Master's Thesis in Computer Systems
8th April 2002

# Differentiated security in wireless networks

Andreas Johnsson

Information Technology
Department of Computer Systems
Uppsala University
Sweden

Instructor and examiner:
Mats Björkman

**Abstract**

This report presents a three step solution to the differentiated security and self-configuration problem in wireless network where users are able to come and go as they wish. Differentiated security is about giving different user's different access rights towards a surrounding infrastructure. Access rights can in some systems be bought and in others they may be related to a users location or real life status. Self-configuration is about minimizing the configuration effort needed by network users.

First a generic architecture is presented, which provides an abstract solution without any requirements on specific techniques or tools. Thereafter a prototype specification and a prototype implementation are derived from the architecture. They show that the desired system is feasible and relatively easy to construct, using well known tools for resource discovery, security and address handling.

# Preface

This thesis is written as part of my studies at the Mathematical and Natural Science Program, Uppsala University, Sweden.

The thesis is written as part of the CONNECTED[1] project. My advisor and examiner is the senior lecturer Mats Björman, Department of Computer Systems (DoCS), Uppsala University. I would like to thank him for valuable input and support. I would also like to thank the CoRe group who has hosted and supported me by during this work.

Finally I would like to thank Malin and my family, not for their great expertise in the area of computer communications, but for their support in all other ways.

[1] sponsored by Ericsson Research and "Svenska IT Stiftelsen" (SITI)

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background and scenarios

In more or less public networks it is desirable to use different security strategies and to give different access rights depending on the identity of a user. This need for differentiated security is especially true in dynamic wireless networks where nodes can join and leave as they wish. In a group of wireless nodes some users should be considered more reliable than others, and therefore be given a larger set of access rights towards the surrounding infrastructure.

The following is a practical example. An employee from company E is visiting a meeting at company N. Company N provides a wireless network for the participants. An individual from company E brings his or her laptop and thereby has the opportunity to join the wireless network supplied by N. Now suppose that the employee from company E needs to fetch a document from his or her home network and then print it on a local printer supplied by company N. If this fetch and print scheme should work without risk for computerized fraud performed by either company several security aspects has to be considered and designed to work together. They are:

- company N must for this occasion give the visiting person from company E access to the Internet
- company N must for this occasion give the visiting person from company E access to the local printer
- the employee from company E must not be able to access anything private in the network of company N
- the employee from company E must be sure of the fact that company N can not access anything in his or her laptop
- the employee from company E must be sure that company N can't intercept, eavesdrop on or steal the communication channel established

between the visiting employee from company E and his or her home network

Another thinkable scenario where a highly dynamic public wireless network may exist is on an airport. The airport network is accessible by travelers from all over the world, hence security is an essential component. Only a subset of the wireless network nodes should be allowed to use the printer and other services. This access right differentiation is due to the fact that use of a service could be an expense for the service provider, or just because that the service provider can make money on the service it provides. Just like the previous example different users has gained or will be given different access rights. The access rights in this scenario could be built upon payment. The security aspects in this example are the same as in the previous one.

Another important feature that the dynamic wireless networks in both scenarios should provide is easy client configuration and setup. This means that a client should not need to reconfigure its laptop manually every time the client enter or leave different wireless domains.

## 1.2 The solution and report layout

This report presents a three step solution to the problems and scenarios discussed in 1.1. The first step is a presentation of an architecture giving a generic overview on how a system could function and what abstract tools that are necessary to meet the stated requirements. The next step is a prototype specification which builds on the architecture and is more concrete in choice of tools and techniques. The third and last step presents and describes an implementation. This implementation is only a prototype and should not be used in a real system. The implementation builds on the prototype specification but lacks several features.

Finally there is an appendix which presents different communication and security techniques. The appendix should be a guide for readers that want more information on some specific topic or technique used in the architecture, the specification or the implementation.

## 1.3 Reader requirements

The reader should be familiar with Java and its documentation and various basic tools. The reader must have basic knowledge in computer communications, especially the Internet and TCP/IP networks, equivalent to a first course in computer communication. In addition to this the reader must posses basic knowledge in theoretical computer security. The following concepts should be known to some extent: authentication, integrity, confidentiality, non-repudiation, certificates (especially X.509), cryptographic hash

functions, digital signatures, encryption and decryption, handshake protocols, basic key agreement, message authentication code, public key cryptography, secret key cryptography and various security attacks. These concepts are well presented in [1], for example.

If the reader intend to setup a demonstration it is preferable to have knowledge about Solaris, Linux configuration and perhaps printer configuration under Unix.

## 1.4 Terminology

### 1.4.1 Mobile node

The meaning of a mobile node is the same as the meaning of a client, host or a user in the report.

### 1.4.2 Priority, Access right or Access right list

A mobile nodes access rights gives information about what service to use and for how many times/for how long a mobile node can use the specific service.

An access right list is a list of access rights.

The word priority can be interpreted in at least two ways. One way it can be interpreted is how prioritized one node is compared to another. An example of this is if two nodes accesses the same service at the same time. Then the node with higher priority is served first, while the other node has to wait.

The way priority is interpreted in this report is the following. The priority of a node is the set of all access rights possessed by that node. A high priority means that the node can access and use many services while a low priority is the opposite.

## 1.5 Trust issues

In the two scenarios in 1.1 it is assumed that the network used by mobile nodes can't be trusted. This means that a mobile node can't rely on having security sensitive code downloaded from a resource location service. Security sensitive code is equivalent to all code that deals with a mobile node's private and symmetric cryptographic keys.

As a consequence of this fact, end-to-end cryptographic applications that must be used to create virtual private networks, IPSec tunnels et cetera must be brought to the network by the mobile user itself. Trust between the mobile node and its home network must have been pre-established.

Pre-key distribution of several public and one symmetric key per mobile user in the proposed architecture is assumed.

# Chapter 2

# Architecture

## 2.1 Introduction

To create a wireless computer system where mobile nodes should be able to come and go without having to perform too much of configuration, many components are needed (see figure 2.1). First of all a resource location server might be present. This server should make it possible for mobile nodes to download executable code which can configure the mobile node, act as a proxy towards some system service or just implement something the mobile node needs to execute at a specific moment.

There are several reasons for having a resource location server instead of letting all nodes be pre-configured with the necessary software for using the system. One major reason is that nodes without system knowledge should be able to use it. Another reason is that all code that is downloadable from the system might not fit into a laptop at the same time. A third reason is the problem of updating the system software. If the software is distributed among all potential users the system must be backward compatible with all previous versions.

Security is an important issue in this kind of dynamic system. This is true because the mobile nodes must execute untrusted code (code downloaded from the system) within it's own environment. Security is also important since a mobile node shouldn't be able to use whatever service it wishes. Restrictions must make sure that only high priority users can access services that must be protected for some reason. All communication inside the system must be encrypted to protect everyone involved.

If mobile nodes should be able to come and go there must exist a dynamic address allocation server, since all nodes in the system need some kind of address to be able to communicate. To protect the system itself and its users all mobile nodes must authorize themselves to get an address. Clients should also be able to authenticate the dynamic resource allocation server. In the process of authentication the Trusted Authority and the Cryptographic code

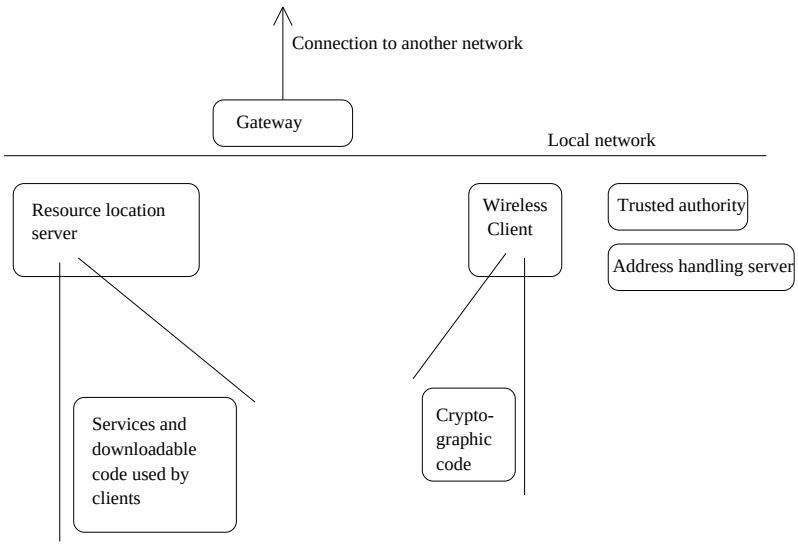


Fig. 2.1: System architecture overview

(see figure 2.1) are used by the client.

To connect the local network to another network a gateway of some sort is needed. This gateway could have different functionality depending on what restrictions the system administrator what to employ.

A proposed architecture which meet the requirements stated in this section and the requirements stated in 1.1 is described in the following sections. First every component in figure 2.1 is described in a bit more detail. Thereafter the most important services are presented. The chapter ends with a short survey and description of the security within the system.

## 2.2 System components

### 2.2.1 Resource location server

The most central entity in the architecture is the resource location server. This server contains several registered services of various kind. The server should allow mobile users to find the services they want to use and to download the corresponding code for local execution. The resource location server should let the mobile users to search for services depending on their use, their price, their owner and other attributes that may be important to a client. The owner of a service registered in the resource location server can be either the system itself or a high priority user.

### 2.2.2 Downloadable code

Because of the fact that downloadable code is used it is convenient to use a common execution platform that all users in the system share. When downloadable code is used in an unfamiliar system several security issues arise. Every client that execute code downloaded from the resource location server must be able to verify the origin and integrity of the code. If the client can verify the origin of the code it also knows whom to "blame" if it runs into trouble. Also, a client should not share any secrets with downloaded code. The client must be able do decide what kind of actions the downloaded code should be allowed to perform on the local computer.

### 2.2.3 Address handling

Every node entering the system needs a network address, an address to the resource location server and other network specific parameters to be able to use the network. This information is given by a central or distributed node to clients that can authorize themselves. Clients should also be able to authenticate the server. Authorization in the host configuration procedure is an important mechanism to avoid malicious nodes in the system. If a malicious client can't get a valid address it's impact on the system is at least reduced.

### 2.2.4 Gateway

The local network can be connected to other networks using one or many gateways. Each gateway should be implemented as a firewall to be able to define what kind of network traffic and which users that should be allowed to access addresses outside the local network. The firewall also has the purpose of protecting the local network from outside security attacks.

### 2.2.5 Trusted authority

To be able to establish trust and secure communication between users and the system in a scalable way there must exist a trusted authority. The trusted authority's job is to ensure a mobile node or a service that a certain cryptographic key or certificate really belongs to a specific entity. If everyone in the system trust the trusted authority, every node can create secure communication without fear of an identity fraud. When using a trusted authority it is presumed that the trusted authority and its clients have exchanged identification material before the client actually use the trusted authority. This is a limitation, but most everyday security (and computer security) builds upon trusted relations.

### 2.2.6 Cryptographic code

The last important component is the code where the mobile node uses its private or symmetric cryptographic keys. This piece of code must not be downloadable from a resource location server. The client should be pre-configured with that code. This is due to the fact that the network is not trustworthy. If you can't trust someone, you can not download security sensitive code from there. The fact that a node must be pre-configured with code is a limitation. However, to download one piece of code before entering the system is not that time consuming for the client.

If the client wishes to establish some kind of cryptographic tunnel with a home network it has to bring these components and trust relationships itself, due to the fact stated above.

## 2.3 Services

If we assume that a resource location server should be used, then all services should be defined in a way which meet the requirements of access right restriction. A service which let users buy, or in some other way gain access rights, must be defined. Furthermore definitions of services which manipulate the gateway must be made, if the system administrator only want a subset of local nodes to access nodes on the outside.

Every service is registered and downloadable from the resource location server. The system can register new services at any time. Also high priority users can register new services. The services are divided into two categories, system services and user services.

### 2.3.1 System services

#### Ticket and payment service

To accomplish differentiated security in the network, every mobile node in the system needs to possess a priority or access right. The priority should give information about which services a mobile node is allowed to use. Priority could be based on a payment system, real life situations or a client's relative position towards a point in space. An example on real life situations is that an employee at a company should be able to access more services than a visitor, i.e. the employee has higher priority than the visitor. An example on position based priority could be that clients in a special area are allowed to use the printer, while others are not.

Priority (or access rights) can be implemented using a ticket system. In the architecture there should be a ticket service which has an access control list which defines whom can access what. When a client needs to access a service which requires any access right it has to be granted a ticket from

the ticket service. The ticket is thereafter transferred from the client to the service the client want to use.

The ticket should contain a cryptographic verification field, so that the service receiving it can verify that it is valid. The verification fields should be written by the ticket service, since the ticket service should be the only node in the system which knows whom can access what. The ticket passing between the ticket service, the client and the service must be confidential to avoid fraud.

