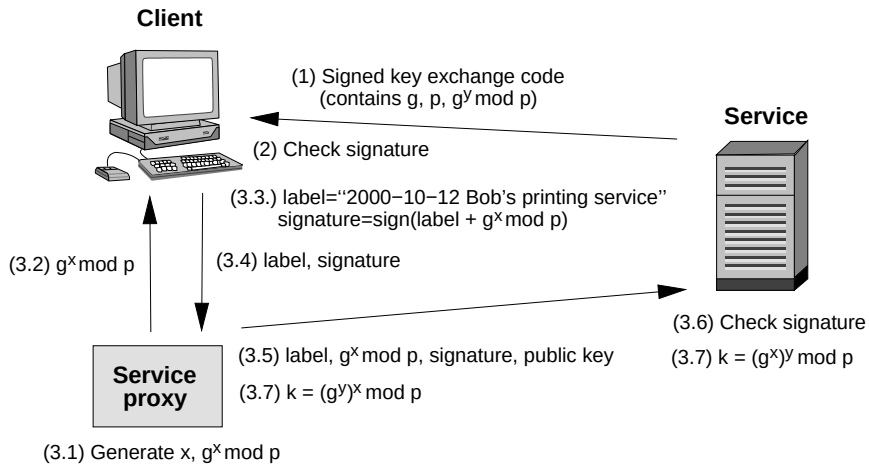

Figure 1: A basic proxy distribution protocol, with the proxy using Diffie-Hellman key agreement.

ing does not necessarily need to use the common computing platform.

2. The client can verify the correctness of a public key signature of arbitrary data. The algorithms used to sign the data are chosen among a quite small amount of possible algorithms. The software program or hardware used for verifying a signature is physically located in the client such that it can not be changed or manipulated by any hostile person. Any software used for the verification does not necessarily need to use the common computing platform.

Using these cryptographic facilities, a client can securely download a Jini proxy, and use the Jini proxy to execute an authentication and key management protocol on its behalf. This gives us complete freedom to utilize service dependent network security solutions.

3.3 Distributed key management

As an example, let us consider using the Diffie-Hellman key agreement protocol [6]. In our architecture, the key exchange code itself is signed by the server and downloaded by the client in the form of a Jini proxy. The signature of the key exchange code is checked prior to the real key exchange. Hence, authenticated information about the server's public key value is available to the client. This means that we can reduce the number of messages needed between the client and the server.

To illustrate, the following sequence of steps can be used to create a secure connection between a Jini based client and a service.

Preliminary steps by the Server. Before any communications are started, the server prepares a secure proxy that the clients can download. The server also signs the proxy, thereby allowing the clients verify the integrity and origin of the proxy before the proxy is started. The proxy code typically includes a public key corresponding to a private key held by the server. The integrity and authenticity of the public key are implicitly checked by the client as it authenticates the proxy code.

- A server that wants to offer a secure communication service has a computer program written in the computer language of the common platform, i.e., Java. In Jini terms, the server has a service proxy. The proxy contains the necessary algorithms and methods needed for doing authenticated key exchange with the server. Furthermore, the proxy contains the necessary algorithms needed to encrypt and protect all data sent between a client and a server in a secure service session. However, the proxy does not necessarily need to contain all code needed to perform cryptographic computation. Instead, it might use APIs defined in the common platform, if feasible.
- The server digitally signs the proxy using its private key. The signature is calculated using the predefined algorithms and formats described in Section 3.2 above. This ensures that the client will be able to verify the signature.
- The server packs the signed code together with the signature, and optionally includes also one or more certificates that certify the public key of the server.

Looking up, Downloading and Starting a Service Proxy. In Jini, and similar environments, communication is started by a client looking for a service. Once the service is found, the client downloads the service proxy for execution. In our case, the service proxy is verified before starting its execution.

- A client searches for a service. The exact nature of search is of no importance here, but one possibility is to use the Jini lookup service.
- When the client finds the service and wants to use it, it downloads a proxy corresponding to the service, together with the signature and optional certificates.
- The client checks the signature of the downloaded package. If the client holds a trusted public key that corresponds to the signature, or the client trusts some of the public keys contained in the certificates included, then the client treats the downloaded code as a trusted code.
- If the verification is successful, the client runs the downloaded code using the common computing platform. Any runtime restrictions may be placed as appropriate; especially, the downloaded code doesnot need to be able to communicate with anybody else but the designated server.

Since the service proxy is used here only for creating a secure connection with the server, and actually accessing the service, the trusted key used may be one of the keys that the client temporarily trusts. In such a case, the service must be considered an anonymous service available in the current ad hoc setting. For practical purposes, such as securely accessing a nearby printer, this is probably perfectly adequate. On the other hand, if the client needs to access a permanently trusted service, the service key must be properly certified.

Authentication and Key Agreement. The downloaded code can ask the client to create a signed ticket. The client may refuse to perform any other cryptographic functions. The ticket creation function takes as its input some arbitrary data and outputs a ticket, which basically is a digital signature of the data plus a specific label added by the client. The label is needed in order to make sure that the resulting item is always recognized as a ticket. The client might also return a certificate containing a public key that can be used to verify signatures made by the client.