The reasons to use one ticket service instead of letting each service have its own verification mechanism are many. If access rights are based on payment it is more comfortable for the user to give its credit card number away to one entity (the ticket service), since a credit card number can be misused. From the service point of view it's easier to verify the origin and validity of a ticket compared to go through the rather complex issue of payment over the Internet.

If the service is running on a platform with little memory it can be difficult to hold all access right information itself. Then it is, again, easier to verify a signed ticket.

Yet another reason to have a ticket service is that tickets could be valid in many different local wireless networks. It is easier to coordinate whom can access what between different networks if the information is somewhat centralized[1]. Then the following scenario is possible: a user could buy access right to a printer in a local network A. The user then moves to another local network B, which also have a printer. To be able to print a paper in B the user could use the very same ticket as it did in A.

### Firewall service

The system must supply a service that can modify the firewall configuration. If a mobile node has the access right, it should be able to communicate through the firewall. The communication is supervised by the firewall. For example, the user could be allowed to use only HTTP or POP, all depending on the specific users access rights.

### Interface service

The last essential system service is the user interface service, which could be a graphical user interface. This service should make the system easy to use. The user should be asked to choose between different user interfaces, if there exists more than one. The main reason for having more than one user interface is the different capacity of mobile nodes (laptop, pda, mobile phone and so on).

[1] even if the information is centralized the ticket service itself could still be distributed to avoid a single point of failure

### 2.3.2 User services

Services not classified as system services (ticket service, payment service, firewall service and interface service) are called user services. These are the services a user is actually paying for while the system services is essential for the system survival. An example on a user service is the printing service which let users print documents on a local printer. Exactly what kind of user services there should be present in a specification of this architecture depends on what users expect from it.

## 2.4 Additional security and trust

All communication between clients and services must use a secure channel. This is necessary, for example, in the case of ticket passing where confidentiality, integrity and anti-replay attack security is needed. Encryption is also needed in other cases, because we do not want eavesdropping. Communication with outside nodes is not secured automatically by the system.

If a user needs to setup a secure communication channel with the outside world it has to provide the necessary tools itself.

Authentication and secure communication setup require that every node in the system posses a public/private key pair, where the public key should be given securely to the trusted authority.

# Chapter 3

# Prototype specification

## 3.1 Introduction

The specification assumes a TCP/IP wireless network between clients and the system itself. For resource location Jini (see A.4) is used even if there exist other resource location services (see A.3). Security is implemented using JSSE (see A.7.5). All components used in the specification are chosen with care at least taken to implementation issues and hardware/software resources at Uppsala University, Sweden.

## 3.2 Overview

The central entity in the prototype architecture is the Jini lookup service in combination with a web-server. Clients and other services can automatically download Java classes from the web-server and the corresponding serialized instance of the Java classes needed from the Jini server. When both the Java class and the serialized instance of it is possessed, the client can execute whatever public methods the object has to offer.

Clients need an IP address to be able to communicate over the TCP/IP network. These addresses are supplied by a DHCP server. To be sure only high priority users can access the Internet there should be a firewall at the gateway connecting the wireless LAN to the Internet.

One of the two most important Jini services is the ticket service, which signs tickets to authorized clients. These tickets are used whenever a client wishes to make use of a Jini service that requires authorization. Every service supported by the architecture can verify whether the ticket transferred from a client is accurate or not.

The second of the two most important Jini services is the trusted authority. In the architecture the trusted authority was described as a component, but it will be easier to implement it as a service. The trusted authority knows the public key of every client and every service. Every node in the

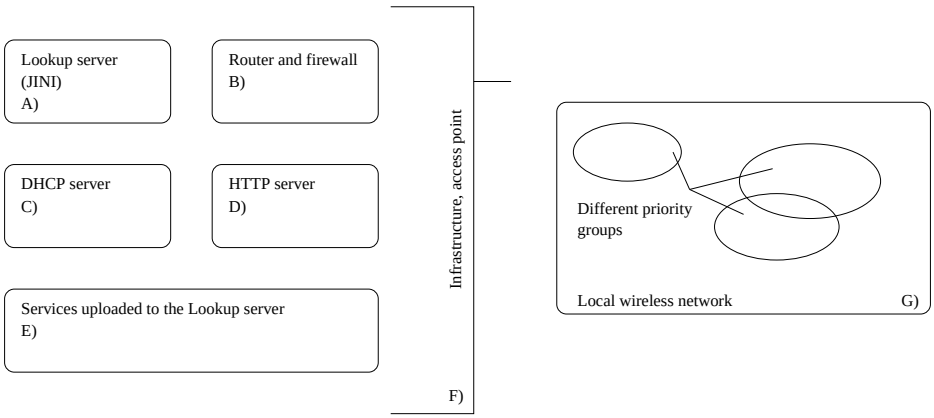


Fig. 3.1: System specification overview

system trusts certificates, which binds public keys, node names and node identifiers together, signed by the trusted authority. The trusted authority makes it possible for both services and clients to trust each other, or at least know uniquely to whom they are talking to and to establish the necessary secure communication channels. Encrypted communication must be used in all communication between clients and services.

To be more concrete, the system is constructed using seven key components (see figure 3.1). These are the Jini lookup server (A), the router and firewall which connects to the Internet (B), the DHCP server (C), the HTTP server (D), a collection of services (E), access points (F) and several mobile nodes with different access rights (G). Every component is described in detail in the following subsections.

### 3.2.1 Jini lookup server

The lookup server in the specification has the purpose of letting nodes find the services they are looking for. Every node that has a service to let other nodes use, registers their service at the lookup service. Other nodes can then ask the lookup service for that kind of service. The lookup service used within this prototype specification is Reggie. Reggie is designed and implemented by Sun Microsystem[1]. All essential services supported by the system are registered here. All nodes can find their way to a lookup service by using the DHCP Jini option (described later) or by broadcasting.

Reggie can not control who registers services in it. Another problem with Reggie is that searching for services is in some cases poor. Yet another problem is that the Reggie implementation is not distributed. In some unreliable systems (such as a wireless LAN) you would like to have an easy tool

[1] http://www.sun.com

for making all registered services distributed over all Jini lookup services supported by the system.

However using Reggie as the lookup service is a good idea - since it is designed to work together with Java code. Java code is portable to almost all current operating systems. Of course the mobile nodes need to be able to execute Java code. This limitation is real only for small units, such as cellular phones.

### 3.2.2 The firewall

The firewall (router) handles forwarding of IP packets to and from the local wireless network. Every node should have a globally unique IP address, which implies that no network address translation [2] is necessary. Of course this could be changed if there is a lack of IP addresses. Linux, as well as other networking operating systems, has several tools making network address translation a relatively easy task (see A.7.1).

The firewall has two main security purposes. The first is to protect the local wireless network from malicious users on the Internet. Protecting the local wireless network is about how to secure each individual mobile node and all essential architectural components from attacks of various kind [1][16]. The second main purpose is to prevent local mobile nodes with bad access rights to access the Internet.

To access the Internet from the local wireless network nodes need a priority higher than or equal to some border priority. If a node possess this minimum priority a Jini service can be found, downloaded and used. This Jini service will, through a proxy, modify the firewall configuration on the computer running Linux using, for example, the *ipchains* command (see A.7.1).

### 3.2.3 The DHCP server

A DHCP server is used to give mobile nodes their IP address and other network information needed to allow a client to use an unfamiliar network (see A.1 and A.2).

Among all the necessary parameters given to the mobile node, a chain of IP addresses to lookup services is given, using a new DHCP option. The option contains an extra field for every IP address specifying the resource location service version and vendor. This is due to the fact that all resource location servers are not Jini implementations, and all Jini implementations will in the future not be Reggie.

Jini supports broadcast for finding lookup servers, so why use this DHCP option? There are at least three arguments for implementing this new DHCP option. Those are:

- the system may want to control packet load on each lookup server

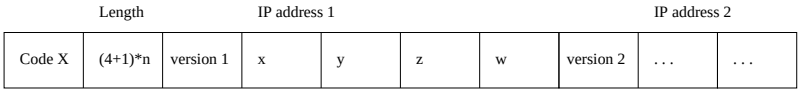


Fig. 3.2: DHCP Jini option

- in a large system it could be inefficient to use broadcast, network traffic is heavy anyway
- the lookup server could be out of range for broadcast packets

### DHCP - Jini option specification

The option has number X (unspecified) and is organized as shown in figure 3.2.

The code field gives information about which DHCP option the following octets should be parsed as. The length of this option is (4+1)*n octets. The minimum length is 5 octets, when leaving the code and length slots outside of the calculation. Thereafter a list of version number and IP address appear. The version number is a value between 0 and 255, letting 1 be the current version of the Reggie lookup server. The value 0 means an unspecified lookup server.

### DHCP security

The DHCP server in this prototype specification should use an option that force mobile nodes to authorize themselves before achieving network information (see A.7.7). An authorized mobile node share a unique symmetric key with the DHCP server, which let the node and the DHCP server authenticate each other. It is important that there occur a mutual authentication process. Otherwise the mobile node could get network information from a malicious DHCP server. And from the other way around, the DHCP server only wants to allocate IP addresses to authorized mobile nodes.

To require that a node and the DHCP server share a symmetric key before the actual handshake starts is a limitation. The pre-distribution of keys can for example be made with some method described in section A.7.3.

Can only authorized clients get an IP address within a network that uses the DHCP server with the authorization option? No, a malicious node can pick its own IP address, without using DHCP. To at least ensure that the malicious node can't make use of its stolen IP address outside of the local network some kind of firewall or router might be present. This router only route packets from IP address that has been given to authorized clients, using the DHCP process.

A malicious node could perform a spoofing attack. Using the spoofing attack a malicious node is able to take over and use an authorized clients IP

address. How those problems are handled is discussed in section 3.9.

### 3.2.4 HTTP server

The HTTP server used is the one supplied with the Reggie system. It can't be used for anything else than downloading of files, but that is all that is needed.

### 3.2.5 Registered services

All essential services supported by the system are registered within the Jini lookup service. Of course they can be spread over multiple nodes. There are two kinds of services, system services and user services. The system services are the services essential for making the system work (i.e. ticket service) while the user services are the services the user actually are paying for. All services are described in a separate subsection.

Registered services are split up in two components. The class file and an instance of the class file (a Java object). The class files are located in the HTTP server and the instances of the class files are located in the Jini lookup service.

### 3.2.6 Access points

All mobile nodes that wish to use the infrastructure must possess a radio network interface (see A.6). These are configured (by the user) and are thereafter ready for communication between the mobile node and an access point.

### 3.2.7 The mobile cloud

The mobile cloud contains all mobile users currently using the infrastructure. They all have different priorities towards each other and towards the infrastructure. Every node has a specific access right list located at the ticket service (described below) which has been obtained either by payment or by some priority in real life (for example employees at a company has higher priority than visitors).

Every node has an initial bootstrap Jini program. This bootstrap program is used to download a user interface from the Jini server. From this user interface the mobile node can call various services supported by the infrastructure. More on bootstrapping in later subsections.

Every node also has in its initial state a shared symmetric key with the DHCP server and a public key corresponding to a trusted authority. All nodes should have given the trusted authority its public key. More on security in later sections.

## 3.3 System service description

### 3.3.1 Ticket service

The ticket service is designed to sign tickets (basically a certificate) used for authorization of mobile nodes. The ticket service has an access control list where it is told what services can be used by whom. When a mobile node asks for a ticket it has to specify what services it wants to use. If the mobile node really is authorized to use that service the ticket service grants the mobile node a signed ticket.

This signed ticket is thereafter shown to the service the mobile node wants to use. The service verifies that the ticket really is signed by the ticket service and that it really is for using that service. If it is, the mobile node can go ahead using that services methods. If not, the mobile node gets back an error message.

The ticket service should be distributed and use threshold security to avoid problems with compromised hosts [21].

The idea of having a ticket service is derived from [22], see A.7.6 for more information. There is a difference between the solution in this thesis and the ticket signing tool in [22]. The difference is that the ticket service in this solution is a stand alone unit, while in [22] it is something within the client. The reason for having it as a stand alone unit is of course the fact that a client should not be able to sign tickets itself to be used in various services.

### 3.3.2 Trusted authority service

The trusted authority (TA) is responsible for holding public keys of all users in the system. Its main purpose is to distribute certificates to nodes that needs them in order to setup a secure communication. Every node in the system must trust this authority, otherwise no one in the system will be able to establish a trust relationship.

The trusted authority should be distributed and use threshold security to avoid problems with compromised hosts [21].

### 3.3.3 Payment service

The payment service is used when nodes wants to buy access rights in the system, for example when a mobile node needs to print a document. The client pays via the payment service. The payment results in a change of the access control list at the ticket service. Now the client can be granted a ticket for the specified service through the ticket server. Exactly how the actual payment service is designed and implemented is out of scope of this thesis. But still it is an important component in a real system. If the architecture presented in this thesis would evolve into a real system the

security protocol SET [1] might be used for transferring credit card numbers and other important parameters between card holders, banks and merchants.

### 3.3.4 Firewall service

The firewall service is used to modify the firewall that guards the local wireless network. This service is probably located at the same node as the firewall itself. When a high priority node wish to communicate with nodes on the Internet it has to be given a ticket from the ticket server that specifies the client IP address, the lease time, type of communication and all the other parameters used in ordinary tickets. The ticket is shown by the client to the firewall service. If verification is okay, the firewall service opens up the firewall for the type of communication specified in the ticket.

### 3.3.5 Cryptographic code service

The cryptographic code service is one or several objects that is used when a mobile node wants to sign, decrypt or use its private key in any other way. This code should be pre-installed at all clients to avoid malicious use of a client's private key. Basically this code is the JSSE package. The idea of a special segment of secure cryptographic code is taken from [22].

### 3.3.6 Interface service

The interface service is a service that is downloaded at bootstrap. The service can be a graphical user interface or a textual. The purpose is to make it easy for the user to handle and call the different services supported. Exactly how the user interface should look and feel is of no importance to the system itself.

### 3.3.7 Service lookup

The service lookup service is a way for mobile users to get lists of all services currently supported by the system. This service could be combined with the interface service.

## 3.4 User service description

### 3.4.1 Group meeting service

This is a service making it easier for parties to establish secure communication within a group. The actual service that is downloaded is a service for secure group communication. Several protocols exists (see A.7.2). The keys agreed on when using this service can later be used in secure FTP or IRC sessions. The service require that the nodes using the keys, derived from the

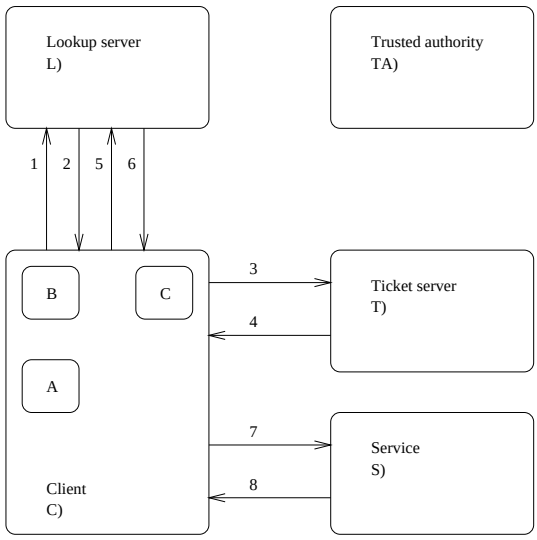


Fig. 3.3: Message transfer when accessing a service
A) Cryptographic code B) T proxy C) S object or S proxy

group meeting service, can use them. The service itself should not supply secure applications.

### 3.4.2 Printing service

This service prints documents for authorized users. Nodes that wish to print a document call the print method with the document and its format as parameter.

This service require that there are printers in the network that supports Java executable code. Otherwise the printer can't offer the Jini service itself, nor can it verify tickets. If there does not exist such printers, a printer can be hidden behind a firewall. No one except the printing service, which can be called and used by the system users, should be allowed through that firewall.

## 3.5 Service management

This section describes the procedures taken before a service can be used. The algorithm is the same for all services. Secure communication, verification of code and other cryptographic issues are not described in this section. Also the role of the trusted authority is left to later subsections.

The client in figure 3.3 is a mobile node Jini client. It is assumed that bootstrapping is already done. The client consists of a user interface and a cryptographic code block. Everything a client download from a Jini lookup service can be either a proxy or just a plain object.

To request a service the following steps are taken.

1. Find the ticket service by asking for a ticket service proxy from the lookup service
2. The lookup service sends back a ticket service proxy (instance of the proxy and its class file)
3. The mobile node requests a ticket for service S
4. If the mobile node is authorized to use S, it receives a ticket
5. Find service S using the lookup service
6. The lookup server sends back an instance of the object and it's class file
7. The mobile node starts using the service object, attaching the ticket in the first call
8. The service S gives a response

## 3.6 Authentication, verification and tickets

### 3.6.1 Secure communication channels

JSSE, one of the Java implementations of SSL/TLS (see A.7.5), is used for mutual authentication, encryption of communication and for signing certificates used in the authorization phase. It is also used to get integrity, protect from replay attacks, agreeing on session keys etc.

JSSE can be used together with both RMI (see A.5.2) and sockets. The specific service type and use decides whether to use RMI or sockets.

When a client wants to talk to a service a secure communication channel is created. Thereafter the ticket is passed. If the ticket is valid from the services point of view, the communication can continue, otherwise the service will break it. In the secure communication setup between a client and an ordinary service authentication of the service is required, from the client point of view. When a client setup a secure communication channel with the ticket service, authentication of both peers is done. This is to ensure the ticket service who is actually requesting a ticket. From the client point of view it is important to know who it talks to, since it passes its ticket to the service.

### 3.6.2 Tickets

Tickets in the architecture are used to authorize mobile nodes when they wish to use a service. Tickets are basically a certificate with fields about issuer, service name, period of validity, cryptographic algorithm choice, how many times that a service can be used, a signature et cetera. Exactly what

fields that must exist depends on the service. A print service for example might need a field telling how many pages this ticket is allowed to print, while an Internet service ticket needs a lease time, giving information on how long time a user can surf the Internet.

### 3.6.3 How services verify tickets

When a service receive a ticket it must verify that it's authentic. The following steps are taken:

1. the service get the ticket services public key if not already possessed
2. verify the ticket using the ticket services public key
3. if the ticket is valid continue, else stop
4. the service checks whether the service asked for is what it can supply
5. check period of validity
6. if the ticket has a lease time, check that time
7. if the ticket has a maximum number of use, compare the requested value with the value in the local database, update the value in the local database, store this information until the period of validity is out
8. check other fields depending on ticket and service
9. if everything okay, let the client use the service

### 3.6.4 Ticket passing

Since a ticket does not contain a field about the owner of the ticket, anyone can use it. Therefore it is very important to make sure that the risk of eavesdropping is eliminated when a ticket is passed from the ticket service or from the client to a service. JSSE together with any communication paradigm (RMI or sockets) meet the requirements for this need.

### 3.6.5 Java class verification and policies

A Jini system builds upon the concept of code migration. This means that a client must run untrusted code in its own Java virtual machine. To prevent a client from running malicious code he or she can set policies to make sure that only trusted code is executed. The client can for example decide only to run code whose class files have been signed by the ticket service, which means that the client *knows* that the code executed really is from the ticket service. The client can also set much more restrictive policies. More information about policies can be found in A.5.1.

The class files in Java is signed with the use of the application *jarsigner* (see A.5.3). This tool creates a hash value corresponding to the *.jar* file, thereafter this hash value is signed using a private key. This signature is now an evidence of the codes origin.

### 3.6.6 Access right structure at the services

A service that require a client to show a ticket need to have a data structure where it stores information about all tickets in use. The service must store ticket information until the ticket's period of validity has exceeded. An example on why this is necessary is when a ticket has a maximum number of use field specified. Then the service accepting the ticket must know how many times that the particular ticket have been used before. Services should only store information about signed tickets.

### 3.6.7 Trusted authority

Whenever secure communication should take place the trusted authority needs to sign certificates for the communicating parts (if it has not done so earlier).

Assume that client C wants to use a service S. First C needs to obtain a ticket for S. This is done using the ticket service T. To be able to establish a secure communication channel between C and T, both C and T have to provide certificates from the trusted authority to each other. Since both trust the trusted authority and both can verify the validity of the certificates they can establish the secure channel. After the ticket is passed, C goes through the same procedure but with the service to use instead of the ticket service as peer. More information about trusted authorities can be found in A.7.4.

### 3.6.8 Client cryptographic code

This code is all code where the client has to use any of its private keys. The cryptographic code [22] must be located at the client, the private key can under no circumstances be transferred across the network. The cryptographic code in the architecture is basically an implementation of SSL/TLS where certificates can be signed, encryption can be made and authentication towards other nodes can be accomplished. The prototype implementation of this specification should use JSSE for this purpose.

## 3.7 Initial key distribution

Some keys used in the system need to be distributed in a secure manner before actually using the system. There exists several methods to accomplish

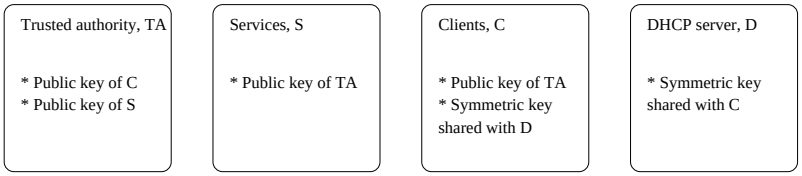


Fig. 3.4: Initial key distribution between nodes

that (see A.7.3). This is because computer security (as well as security in general) relies on trust relationships. The keys that need to be pre-distributed are the symmetric keys used in DHCP authentication, the public keys of each individual user and the public key of the trusted authority (TA) (see figure 3.4).

Each client needs to know the unique symmetric key that the DHCP server shares with them and the public key of the trusted authority. If the key of the trusted authority is known then the client can obtain other public keys by getting signed certificates from it.

## 3.8 Bootstrap

This subsection is about what all the mobile nodes entering the system needs to know before any communication take place in the system. In 3.7 it has already been mentioned what keys need to be where. Every client need a physical radio device configurated to work with the current parameters. Clients also need a version of Java, Jini and JSSE. The client need to be able to communicate using RMI and/or sockets.

Every client also needs a bootstrapping code sequence. This code learns where the Jini service is located from the DHCP Jini option. Thereafter it downloads the user interface and a service lookup list. After this is done the client can start use the Jini system.

The system is not totally self configuring, as a user might want. But since nodes can not trust the infrastructure there must be some initial trust relationships established and various code downloaded before start.

## 3.9 Security threats

There exists many known security attacks and threats. In this section the issue of spoofing is discussed. Other security attacks like denial of service [1] is not handled by the prototype specification.

**Spoofing** IP addresses of authorized nodes can be spoofed and thereby be used by a malicious user. Since the communication within the system is secured using JSSE a part of this threat is prevented. A malicious node can't

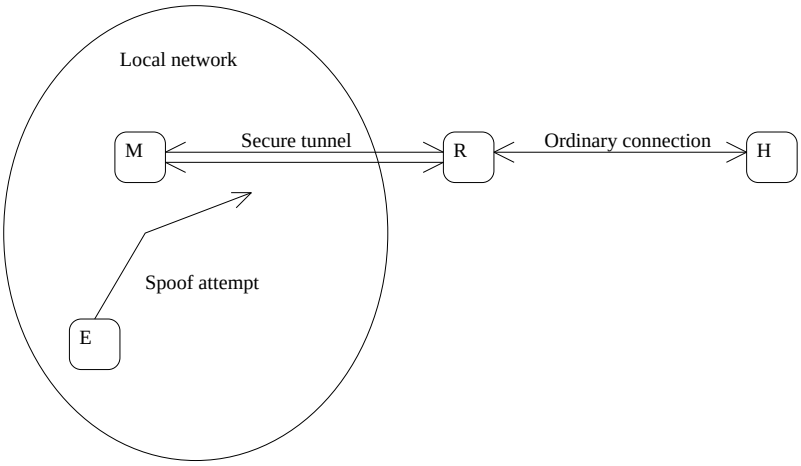


Fig. 3.5: E tries to spoof M's connection

pretend to be someone he or she isn't with the respect to identity based on certificates and tickets.

However when a client has bought access to the Internet the communication is not encrypted if the client and the Internet server is not using end-to-end encryption themselves. This means that a malicious user can spoof the clients IP address and use the Internet.

One way to prevent spoofing when a node is using the Internet is to introduce secure tunneling. To illustrate how this could work, four nodes M, E, R and H are needed (see figure 3.5). M is the mobile node inside the local network. E is an evil node trying to spoof M's address. R is a router connecting the local network and the Internet together and H is a random host on the Internet. Assume M wants to communicate with H. M must first get a ticket for Internet communication. This ticket is shown to R. If the ticket is valid, M and R establish a session key using their public key pair. Now, when M wants to contact H all communications goes through R. This means that the communication between M and R, on its way to H, can be encrypted using the session key. R only allow valid encrypted packets from M. This means that the evil node E has no ability to spoof M's address in the purpose to get Internet access, since it has no knowledge about the session key used.

This solution has several drawbacks. One problem is that the client must have been pre-installed with software that can handle tunneling. Another problem is how to solve the tunneling itself. Should the TCP/IP connection between M and H be broken up in two parts, one connection between M and R and one between R and H? And how should that be done. These problems are left to later work.