The ticket label typically designates the service the client has requested the proxy for, and a time stamp. In the case when the client device is a personal device and the signature key can be used for non-repudiation services, the label should be displayed to the user before signing (or otherwise verified to be acceptable).

Thus, the proxy and the server can authenticate each other as follows.

- The proxy performs authenticated key exchange with its origin server. The actual protocol used can basically be any standard authentication and key exchange protocol. When performing the key exchange, the service code can request tickets to be signed by the client. If the authentication succeeds, the proxy sets up a secure communication link with the server.
- The service provider writes the security code that is used to create a secure connection, so it can implement the authenticated key exchange and communication protection as it wishes, but should follow good cryptographic principles.

In the actual authentication protocol, the protocol can be somewhat simplified by the fact that the downloaded proxy code may already contain authenticated information. For example, if Diffie-Hellman key agreement is used, the public key exchange value of the server can be contained in the service code. Hence, the key exchange can be performed with one single transmission from the client to the server and we save one transmission. This is illustrated in Figure 1.

4. Authorization in Jini

One of our goals was to study how to add authorization and delegation to Jini. Therefore, we implemented a security framework providing authorization to Jini based services and applications.

The users can receive authorization to use some Jini service (for example, a printer) from the administrator of the service. The authorization is issued as a SPKI certificate [7] written to the user's key. The certificates and the user's private keys are stored in her computer.

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

4.1 Changes to the Jini operation model

The default behavior of a Jini application and a Jini service is shown in Figure 2, on next page, where an application prints a document using Jini. Before the invocation is possible, a number of activities must take place.

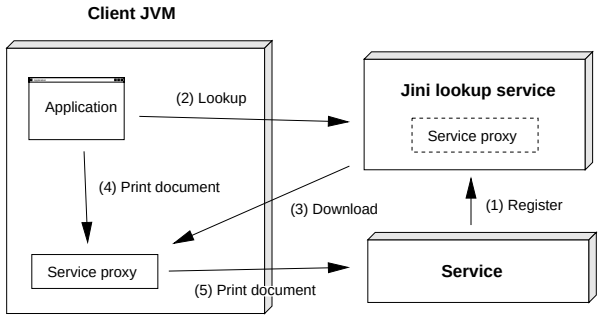

Figure 2: Default behaviour in Jini service lookup.

1. The service initializes its proxy object and registers it with the lookup service.
2. The application, wishing to use the service, queries the lookup service for services providing the desired functionality.
3. The serialized proxy object is returned to the client, and the corresponding Java byte code is downloaded.
4. The application calls some method on the proxy object, requesting it to do whatever the service does.
5. The proxy sends the request to the service.

With our authorization mechanisms, the picture is a little bit different (Figure 3, below). Note that in this case, the security features are completely transparent to the client application.

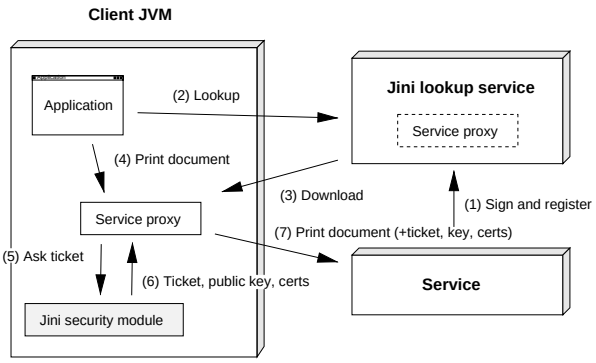

Figure 3: Jini service access with authorization.

When accessing the service, the four first steps are almost identical with the default Jini behavior.

1. Before sending the proxy to the lookup service, the service signs the proxy code and state. The proxy now also contains the key exchange code, and perhaps some key exchange parameters as well.
2. The applications contacts the lookup service, and performs an appropriate search, as before.
3. The proxy object is downloaded, and the signature of its code and state are checked. The service's public key and any additional certificates are stored for future use.
4. Application calls a method on the proxy object, as before.
5. When the proxy receives a request, it needs to open a secure channel to the service. It asks the Jini security library to sign a ticket for it.
6. The Jini security library first checks if the client application is allowed to access this kind of service, and if it is, writes a ticket (in this case represented as a SPKI certificate) and signs it using the user's private key. The proxy can also ask the security library to search for any additional authorization certificates which should be presented to the server.
7. The proxy contacts the service and runs the authentication and key agreement protocol. It then sends the client request (in this case, the document to be printed) and any relevant certificates.

   The service gives the user's public key, certificates, and the requested action to a "trust management engine", which checks the certificate chains. If the authorization is ok, the service performs the requested action.

Since our initial application is a personal calendar, we also wanted to authenticate the server using some human-recognizable identity (i.e., a name). Therefore, the proxy can also contain additional certificates, which are checked by the application before performing step 4. We argue that this is best left to the application, since the trust models are very application specific (cf. [24]).