# Chapter 4

# Implementation

## 4.1 Introduction

The implementation of the proposed architecture and specification is a prototype, visualizing the most important parts of the system. The prototype should not be used as a foundation for any further development.

To really understand the prototype implementation it is important to have access to the source code or at least the source code documentation[1].

This section is a description of the prototype implementation and is organized as follows. First the prototype architecture is presented, which gives an overview of all components in the system. Thereafter a system description is given which introduce all Java classes, how security has been implemented and so on. The section ends with a demonstration setup description and a survey of differences between the specification and the prototype.

## 4.2 Prototype architecture

In this section the overall prototype architecture is presented. The main components are the access point, the infrastructure node, the firewall and the client, which can be seen in figure 4.1. A more detailed description which show what kind of servers and what code runs on which machine is shown in figure 4.2.

### 4.2.1 Infrastructure node

In the prototype implementation the HTTP server, Jini server, RMID[2] and all services except the *InternetService* runs on the same machine (infrastructure node). Making all these servers and services centralized is not a

[1] the source code and the documentation can be found at http://www.docs.uu.se/~anjo6413/

[2] RMID must run at the infrastructure node because Jini uses RMI internally

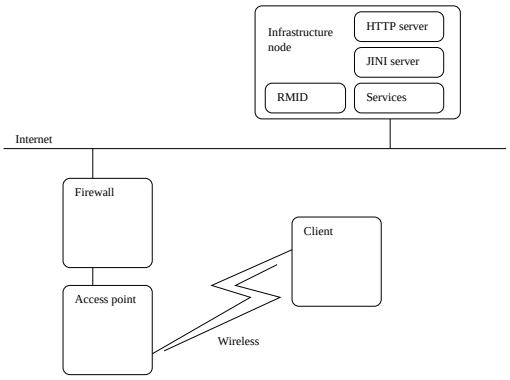


Fig. 4.1: Prototype architecture

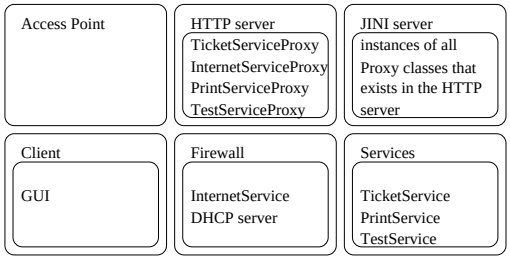


Fig. 4.2: Each component in the prototype architecture in more detail

requirement. The infrastructure node is connected to the Internet.

The HTTP server contains all proxy classes (TicketServiceProxy, InternetServiceProxy, TestServiceProxy and PrintServiceProxy), signed by the TicketService. The Jini service contains instances of all proxy classes, which have been initialized by its corresponding service object before it was uploaded to the Jini service. When a client wants to use a service it must download the service class file from the HTTP server and the corresponding instance of it from the Jini service. This is done automatically by the Jini system.

### 4.2.2 Firewall node and Access point

All user nodes in the system connects to the access point which is connected to the firewall node. The firewall node distributes local IP addresses via a DHCP server to all requesting clients. On the firewall the *InternetService* is running. The firewall is connected to the Internet.

### 4.2.3 Client

The client runs the graphical user interface (GUI), and downloads proxy class files and instances of them when it needs a service. Before actually

using a service the signature of the class file is checked. When download and signature verification is done the user connects to the service object via the proxy object. Authentication and authorization is performed between the client and the service.

## 4.3 System description

### 4.3.1 Policy files

To be able to execute any part of the system there must exist a *policy* file. This file decide what kind of code is allowed to execute in a Java virtual machine and what that code is allowed to do. For example if a user wants to execute an instance of the *TicketServiceProxy* it must grant execution of code signed by the *TicketService*. The user must also allow its installed components to execute, for example its Jini and JSSE implementation.

All services uses a policy file called *policy.all*. This file gives all code running at the service freedom to do whatever it want. This is not a security hole, since each service trust its own code.

To see examples on policy files the reader is referred to the source code. All policy files are in the directory *policy/*. More information on policy files can also be found in A.5.1.

### 4.3.2 Certificate stores

Each client and service must have a certificate store where it hold its private and public key pair and certificates of trusted entities. In the prototype implementation the *test* client node trust certificates issued by the *TicketService*, *PrintService*, *InternetService* and the *TestService*. The *TicketService* require mutual authentication and therefore it must trust the clients certificate. The other services do not require mutual authentication, since they accept and trust anyone with an accurate and signed ticket.

A certificate store is managed using the *keytool* (see A.5.4) application.

### 4.3.3 Socket communication

The interesting communication that take place in the system is the communication between the proxy objects and their corresponding services. The proxy objects act as interfaces towards the service objects. All communication uses the TCP/IP stack. On top of this, all communication within the system is secured using JSSE. Observe that communication outside the system, e.g. from a client to the Internet, is not encrypted. If a user needs encryption outside the system he or she must provide it him or herself.

The communication protocol implemented by the proxy and the service is on top of JSSE. This communication is very simple. For example, the

*PrintServiceProxy* sends a ticket, a file type and a printable file. Thereafter it waits for a string, used as acknowledgment, from the *PrintService*. More information on exactly how the other protocols are implemented can be found in the source code.

### 4.3.4 Class description

A short summary of the class usage is presented below. To get more knowledge about the class definitions the reader is referred to the documentation shipped with the system source code.

Several times the word *interface* show up. This should not be interpreted as the key word interface in Java. Instead it has a more general meaning.

**SystemServices.Ticket** The SystemServices.Ticket class represents access rights in the system. The class contains various fields that is needed to define what a user can do, to whom, for how long and for how many times.

A ticket is useless to a user if it has not been signed. In the prototype implementation the java.security.SignedObject class is used to sign tickets.

**SystemServices.TicketStore** The SystemServices.TicketStore class is used to create and modify a ticket store used by the active instance of SystemServices.TicketService. The creation and modification is done "offline". A ticket store is a file that contains unsigned SystemServices.Ticket objects.

**SystemServices.BaseService** The SystemServices.BaseService is a class that is the super class to all service classes (SystemServices.TicketService, SystemService.InternetService and so on). The class contains various methods that is shared between most of the service classes. Examples on methods are verification of tickets and registration of proxy objects.

**SystemServices.TicketService** The SystemServices.TicketService class inherits from SystemServices.BaseService. The service is indirectly used by all users in the system that needs any access right. The instance of this class check the identity of a requesting user and looks for a ticket matching the users access right demands in it's SystemServices.TicketStore object. If there exists a SystemService.Ticket object that match the user and its demands, the instance of SystemServices.TicketService respond with that ticket wrapped into a java.security.SignedObject.

**SystemServices.InternetService** The SystemServices.InternetService class inherits from SystemServices.BaseService. It is the service a user must call to get Internet access. The instance of this class validates the ticket sent

to it, by the user. If the ticket is signed by the instance of SystemServices.TicketService this service trusts and contains valid fields it open the firewall for the requesting user node.

The instance of the class SystemServices.InternetService must run on the node where the firewall is placed. The implementation assumes that the command *ipchains* (see A.7.1) is used to modify the firewall.

**SystemServices.PrintService** SystemServices.PrintService inherits from SystemServices.BaseService. It is used by nodes that wish to print files. It functions in the same way as SystemServices.InternetService when verifying tickets. The user must also supply the file type and the file it wishes to print.

The instance of the class SystemServices.PrintService must run on a node where printing is done using the commands "lp" or "a2ps", i.e. a Unix system of some kind.

**SystemServices.TestService** SystemService.TestService inherits from SystemServices.BaseService. This class is an implementation test class. It gives different access rights depending on what kind of ticket it receives. This class has no useful meaning.

**SystemServices.BaseProxy** The SystemServices.BaseProxy class is the super class to all proxy classes (SystemServices.TicketServiceProxy, SystemServices.InternetServiceProxy and so on). It contains methods common to all proxy classes. For example object init-methods and secure socket creation.

Every proxy class implements the communication protocol between a user and the service the user wants to call. So, the proxy object is an interface to the user when using a service. Every proxy class and instance of the class is downloaded by the user program from a HTTP server and Jini service.

**SystemServices.TicketServiceProxy** The SystemServices.TicketServiceProxy class inherits from SystemServices.BaseProxy. This class is an interface when using the service SystemServices.TicketService.

**SystemServices.InternetServiceProxy** The SystemServices.InternetServiceProxy class inherits from SystemServices.BaseProxy. This class is an interface when using the service SystemServices.InternetService.

**SystemServices.PrintServiceProxy** The SystemServices.PrintServiceProxy class inherits from SystemServices.BaseProxy. This class is an interface when using the service SystemServices.PrintService.

**SystemServices.TestServiceProxy** The SystemServices.TestServiceProxy class inherits from SystemServices.BaseProxy. This class is an interface when using the service SystemServices.TestService.

**SystemServices.UserInterface** The SystemServices.UserInterface class is used for downloading of services from the Jini service and to access the different proxy objects when the user wants to call a specific service. The instance of this class assume that a graphical user interface is available.

**SystemServices.Console** The SystemServices.Console is a help class that is needed to start SystemServices.GUI from a command line, for example a UNIX bourne-again shell or Microsoft Windows DOS. This class is written by Bruce Eckel [4].

**SystemServices.GUI** The SystemServices.GUI class uses the SystemServices.Console class to be able to execute from the command line. It implements the graphical interface used by the user. It creates a SystemService.UserInterface object that is doing all the real work.

**Debug.DebugPrint** The Debug.DebugPrint class is used by various classes (most service classes) when printing to standard output. Debug messages are sometimes important and sometimes not. The debug messages can either be printed on the screen or to a file.

## 4.4 How to setup and run demo

All steps on how to start the system are necessary. It may seem a bit complicated, but observe how easy it is for a *client* to *use* the system. At least after Java, Jini and JSSE has been installed.

All relative paths used in the setup description assumes that the current working directory is the directory where the system is unpacked.

### 4.4.1 Pre-configuration

All nodes that is going to execute any part of the Java programmed system must be pre-installed with Java, Jini and JSSE. Within this work Java standard edition version 1.2.2 (and 1.3.1), Jini version 1.1 and JSSE version 1.0.2 have been used.

Even if Java, Jini and JSSE themselves do not depend on operating systems, other part of this work do. On the client node Microsoft Windows98 has been used. The node acting as firewall used RedHat Linux with kernel version 2.2.12. The infrastructure node used SunOS 5.7.

The firewall node must be installed and compiled to support IP forwarding and IP masquerading, and of course other more basic network operations to make those options work. The firewall must also be configured with two networking devices. One connected to the Internet while the other is connected to the access point. The device connected to the Internet must be configured with an IP address, while the device connected to the access point can be left unconfigured. Last but not least the firewall must have a DHCP server installed. This work assumes "Internet Software Consortium DHCP Distribution, version 3"[3] to be used.

The infrastructure provider node must be able to print documents using the UNIX commands *lp* and *a2ps* to be able to take advantage of all benefits provided by this work.

#### Access point setup

Configure the access point to suite your needs of a local network. Enable or disable encryption, set the IP address so that the firewall node can find it and so on. In this work a WavePOINT-II Access Point from Lucent Technologies has been used.

### 4.4.2 Infrastructure node

1. Start the Jini lookup server, HTTP server and RMID using the script *txt/start_lookup*. Change the paths in the startup script to match your system configuration. Check for example where you have installed *tools.jar*, *reggie.jar* and where your nearest *policy.all* file is. Execute *txt/start_lookup <port no. of HTTP server> <localhostname>*

2. Open the file *SystemServices/BaseService.java* and set the variable *JINILOOKUPSVR* to match your Jini lookup server hostname.

3. Execute *txt/tarSystemFiles* and *txt/tarClientFiles*

4. Compile the system. Execute *txt/makeAll <path to codebase>*. The path to the codebase should be the document root path for the HTTP server.

5. Execute *DemoScripts/demoTicketService <httpserver:port>* to start the Ticket Service. The certificate file name is *txt/TicketService.sec* and the passwords are *changeit*.

6. Execute *DemoScripts/demoPrintService <printer> <httpserver:port>* to start the Printer Service. The certificate file name is *txt/PrintService.sec* and the passwords are *changeit*.

[3] http://www.isc.org

7. Execute *DemoScripts/demoTestService <httpserver:port>* to start the Test Service. The certificate file name is *txt/TestService.sec* and the passwords are *changeit*.

### 4.4.3 Firewall node

1. Download *SystemFiles.tar* from the infrastructure node.
2. Unpack in a suitable directory.
3. Execute *cp txt/dhcpd.conf /etc*
4. Execute *FirewallStartup/startnetwork*. Alter script to match your system, for example the local IP address which must have the same network number as the access point.
5. Execute *FirewallStartup/startfirewall <global IP address>*. Alter the local IP addresses in the script. The firewall must always forward traffic to at least one nameserver, the infrastructure node and the firewalls own global IP address.
6. Execute *javac SystemServices/*.java*.
7. Execute *DemoScripts/demoInternetService <global IP address> <httpserver:port>*. The certificate file name is *txt/InternetService.sec* and the passwords are *changeit*.