5. Related work

The Jini architecture doesn't include any security in addition to the normal Java 2 security facilities (for protecting the JVM from malicious proxy code). The solution proposed by Sun is the RMI security extension [26], currently in specification phase. A concrete implementation is expected to be included in JDK 1.4 ("Merlin"), which will probably be released last quarter 2001 (public betas earlier).

In the RMI security extension, a trusted component (not downloaded from the network) is responsible for actually opening the network connections and performing the authentication protocol, so this might limit the protocol independence offered by Jini. Furthermore, the ad hoc trust establishment problems remain the same.

6. Conclusions

One of the main security problems in ad hoc networks is obtaining the necessary trust relationships. In this paper, we have used a simple trust model, and shown how it can be applied to secure ad hoc services. We suggested a trust distribution protocol that minimizes the number of manual interaction needed when setting up the necessary trust relations. Furthermore, we have shown a "minimal" pre-configuration solution for securing the communication of an ad hoc service. The technique allows establishment of authenticated connections without allowing undue access to the client's private key. We have suggested changes to the Jini operational model that makes it possible to add authorization to Jini. Our example implementation uses Java and Jini, but the same principles can be used with other service discovery techniques.

Our trust distribution protocol can be combined with other trust establishment procedures. For example, some ad hoc networking devices have communication channels which are inherently reasonably secure (such as short-range infrared or physical contact). Stajano and Anderson discuss such situations in [24] and [25]. These could of course also be combined with our Jini solutions.

Another interesting topic is the human interface aspect of ad hoc security. Since the level of security provided can vary a lot, a lower level might be acceptable for some use but not for other cases. The user should be aware of the level of security provided, so he or she can make an informed choice based on the requirements of the task in question.

Acknowledgements

We would like to thank Jonna Särs and the anonymous reviewers for their valuable comments and suggestions.

References

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

[2] Bluetooth SIG. *Specification of the Bluetooth System, version 1.0 B, part E: Service Discovery Protocol (SDP).* December 1999. Available from http://www.bluetooth.com/developer/specification/specification.asp.

[3] Jon Callas, Lutz Donnerhacke, Hal Finney, and Rodney Thayer. OpenPGP Message Format. RFC 2440, IETF, November 1998.

[4] Lucas Cardholm. Building Trust in an Electronic Environment. In *Proceedings of the 4th Nordic Workshop on Secure IT Systems (Nordsec '99)*, pages 5–19, Kista, Sweden, November 1999.

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

[6] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[7] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas and Tatu Ylönen. SPKI Certificate Theory. RFC 2693, IETF, September 1999.

[8] 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.

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

[10] Erik Guttman, Charles Perkins, John Veizades, and Michael Day. Service Location Protocol, Version 2. RFC 2608, IETF, June 1999.

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

[12] Dan Harkins and Dave Carrel. The Internet Key Exchange (IKE). RFC 2409, IETF, November 1998.

[13] ITU-T Recommendation X.509 (1997 E): Information Technology – Open Systems Interconnection – The Directory Authentication Framework. June 1997.

[14] Audun Jøsang. *Modelling Trust in Information Society.* PhD Thesis, Department of Telematics, Norwegian University of Science and Technology, Trondheim, Norway, 1998.

[15] Audun Jøsang. Trust-based decision making for electronic transactions. In *Proceedings of the 4th Nordic Workshop on Secure IT Systems (Nordsec '99)*, pages 246–268, Kista, Sweden, November 1999.

[16] Stephen Kent and Randall Atkinson. Security Architecture for the Internet Protocol. RFC 2401, IETF, November 1998.

[17] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography.* CRC Press, October 1996.

[18] Microsoft Corporation. Universal Plug and Play Device Architecture, version 1.0. June 2000. Available from http://www.upnp.org/.

[19] M. Mühlfelder, U. Klein, S. Simon, and H. Luczak. Teams without Trust? Investigations in the Influence of Video-Mediated Communication on the Origin of Trust among Cooperating Persons. *Behaviour and Information Technology,* 18(5):349–360, September 1999.

[20] Pekka Nikander. *An architecture for Authorization and Delegation in Distributed Object-Oriented Agent Systems*. PhD Thesis, Helsinki University of Technology, March 1999.

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

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

[23] Gustavus J. Simmons and Catherine A. Meadows. The role of trust in information integrity protocols. *Journal of Computer Security,* 3(1):71–84, 1995.

[24] Frank Stajano and Ross Andersson. The resurrecting duckling: Security issues in ad-hoc wireless networks. In Bruce Christianson et al., editors, *Security Protocols, 7th International Workshop Proceedings*, Cambridge, UK. Lecture Notes in Computer Science, volume 1796, Springer, April 1999.

[25] Frank Stajano. The resurrecting ducking — what next? In Bruce Christianson et al., editors, *Security Protocols, 8th International Workshop Proceedings*, Cambridge, UK. To appear in Lecture Notes in Computer Science, Springer, April 2000.

[26] Sun Microsystems. Java remote method invocation security extension. Early look draft 3, http://java.sun.com/products/jdk/rmi/, April 2000.

[27] 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.