### 4.4.4 Client node

1. Before connecting to the access point providing the system, download *ClientFiles.tar*.
2. Connect to the access point. Make sure you get a local IP address.
3. Unpack *ClientFiles.tar* in a suitable directory.
4. Compile the code using the command *javac SystemServices/*.java*
5. Execute *DemoScripts/demoUserInterface.bat*. You have to alter the files in the *policy* directory to suite your system. If you want to make it easy just grant *AllPermission* to all code installed on your system, and grant *AllPermission* to all code you download and is signed by *TicketService*.

If everything has been installed, configured and executed correctly the users graphical interface will show up, on the client node. The graphical user interface is identical to figure 4.3. To actually use the program it has to be configured, which should be fairly easy. The *Jini server* must be the

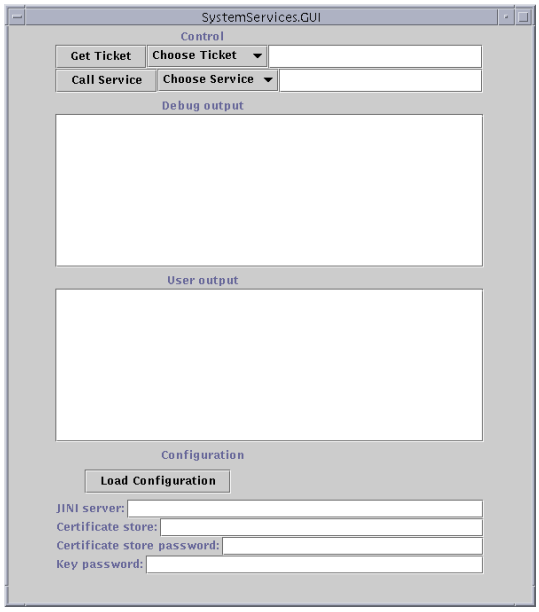


Fig. 4.3: Graphical User Interface

hostname (or IP address) of the infrastructure node. The *Certificate store* can be *txt/client*, *txt/client2* or *txt/unholyclient* depending on what a user should be able to perform. *txt/client* is the demo user, who can access all services. The passwords are both *changeit*.

Other fields are quite self explained when running the GUI.

## 4.5 Specification and prototype differences

### 4.5.1 What has not been implemented

**Trusted Authority** A trusted authority has not been implemented. The reason for this is that in a prototype you can decide exactly whom every node should trust since there are so few nodes. Another reason for not implementing a trusted authority is that there exist many trusted authority implementations on the Internet, although you have to pay to use them.

Yet another reason for not implementing a trusted authority is due to limitations in the certificate generating Java tool *keytool* (see A.5.4).

**Authentication for DHCP and DHCP Jini option** Neither of the DHCP authentication nor the DHCP Jini option has been implemented. DHCP authentication implementation requires very much work, which has not been in the scope of this thesis. Another reason is that changing the DHCP server and adding a Jini option require changes on the DHCP client.

Since the client in this prototype run Microsoft Windows98 that could be very time consuming.

**Group meeting service** A group meeting service has not been implemented since it is not a key service. It is just a service that is nice to have in a system, like the implemented printing service.

**Service lookup** A service lookup service has not been implemented. Instead the client GUI gives information on which services exist in the system. This is not what the specification says, but it is good enough for a prototype.

**Payment service** Micro payment theory and how to implement it is out of scope of this thesis. Still it is a very important service in a real system. Therefore it appears in the prototype specification but not in the prototype implementation.

**Bootstrap** A bootstrap code sequence has not been implemented since I have not found a way for Java to serialize graphical components. Therefore a graphical user interface was implemented instead. Of course the class *UserInterface* could have been downloaded by the GUI but that did not seem so important since the bootstrap code idea already was outplayed.

### 4.5.2 Cryptographic code

The idea behind the cryptographic code was to make sure that the user only gives its private keys to code downloaded and installed "off line" by the client. This has almost come true, since the client must have JSSE pre-installed. However all proxy classes require the passwords to the clients certificate store. This could be fixed, but since the proxy classes are signed by the *TicketService*, the user should be able to trust that the code is not doing anything bad with its private key.

Again, the implemented cryptographic code behavior is okay in a prototype but not in a real system.

### 4.5.3 Services and GUI limits

The services are very simple. They to not meet all the requirements in the specification. For example a user can send the very same ticket as many times as it wishes, since the service do not remember what has been sent by whom. In a real system this must of course be fixed.

The GUI lacks support to add new services on the fly. This is of course not good, but the GUI has enough functionality to show how the implemented services work.

## 4.6 Implementation problems

### 4.6.1 RMI vs. Sockets

The major problem encountered during the implementation phase of this master thesis work was the choice between using RMI or sockets when implementing the secure communication between proxy objects and service objects.

At first an RMI implementation was partly finished. But since RMI hide much of the low level communication details there was no way of finding out what certificate the user had used to identify itself. This caused problems since the *TicketService* need to identify the user, to be able to decide what ticket it should process. Also, RMI was quite slow. Therefore the RMI implementation evolved into a bare socket implementation.

In later JSSE implementations there has been "promises" that there will be more support when using RMI.

### 4.6.2 Tickets

The signature of the first version of the SystemService.Ticket class was a calculation depending only of the data fields. This was of course not good since a malicious node then could change the methods of a SystemService.Ticket object. After quite a bit of work the class java.security.SignedObject was found. This class signs a whole instance of any class, both the data fields and it's methods. Therefor java.security.SignedObject is now used to sign tickets.

## 4.7 A weakness in the implementation

Apart from what has not been implemented the prototype implementation suffers from one major weakness. The client only searches for a specific service in the lookup service once. This means that if the service provider changes the implementation and uploads a new version to the lookup service, the client will not notice. The client uses a static service lookup, instead of using the dynamic lookup which a user or the service provider might expect.

# Chapter 5

# Conclusions, related work and future work

## 5.1 Conclusions

In this work a system for differentiated security has been specified and partly implemented, based on an architecture. The architecture is a generic construct for differentiated security in wireless networks while the specification and implementation, which builds upon the architecture, use a specific set of tools. The system provides easy access and low configuration services, using the service discovery system Jini, to nodes that wish to use the system. All communication within the system are secured and all nodes are authenticated using JSSE. If a client wants to use a service provided by the system it needs a ticket, to authorize itself. All services in the system trust tickets signed by a ticket service.

The specification and implementation assumes that the client is pre-configured with several components (Jini, JSSE and the system GUI). The client must also have had a secure contact with a trusted authority and a secure DHCP server. When this contact was made the nodes exchanged cryptographic keys (both asymmetric and symmetric) with each other.

In the specification, priority is based on real life situations, payment or location. While the prototype implementation only base priority on real life situations, i.e. the access rights are pre-configured.

The services that have been implemented in this work are the ticket service, the print service and the Internet service. These services can be used by all clients that possess the correct access rights. The ticket service grant tickets to authorized users, the print service prints documents for authorized users while the Internet service opens the firewall to authorized users.

## 5.2 Related work

All related work is presented in appendix A. Ideas about the ticket service system and the security add-on to Jini is directly taken or derived from [22].

There may exist other solutions that solve the problems stated in 1.1, but we have not found any similar system that builds upon the ticket service management and Jini.

The Kerberos system [1] does have a ticket service and could be used to accomplish differentiated security. But Kerberos does not solve the demands on easy configuration when entering a new network. Another difference is that Kerberos use symmetric key cryptography, which in this thesis work would be insufficient.

## 5.3 Future work

This thesis report provides an architecture, a prototype specification and a prototype implementation that combined are a solution to the introductionary scenarios. Future work is then to create a better specification and implementation. It can also be desirable to make the ticket handling and service usage more invisible to the user. For example, a user that wants to print a document should not need to start a special software to print it. Instead the ticket handling and service usage should be automatically activated and executed by the word processor's own printing command. The only thing a user should need to do is to give the passwords to his or her file where private/public keys and trusted certificates are stored.

Another interesting issue is how to do handover between different security domains. Suppose that there are at least two security domains (with one DHCP and one Jini server each) at an arcade and that a user wants to belong to both of them. One question that needs to be solved is how address handling and handover should be accomplished. A node that belongs to both domains may need a local address in both domains to be able to use their services. Another question is if the two security domains needs to cooperate to allow a node to belong to several domains or if this can be left to the user software.

# Appendix A

# Description of important nontrivial techniques

## A.1 BOOTP

BOOTP [9], short for Bootstrap protocol, is an early service discovery protocol built upon UDP/IP. It is used to make diskless clients able to discover their IP address, the address of a server and the location of a bootstrap file. This bootstrap file is downloaded from a server and then executed locally in the clients memory. BOOTP consists of two phases. In the first phase all information is gathered while in the second phase the actual file transfer of the bootstrap file is made, using TFTP [10] or similar protocols.

When a client needs to be configured (i.e. boot a machine) it broadcasts a *bootrequest* packet. If a BOOTP server is available it will respond with a *bootreply* packet. If the client did not get a *bootreply* packet within a certain amount of time it will retransmit until a reply is received. Both *bootrequest* and *bootreply* are of fixed size and share the same data fields.

## A.2 DHCP

The dynamic host configuration protocol [3][11] (DHCP) is used to simplify installation and maintenance of computers in a TCP/IP network. DHCP is based on the Bootstrap protocol (A.1), adding automatic allocation of reusable network addresses and the concept of options. From the client point of view DHCP is a discovery protocol whish gives information about what IP address to use, the default router IP, subnet masks, for how long it is allowed to use a specific configuration and so on. From the server point of view DHCP is a resource allocation protocol that keeps information on what has been allocated by whom and for how long.

DHCP is built on a client/server model. The DHCP client sends a request to a DHCP server. The server, after receiving the request, responds to

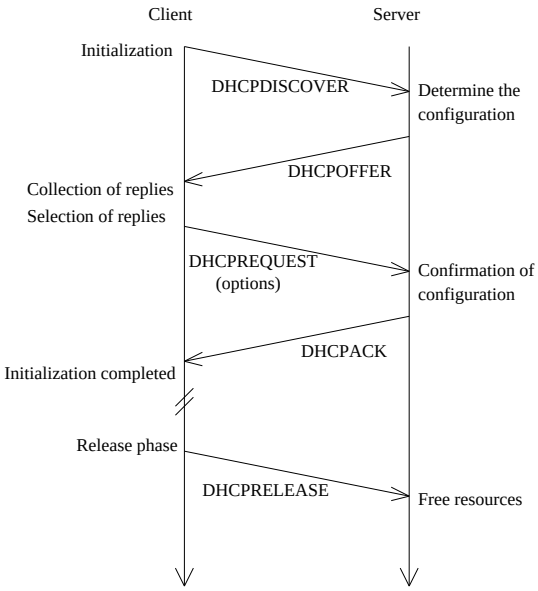


Fig. A.1: Client initialization and release via DHCP

the client. Figure A.1 shows a normal DHCP session between a client and a server. The client first broadcast a *dhcpdiscover* message. This message may include suggestions for different fields, such as the network address and lease duration. The local DHCP server process the request and thereafter sends back a *dhcpoffer* message, which contains a list of different possible network configuration parameters. The client accept one of the configurations and replies its choice with a *dhcprequest* message to the server. The server confirms the configuration choice by sending a *dhcpack* to the client.

Since there is a limited amount of resources in a network, a client lease its configuration (i.e. network address). This lease must be renewed every now and then. If a client does not renew its lease, the DHCP server will free the resources associated with that client. When a client leaves the network it should send a *dhcprelease* to tell the DHCP server to release the resources held by that client.

There are a number of DHCP options [12] that may be relevant in some networks. Different DHCP servers and clients may have a different set of supported options. Therefore the policy of dropping all options that a client or a server can not recognize is used. An example of an option useful in wireless environments is the "Mobile IP Home Agent option" which specifies a list of IP addresses indicating mobile IP home agents available to the client. Other useful (useless?) options are the "Default Internet Relay Chat Server option" and the "Default Web Server option". A system administrator could define new DHCP options used in his or her network. If these new options are to be used the DHCP clients also need to be updated. New DHCP

options should be defined in an RFC.

## A.3 Service discovery protocol overview

Today, devices of all kinds are often allowed or required to be connected to a network. If they are not they may not even function the way it is supposed to. Regardless if the device is in your home or at your office, network configuration of these devices has to be an automatic task, since the number of devices can increase uncontrolled. Because of this fact multiple device and service discovering protocol have emerged, both of research interest and commercial products. Examples of service discovery protocols are Salutation, SLP, Universal Plug and Play, Jini and Bluetooth.

All of these proposed protocols share a common set of features. One of the most important feature is the registry and registry update mechanism. All services supported by a network are registered in the registry, using the registry update mechanism. Both clients and servers should be allowed to register services. Another equally important mechanism is the client search, which let a client search the registry and hopefully find the specific service it needs.

The rest of this subsection gives a very light overview of different service discovery protocols. A good starting point for more information is [27].

**Salutation** is an architecture for looking up, discovering and accessing services. Most implementations of Salutation have been focused on enabling access to office equipment like fax machines, printers et cetera.

**SLP** (Service Location Protocol) is a standard developed by IETF[1]. SLP tries to solve the problems of self-configuration and service discovery by using existing Internet applications and services, for example DHCP.

**Jini** is developed by Sun Microsystems[2] and solves the service discovery needs for Java enabled devices and systems. Other devices can also be discovered and used since a Java device or Java code can act as a proxy to a non-Java device or non-Java code. Jini will be described in more detail in another subsection.

**Universal Plug and Play** is a creation of Microsoft[3] and tries to define a set of discovery protocols to allow appliances such as telephones, televisions, printers and game consoles to exchange data among themselves.

[1] http://www.ietf.org
[2] http://www.sun.com
[3] http://www.microsoft.com

**Bluetooth** is a consortium developing a short-range wireless communication protocol. As part of this development a service discovery layer is also implemented. Bluetooth was founded by Ericsson[4], Intel[5], IBM[6], Nokia[7] and Toshiba[8].

## A.4 Jini

Jini [4][6][7][8] is a Java component which aids distributed system development, where the distributed system is a *federation of services*, which basically means a set of services where no central controlling authority is present. A service is anything that provides functionality towards a user in a network. Printers, fax machines, communication channels and software are examples of entities that can provide a service towards a user. Peers using Jini can add services, remove services and search the federation for useful services. Jini amis for and should provide easy access, ease of administration and resource sharing to every member of the federation.

The rest of this subsection will be organized as follows. First the key concepts of a Jini system will be presented. Thereafter the service architecture, including the different protocols used within the Jini architecture, is presented.

### A.4.1 Key concepts

**Services** The services are the central concept within the Jini architecture. A service is something that a person, a program or another service can use. A service provides functionality towards its user. The functionality could be almost everything imaginable (computations, storage, communication channels, security algorithms, various software programs, hardware devices et cetera, et cetera).

**Lookup service** The lookup service is where services are found by the users. Users that search the lookup service for a specific service do so by specifying a well known Java interface. The lookup service maps interfaces indicating the functionality of a service with a set of objects, implementing the service itself. A service can either be a piece of code doing something useful or a proxy which provides an interface towards the real service, which could be located anywhere. A service can also provide other lookup services, i.e. hierarchical lookup.

[4]http://www.ericsson.com
[5]http://www.intel.com
[6]http://www.ibm.com
[7]http://www.nokia.com
[8]http://www.toshiba.com

**Security** Jini security builds upon *principals* and *access control lists*. When a user (or principal) tries to access a service, the service looks in its access control list to find out whether to accept or deny usage.

**Leasing** A lease is something that tells a user of a service for how long it may access the service. The lease is negotiated between the user and the service provider. If a user requests a too large lease the provider decides a suitable length of the lease itself. If a user wants to use a service for a longer time than the negotiated lease, the lease must be renewed. If the lease is not renewed or if the provider will deny further lease the user will fail when accessing the service.

Leases can be either exclusive or non-exclusive. Exclusive leases garanties that no other user can access the service during the first user's lease. The non-exclusive lease mode does not ensure this facility.

**Transactions** A series of operations can be wrapped into a Jini transaction. The Jini transaction feature supply a *two-phase commit*[5] protocol.

**Events** Jini supports distributed events. This means that if an object on a machine A triggers an event, an object at machine B will be notified about what happened. To be notified about events the event listener object must first register interest in the specific event, at the object throwing it.

### A.4.2 Service architecture and protocols

The Jini architecture consist of clients, service providers and at least one lookup service (figure A.2). There are three protocols in the Jini system that are of special importance. They are called *discovery*, *join* and *lookup*. These three protocols are described below. Good program examples can be found in [8].

**Discovery** The *discovery* protocol are used by all entities in the Jini federation that wish to find lookup services in the network. The discovery of a lookup service is done by multicasting a request on the local network. All lookup services should be accessible on a well known port number. The node running the discovery protocol will gain a list of available lookup services. The discovery can also be direct, which means that the node looking for a lookup service already know the network address. The direct method allows the lookup service to be out of range of multicast or broadcast requests.

**Join** The aim of the *join* protocol is to let service providers register service objects within the lookup service. The service object contains the Java interface for the service as well as the implementation of methods a user can

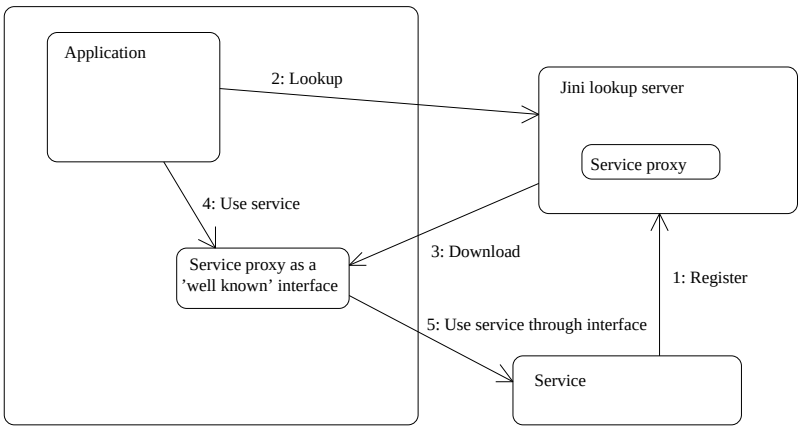


Fig. A.2: Using Jini

call. Together with the service object the service provider can specify a set of attributes. These attributes are used to ease the search for the service inside the lookup service. The join process is shown in figure A.2, step 1.

**Lookup** The *lookup* protocol is used by clients that wish to use a specific service. The *lookup* protocol can only be used after the *discovery* protocol has returned at least one lookup service. When a client needs a service it specifies the services Java interface as well as optional attributes. The lookup service only returns a service if the Java interface and the specified attributes supplied by the client exactly matches the interface and the attributes specified by the service provider in the *join* protocol. The lookup process is shown in figure A.2, step 2. In step 3 the actual service is downloaded to the client.

**Client uses the service** When the client has successfully executed the *lookup* protocol it has a downloaded object in its own environment. Now the client can execute whatever methods that is available in the object (step 4, figure A.2). Depending on how the service was implemented the service either executes on the clients local machine. In the other case, when the downloaded service is a proxy, the actual execution is done at the service provider (step 5, figure A.2).

## A.5 Java related

### A.5.1 Java security model

The Java security model [23] consists of several methods to make the user unable to execute erroneous or malicious code. The security model is called the basic sandbox. It is built up on four main components:

- the class loader
- the class file verifier
- safety features built into the Java virtual machine
- the security manager

**The class loader** The class loader maintains a set of *name-spaces*. Each name-space is a set of unique names, one name for each loaded class. If a class with the name *TestClass* was loaded into a particular name-space, a class with the same name can't be loaded into that name-space again. This helps to prevent malicious code from interfering with pre-installed components and classes trusted by the user. However, a class with the same name, *TestClass*, can be loaded into another name-space.

Classes inside a single name-space can interact with each other. While classes from different name-spaces does not even know the existence of other name-space classes. Of course a programmer can give knowledge about other name-spaces, but that is done in an explicit manner.

The user can set protection domains for each name-space. A protection domain gives information on what classes in a name-space are allowed to do. Actions that can be controlled are for example file access. This enhance the security in the system.

**The class file verifier** The class file verifier ensures that loaded class files have a accurate structure and that they are consistent with each other. Four different checks are performed. The first checks the basic structure of a Java class file, for example it checks that the class file start with the same sequence of bytes: 0xCAFEBABE. The second check looks at the semantics, for example it is looking at the return type and number of parameters in method calls. The third check is a bytecode verification, which performs a data-flow analysis. The fourth and last check deals with verification of symbolic references contained in a class file that should be resolved in the process of dynamic linking. This means that it is checked that every class that should be dynamically linked must be present in some way.

**Safety features built into the Java virtual machine** The safety features deals with checking and verification of for example the following subjects:

- type-safe reference casting
- structured memory access
- automatic garbage collection

- array bounds checking
- checking references for null

By making all these checks a Java program will be robust and safe to run. For example no pointer errors can occur.

**The security manager** While the three former parts of the Java sandbox is about protecting the internal structure of a Java execution environment the security manager protects from malicious code. Either this code is downloaded at run-time or installed and later used.

A user can explicitly tell the Java virtual machine to allow different actions to be taken inside different protection domains. For example a user can specify that downloaded code must have been signed by a specific user, otherwise the Java virtual machine will not run that code. The user can further specify that the downloaded code only is allowed to read a specific file.

The protection domain configuration is made in a policy file. An example of a policy file is:

```
keystore ‘‘myKeyStore’’;

grant signedBy ‘‘friend’’ {
permission java.io.FilePermission ‘‘question.txt’’, ‘‘read’’;
permission java.io.FilePermission ‘‘answer.txt’’, ‘‘read’’;
```

This policy file first says that certificates trusted by a user and keys owned by a user are stored in the file *myKeyStore*. Then it says that all code signed by *friend* is allowed to read the files *question.txt* and *answer.txt*. More information on how to define policy files can be found in [24].

### A.5.2 RMI

RMI [4] is a Java implementation of remote procedure calls. The basic use of RMI is to let users execute a piece of code on a different machine. The user that wants to execute a specific piece of code download a proxy interface corresponding to that code, which is called a stub in Java terminology. By executing various methods in the stub the actual code, located on another machine, is accessed.

### A.5.3 jarsigner

The Java tool *jarsigner* [31] is shipped along with the Java standard edition[9]. It's purpose it to sign a class files bundled together in a .jar[10] file. If a .jar file is signed the user of it can verify it's origin, for example using *jarsigner*.

If a user wants to grant execution permission to a specific signer dynamically, he or she specifies that requirement in his or her policy file. Examples on policy files can be found in the *Implementation* section.

### A.5.4 keytool

The Java tool *keytool* [32] is shipped along with the Java standard edition [11]. The purpose of *keytool* is to create and maintain certificate stores where a node has its public key pair and a chain of trusted certificates. The certificates should be of X.509 type [1].

The main drawback with *keytool* is that there is no possibility to sign others certificates. This means that *keytool* can not be used when implementing a trusted authority database.

## A.6 Wireless communication

In many system nodes can not be connected together using wire, but still a network must exist. Such a scenario could be a network connecting soldiers in a war zone. Another example could be if a network is needed in a historical building. Drilling holes for the wires could seriously harm the building and therefore a wireless network is needed. There exist many devices and protocols for wireless communication, both for data and tele communication. Two popular device and protocol specifications for data communications are the IEEE 802.11 and Bluetooth [3].

All wireless communication devices use electro-magnetic waves in different frequency bands to communicate with each other. A digital bit can be encoded using different methods. The straight forward methods are amplitude shift keying, frequency shifting keying and phase shifting keying. For example a high amplitude on the electro-magnetic wave signals a 1 and a low amplitude signals a 0. In real devices a complex combination of these methods are used to encode bits.

When more than one device is using the same frequency band multiplexing must be adopted. In wireless communication this is done either by dividing the space where a single entity is allowed to send, by dividing the

[9] java.sun.com

[10] .jar files is a way of putting together several .java and .class files to make it easier handle them

[11] java.sun.com

frequency band over multiple users, by dividing the time so that one user has one time slot or by using digital codes.

Since there is more than the wanted electro-magnetic waves in the air, methods for minimizing the impact of interference must be taken. This is done by spreading the senders frequency spectrum, either by direct sequence spread spectrum (DSSS) or frequency hopping spread spectrum (FHSS). When using DSSS the bit stream to be sent is manipulated with different bit codes, that both the sender and the receiver know, to spread the signal over multiple frequencies at the same time. The FHSS sends the bit stream over one frequency at the time. After a certain amount of time the frequency changes. Thus it is called frequency hopping. The frequency can either change x times for every bit sent, or change after y bits sent.

### A.6.1 IEEE 802.11

IEEE 802.11 belongs to the same family as other 802.x standards. This implies that 802.11 shares the same user interface towards higher protocol layers as for example 802.3 (Ethernet) and 802.5 (Token ring). The 802.11 standard specifies the physical and medium access layers, according to the OSI model [2].

The main components in a 802.11 network is the station (the client node) and the access point. Stations connect using no wire to the access point. The access point in it's turn is connected to some distributed system, maybe the Internet.

The 802.11 physical layer (OSI model) is using either FHSS, DSSS or infrared. The DSSS is most common when using 802.11 devices. The mandatory basic method for medium access control is carrier sense multiple access with collision avoidance [3] (CSMA/CA). In this scheme any node can access the medium at any point in time, but only if the carrier is free. If a node detects a collision it backs off for a random amount of time, and thereafter tries to resend.

## A.7 Security

### A.7.1 IP-chains

*ipchains* [29][30] is a tool under the Linux operating system that is used to create, maintain and inspect the firewall rules. *ipchains* has its firewall rules divided into 4 categories: the IP input chains, the IP output chains, the IP forwarding chain and user defined chains. Each chain contains a set of rules that every packet that is going through the firewall has to meet.

The firewall under Linux bases its rules on the destination and source addresses as well as port numbers in the packets flowing through. This means that an administrator can give permission to packets trying to reach

a specific destination while other packets will be caught in the firewall. As was said above *ipchains* has 4 chains. Every packet trying to get through the firewall encounter the different chains in the following order: input chain, forwarding chains and the output chain. From these chains a packet can be forced to jump to user defined chains. Error checks are done before a packet enters the flow of chains.

When a packet first enters the input chain its source and destination fields are examined. If there is a valid match within the input chain the packet is forwarded to the kernel routing code. The basic routing tables under Linux can be configured using the *route* command [28]. If the packet is destinated to a local process the packet is directly forwarded to the output chain, while if the packet is destinated to some other machine it is forwarded to the forward chain. Again, in the forward chain, if there is a valid match within it the packet is forwarded to the output chain. The same process is done at the output chain. This means that every packet that is not destinated to the local machine has to traverse at least three different firewall chains.

*ipchains* also have other properties. It can set masquerading on and off to specific packets trying to reach a destination or coming from a specific source. If masquerading is used the traverse through the chains will be a little bit different.

### A.7.2 Secure group communication

Security in groups is an increasing requirement in applications today. In a not so far future news companies, for example, want to distribute their video reports on the Internet using multicast to paying consumers. To make sure only the subscribers can get the video stream all data has to be encrypted using a key, known to all subscribers. Updates of this key needs to be done when new subscribers enter or if someone leaves the group. Of course, real time demands and multicast problems must also be handled. Another scenario where security in a group is crucial is in a war zone. Suppose several soldiers share a common secret key for secure communication. If one or more soldiers get compromised the key must be updated. The compromised nodes must be detected before the key update is made. The key also needs to be updated if new soldiers join the group.

The two scenarios described above deals with the same problem: dynamic secure group communication. There are several methods and architectures on how to distribute and update keys in a secure group [16][17][18][19][20]. The protocols span over a wide range of methods: distributed, centralized, flat and hierarchical. The design issue all important protocols tries to meet is scalability, which is also called "1 affects n" scalability. That means that if one node enters or leaves the dynamic secure group, everyone should not be directly involved in the action. Another important design issue is robustness, or avoidance of single point of failures.

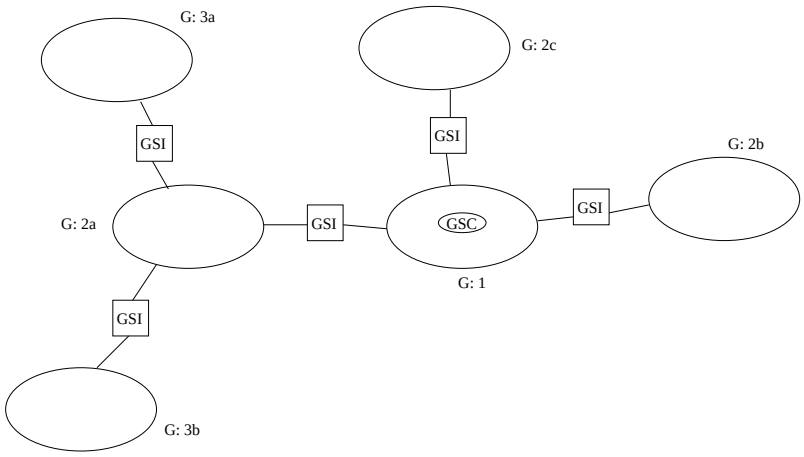


Fig. A.3: Example of Iolus tree

One of the easier protocols to understand is Iolus [17], which is a hierarchical half-distributed dynamic secure group communication protocol. Iolus divide a group into subgroups and thereafter arrange the subgroups in a hierarchical manner. Each subgroup has a common secret key for encryption and decryption of data. If a node enters or leaves the subgroup only this secret key needs to be updated. All other subgroups will be unaffected of that action. The central subgroup has a node called GSC[12] which manage the top-level subgroup. All other subgroup has one GSA[13] which handles key management within the subgroup. Every subgroup is connected to other subgroups via a GSI[14]. The GSI convert encryption in one subgroup to fit encryption in another subgroup. An example will clarify this, see figure A.3. Suppose a node A in *G: 3a* wants to send a multicast message to the group *G: 3b*. A encrypts the message using the secret key used in group *G: 3a* and sends it to the multicast address representing group *G: 3b*. The GSA in A's subgroup receives the message and learns that it is destinated to *G: 3b*. The GSA decrypts the message and thereafter encrypts it with a key it shares with the GSI connecting *G: 3a* and *G: 2a*. That GSI forwards the message to the GSA in *G: 2a* which learns where the packet should go. The operation of encryption and decryption is then iterated until the message has arrived at the destination.

In this kind of architecture the GSA and GSI are very sensitive. If they get compromised the whole system will be insecure, at least for the subgroup trusting the entity. There are methods to protect and prevent many kinds of security attacks, in Iolus and in all the other important dynamic secure group communication protocols. The interested reader is referred to the reference

[12] Group security controller
[13] Group security agent
[14] Group security intermediaries

list.

### A.7.3 Secure key exchange

Cryptographic keys needs to be transferred in a secure manner. The basic demand in all key transfer solutions is to avoid all attacks where the key can be possessed by a third party. When two parties wish to use secure communication over a wireless medium and does not share a common encryption scheme with keys it is impossible to setup a secure communication due to the fact that of security attacks [1]. Therefore several solutions have been proposed to achieve the goal of transfer keys in a secure manner. These are optical reading, physical transmission and secret password key evolvement.

**Optical readings** By adapting optical readers transmission of public keys can be made easy [26]. For example a users public key can be printed in the barcode format. Then the key can be printed on business cards, credit cards or on any other place. Instead of printing the whole key a shorter fingerprint can be used to identify it. One problem with this solution is to assume that the print of the key (or its fingerprint) is valid. But that is easier to accomplish in the physical world than in the digital one.

**Physical transmission** Keys can also be exchanged via physical transmission, then the key or its fingerprint never needs to be viewable in any way. One way is to connect two nodes [14] via some device and exchange the key.

**Key exchange with known password** If a group of users sits in a room and want to communicate in a secure way using their radio interfaces they can use a method for key distribution called password-based authenticated key exchange [15]. This means that the group can write a secret password on a white-board for example. Then, only the participants knowing the secret password in the room can calculate the session key to be used.

### A.7.4 Trusted authority

A trusted authority [1] (also named certificate authority) is a node, single or distributed, that everyone trusts. Then if two parties that have never met each other, and thereby do not trust each other, they can both ask the trusted authority to get an insurance of the other node's identity. Now the nodes can decide whether or not to continue the communication setup. This requires that both nodes have been registered within the trusted authority previously.

The information that a node receives from a trusted authority (TA) is most often a certificate signed by the TA. Within this certificate there is a

binding between a node name and its public key. This makes it possible to setup an encrypted communication with the other end.

### A.7.5 SSL/TLS and JSSE

Since more data is stored on computers and transferred across networks today, tools providing basic security features must be easily accessible, installed and maintained for everyone. SSL [1][25] is one such tool which provides a wide spectrum of security features and protocols, including DSA and RSA [1]. SSL was designed by Netscape[15] in 1994 to be used within web cryptography. Version 3 of SSL took input from the public and thereby became a standard. Later the TLS working group was formed within the IETF[16]. This group took control of the further development of SSL, under the new name TLS. In this thesis only the basics of SSL and TLS will be presented, and while only looking at the basics, SSL and TLS are quite the same.

SSL basically provides a security enhancement to TCP/IP. SSL is placed above the IP and TCP layer in the TCP/IP stack, but below the application layer. The current implementations require that the application layer is aware of the TCP/IP security enhancement. One reason SSL has been widely spread and used is due to the fact that it supports many different cryptographic processes. SSL uses public key cryptography in optional combination with public key certificates to provide authentication and secret key cryptography and digital signatures (and/or cryptographic hash functions and message authentication codes) to provide confidentiality and integrity. Although SSL without any options specified does not provide non-repudiation, even that security feature is feasible with SSL, as many others.

Before actually using SSL in communication, a secure communication handshake is performed between the client and the server. The handshake process is shown in figure A.4.

1. Client hello: The client initiate the handshake by sending, among other things, the highest version of SSL it supports. Choice of, for example, cryptographic algorithms, hash functions and key sizes are supplied.

2. Server hello: The server chooses the highest version of SSL and the best cryptographic algorithm and key size that both the server and the client support.

3. Certificate: If the client require authentication of the server, the server must respond with a certificate to prove its identity. This certificate must have been signed by a certificate authority trusted by both the client and the server.

[15] http://www.netscape.com
[16] http://www.ietf.org

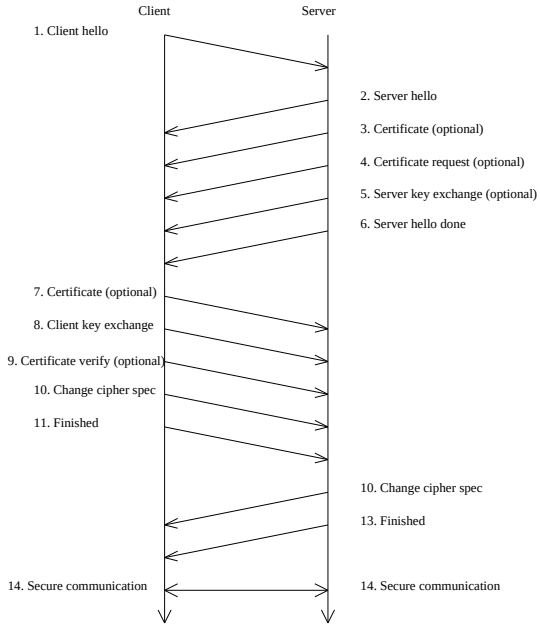


Fig. A.4: The SSL handshake

4. Certificate request: If the server needs to authenticate the client, a request of that fact is sent.

5. Server key exchange: If the public key information sent in step 3 is not enough for key exchange, the server sends the client a server key exchange message.

6. Server hello done: The server tells the client that the server has finished this part of the handshake.

7. Certificate: If the server sent a certificate request, the client must respond with a signed certificate to prove its identity.

8. Client key exchange: The client generates information used to create a cryptographic symmetric key. This information is encrypted (using for example the servers public key) and sent to the server.

9. Certificate verify: The client sends information to the server to make it able to verify that the client really is the owner of the certificate sent in step 7.

10. Change cipher spec: Message sent to the server letting it know that the client wants to change to encrypted communication, using the parameters agreed on.

11. Finished: Client done and ready for secure communication.

12. Change cipher spec: Message sent to the client letting it know that the server wants to change to encrypted communication, using the parameters agreed on.

13. Finished: Server done and ready for secure communication.

14. Secure communication: The client and server communicate using the agreed symmetric key together with the symmetric encryption algorithms and cryptographic hash functions negotiated in the first pair of messages sent.

When keys and other parameters has been agreed on communication can take place. Every message sent from a client or from a server goes through the following steps. First the data to be sent is fragmented. Each fragment is compressed. A message authentication code (MAC) is added to each fragment to ensure integrity. Encryption is thereafter done to ensure confidentiality. The last step is to add a SSL header. Thereafter the fragment can be sent to the other peer.

**JSSE** JSSE[17] [25] is a Java specification and reference implementation of SSL/TLS. Next, a client and a server written in Java using JSSE to setup a secure communication is shown. In the server a *SSLServerSocketFactory* is created. With this class decisions can be made on what certificates to trust, and what public keys to use and so on. In the example a default *SSLServerSocketFactory* is created. Using the *SSLServerSocketFactory* a *SSLServerSocket* can be created. This object listens on a specified port for incoming connections. When an incoming connection has been received a *SSLSocket* is created, which is later be used when transferring data across the network.

```
// Server WITH SSL
1.  import java.io.*;
2.  import javax.net.ssl.*;

. . .

3. public static void main(String [] args) {
4. int port = somePort;
5. SSLServerSocket s;

6. try {
7. SSLServerSocketFactory sslFact =
(SSLServerSocketFactory)
```

[17] Java Secure Socket Extension

```
SSLServerSocketFactory.getDefault();
8. s = (SSLServerSocket) sslFact.createServerSocket(port);

9. // Now the socket s can be used to send and receive information, securely
10. SSLSocket servSock = (SSLSocket) s.accept();
11. } catch(IOException e) {
12. }
```

The client first create a *SSLSocketFactory*. This class can be used to specify what certificates to trust and what public keys to use and so on. In the client example a default *SSLSocketFactory* is created. Using the *SSLSocketFactory* a *SSLSocket* is created. The *SSLSocket* is used when transferring data across the network.

```
// Client WITH SSL
1.  import java.io.*;
2.  import javax.net.ssl.*;

. . .

3. public static void main(String [] args) {
4. int port = somePort;
5. String host = ‘‘serverHost’’;

6. try {
7. SSLSocketFactory sslFact =
(SSLSocketFactory)
SSLSocketFactory.getDefault();
8. SSLSocket s = (SSLSocket) sslFact.createSocket(hort, port);

9. // Now the socket s can be used to send and receive information, securely
10. } catch(IOException e) {
11. }
```

### A.7.6 Authentication and authorization for Jini

Jini itself does not have much support to authorize users in a network. Also, clients wish to authenticate the services within a Jini network. Therefore other tools needs to be used.

[22] presents an architectural solution. Instead of just letting the client request a proxy object from the Jini server and then execute it on its local machine the following steps are taken (figure A.5):

1. Before sending the proxy to the lookup server, the service signs the proxy code and state.

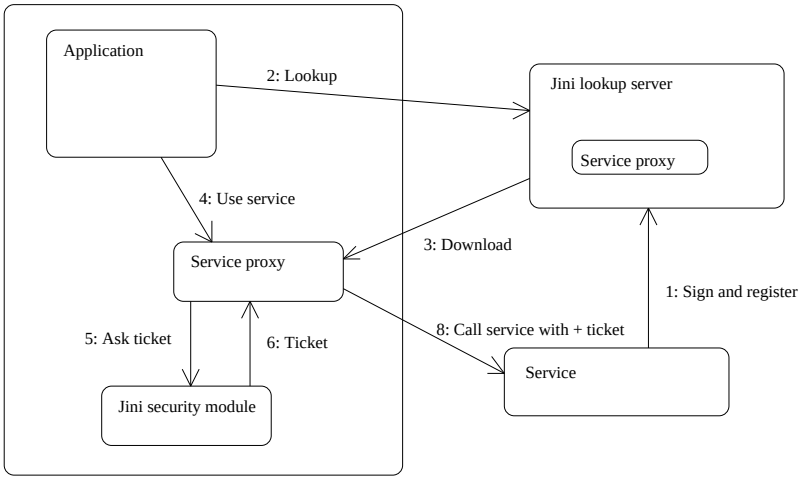


Fig. A.5: Using Jini with authorization

2. The application tries to find the service in the lookup server.
3. The proxy object is downloaded, as well as its class file. The signature of the code and state are checked.
4. The application calls a method on the proxy object.
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. The ticket is signed by the Jini security library.
7. The proxy connects to the service and runs the authentication and key agreement protocol. Thereafter the client request is sent.

It is not specified exactly how the Jini security library should be implemented and what properties it should have. This could be up to the system specification.

### A.7.7 Authentication for DHCP messages

In networks where DHCP is used to configurate a clients network properties some administrators way wish to constrain the allocation of addresses to authorized clients. Also a security sensitive client may want to authenticate the DHCP server, since there is a possibility that there exists malicious DHCP servers in the network. RFC 3118 [13] defines a new DHCP option whish increases the security when using the original DHCP protocol. The

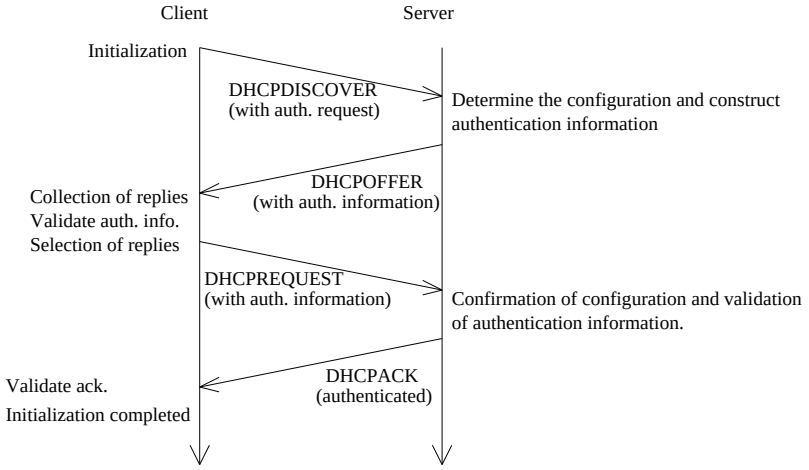


Fig. A.6: Client initialization using the authentication DHCP option

new protocol option does not attempt to solve the problem when clients can roam between different administrative domains.

All DHCP clients must share a secret key with the DHCP server. This key should have been distributed "out-of-band" before any DHCP messages are exchanged between the client and the server. The key is used to make the server able to authorize clients and to make the clients able to authenticate the server. Making the DHCP server hold all keys available in the system is not scalable. One solution to this is to use a trusted authority where both clients and servers can obtain keys or certificates.

The authentication option contains fields for replay detection and authentication of peers. In [13] the replay detection method is unspecified while the authentication method uses HMAC and MD5 [1].

When the shared key is possessed by both the client and the server the actual DHCP process can take place, see figure A.6. In the *dhcpdiscovery* message send by the client the authentication option tells the DHCP server that the client wants authorization and authentication. The DHCP server find the key it share with that client and construct the authentication information (using the shared key) which it passes along with it's *dhcpoffer* message. The client validates this authentication information. If validation is okay the client replies with a *dhcprequest* message, including it's own authentication information (using its shared key). The DHCP server validates the clients identity and replies with an authenticated *dhcpack* whish is validated by the client.

If all validations are okay from the view of the server and the client, the client really allocates network resources at the DHCP server.

The authentication method shortly described above is called "delayed authentication" in [13] and has at least two vulnerabilities. The DHCP server

could be flooded with *dhcpdiscover* messages and thereby exhaust the available network addresses. This is possible since the *dhcpdiscover* message is not authenticated. The second vulnerability is to flood the DHCP server with authenticated messages. In this case the DHCP server will be overwhelmed when it computes the authentication information it should send back for all incoming messages.

# Appendix B

# Acronyms

| | |
|---|---|
| BOOTP | Bootstrap Protocol |
| CSMA/CA | Carrier Sense Multiple Access/Collision Avoidance |
| DES | Data Encryption Standard |
| DHCP | Dynamic Host Configuration Protocol |
| DSA | Digital Signature Algorithm |
| DSSS | Direct Sequence Spread Spectrum |
| FHSS | Frequency Hopping Spread Spectrum |
| FTP | File Transport Protocol |
| GSA | Group Security Agent |
| GSC | Group Security Controller |
| GSI | Group Security Intermediaries |
| GUI | Graphical User Interface |
| HMAC | Hash MAC |
| HTTP | Hyper Text Transport Protocol |
| IEEE | Institute of Electrical and Electronics Engineers, Inc. |
| IETF | Internet Engineering Task Force |
| IP | Internet Protocol |
| IPSec | IP Security |
| IRC | Internet Relay Chat |
| JAR | Java ARchive |
| JSSE | Java Secure Socket Extension |
| LAN | Local Area Network |
| MAC | Message Authentication Code |
| MD5 | Message Digest, Version 5 |
| OSI | Open System Interconnection |
| PDA | Personal Digital Assistant |
| POP | Post Office Protocol |
| RFC | Request For Comments |
| RMI | Remote Method invocation |
| RMID | RMI daemon |

| | |
|---|---|
| RSA | Rivest-Shamir-Adelman |
| SET | Secure Electronic Transaction |
| SLP | Service Location Protocol |
| SSL | Secure Socket Layer |
| TA | Trusted Authority |
| TCP | Transmission Control Protocol |
| TFTP | Trivial FTP |
| TLS | Transport Layer Security |
| UDP | User Datagram Protocol |

# Bibliography


[1] W. Stallings. *Cryptography and network security: principles and practice, 2nd edition.* Prentice Hall, 1999.

[2] Larry L. Peterson, Bruce S. Davie. *Computer networks: a systems approach, 2nd edition.* Morgan Kaufmann, 2000.

[3] J. Schiller. *Mobile Communications.* Addison-Wesley, 2000.

[4] B. Eckel. *Thinking in Java, 2nd edition.* Prentice Hall PTR, 2000.

[5] A. Tanenbaum. *Distributed operatingsystems* Prentice Hall, 1995.

[6] Sun Microsystems. *Jini Architecture Specification.* http://www.sun.com/jini/specs/jini1_1.ps, October 2000

[7] Sun Microsystems. *Jini Technology Core Platform Specification* http://www.sun.com/jini/specs/core_1.ps, October 2000

[8] Jan Newmarch. *Jan Newmarch's Guide to JINI Technologies* http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml, June 2001.

[9] B. Croft, J. Gilmore. *Bootstrap Protocol.* RFC 951, Standford University and Sun Microsystems, September 1985.

[10] K. Sollins *THE TFTP PROTOCOL (REVISION 2)* RFC 1350, MIT, July 1992.

[11] R. Droms. *Dynamic Host Configuration Protocol.* RFC 2131, Bucknell University, March 1997.

[12] S. Alexander, R. Droms. *DHCP Options and BOOTP Vendor Extensions.* Silicon Graphics, Inc., Bucknell University, March 1997.

[13] R. Droms, W. Arbaugh. *Authentication for DHCP Messages.* RFC 3118, Cisco Systems, University of Maryland, June 2001.

[14] F. Stajano and R. Anderson. *The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks.* B. Christianson, B. Crispo, and M. Roe (Eds.)., Security Protocols, 7th International Workshop Proceedings, Lecture Notes in Computer Science, 1999.

[15] N. Asokan, P. Ginzboorg. *Key Agreement in Ad-Hoc Networks.* Computer Communications 23, pp. 1627-1637, 2000.

[16] M. Waldvogel, G. Caronni, D. Sun, N. Weiler and B. Plattner. *The VersaKey Framework: Versatile Group Key Management.* IEEE Journal on Selected Areas in Communications (Special Issue on Middleware) 17(8), pp. 1614-1631, August 1999.

[17] Suvo Mittra. *Iolus: A Framework for Scalable Secure Multicasting.* In Proceedings of ACM SIGCOMM '97, Cannes, France, September 1997.

[18] Chung Kei Wong, Mohamed Gouda and Simon S. Lam. *Secure Group Communications Using Key Graphs.* Proceedings of ACM SIGCOMM, Vancouver, British Columbia, September 1998.

[19] L. R. Dondeti, S. Mukherjee, A. Samal. *A Distributed Group Key Management Scheme for Secure Many-to-many Communication.* http://www.cs.umd.edu/users/saritm/publications.html, PINTL-TR-207-99.

[20] L. R. Dondeti, S. Mukherjee, A. Samal. *Survey and Comparison of Secure Group Communication Protocols.* http://www.cs.umd.edu/users/saritm/publications.html.

[21] L. Zhou, Z. J. Haas. *Securing Ad Hoc Networks.* 13(6), November/December 1999.

[22] P. Eronen, C. Gehrmann, P. Nikander. *Securing ad hoc Jini services.* Proceedings of the 5th Nordic Workshop on Secure IT Systems (NordSec 2000), pp. 169-177. Reykjavik, Iceland, October 2000.

[23] Bill Venners. *Inside the Java Virtual Machine.* http://artima.com/insidejvm/ed2/

[24] Sun Microsystems. *Permissions in the JavaTM 2 SDK.* http://java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html

[25] Sun Microsystems. *JavaTM Secure Socket Extension (JSSE) 1.0.2.* http://java.sun.com/products/jsse/doc/guide/API_users_guide.html

[26] Lars-Arne Mattson. *Secure Group Communication over Ad-Hoc Networks: Master Thesis Report.* http://www.docs.uu.se/~larssa/

[27] IBM. *Discovering Devices and Services In Home Networks.* IBM White Paper, June 1999.

[28] Phil Blundell. *Route.* Linux manual pages.

[29] Rusty Russel. *ipchains - IP firewall administration.* Linux manual pages.

[30] Rusty Russell. *Linux IPCHAINS-HOWTO.* http://www.linuxdoc.org/HOWTO/IPCHAINS-HOWTO.html

[31] Sun Microsystems. *jarsigner - JAR Signing and Verification Tool.* http://java.sun.com/j2se/1.3/docs/tooldocs/win32/jarsigner.html

[32] Sun Microsystems. *keytool - Key and Certificate Management Tool.* http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html