Secure Jini Services in Ad Hoc Networks

Fredrik Andersson and Magnus Karlsson

Stockholm, 2000

Master of Science Thesis
Royal Institute of Technology (KTH)

Supervisor
Christian Gehrmann
Communication Security Lab
Ericsson Researce

Examinator
Björn Pehrsson
Department of Teleinformatics

Abstract

Providing secure services in Ad Hoc networks is an important issue. Ad Hoc networks can in the near future, be created by using the Bluetooth technology. Due to the lack of security in today's applications, there is a lot of work to be done to secure Ad Hoc applications. In this work, secure Ad Hoc applications will be discussed. As well as how and where to implement security, in an Ad Hoc application.

Jini, which is written in Java, and its service "lookup" will be used as the basic programming technology.

Table of Contents

Preface

This report is the result of our Master of Science thesis done at Ericsson Research, Communication Security Lab in Kista, under supervision by the department of Teleinformatics at the Royal Institute of Technology in Stockholm.

During a presentation of the Bluetooth project at the Royal Institute of Technology we became interested in the Bluetooth technology and what made us really hooked up, was that it involved Java programming. We both felt that after four years of studying, it would be nice to do a practical Master of Science work and not a theoretical one.

We would like to thank: professor Björn Pehrson at the Royal Institute of Technology for his supervision in our master of science work and Christian Gehrmann at Communication Security Lab for his belief and encouragement in our work. Others we would like to thank is Annika Johansson for arousing our interest in this field.

Magnus Karlsson

e95_kar@e.kth.se

Fredrik Andersson

e95_fan@e.kth.se

Stockholm, February 2000

1 INTRODUCTION

To share services in open networks, such as Ad Hoc networks, some service discovery protocol can be used. An example of an Ad Hoc network technology is the Bluetooth technology, which is described in the first part of the report. Some of the available service discovery protocols are described as well. When using services in open networks, security issues arise. The service user must be able to use a service, knowing that the service can not do anything. These kinds of problems are not yet solved in the current versions of any of the available service discovery protocols.

As service discovery protocol we used Jini and implemented a simple service. Our solution is described in the second part of the report. Jini is built in Java and has no built in security. Hence, to be able to create secure Jini services, some additional programming considerations have to be taken into account. We need to design both the service provider and the service user correctly. To be able to describe the existing security problem and solutions to these problems, a simple fileserver was implemented. Examples of security problems connected to our application are:

- Authentication, how to make sure that a person is who he claims to be.
- Communication privacy, how to protect data from being visible to other users while transferred between the service user and the service provider in open networks, such as the Internet.
- Communication reliability. How to make sure that data has not been changed during transfer.
- Key management, how to transfer secret keys used to provide the security without showing them to others.

These security issues are discussed and solutions of the problems are given in part II of the report.

The Thesis is organised as follows:

In part I the background is given which is needed to be able to present our solution.

In part II our solution of how to secure a Jini service is presented.

In part III conclusions and future work is discussed.

2 BLUETOOTH

2.1 INTRODUCTION

The Bluetooth technology enables devices equipped with Bluetooth interfaces to in short range and wireless connect with each other and form an *Ad Hoc* network [1]. Each unit can simultaneously communicate with up to seven other units per *piconet*, i.e. a small network that only contains seven members. A unit can also belong to other piconets.

2.2 BACKGROUND

The idea of Bluetooth was born in the Ericsson Mobile Communications AB in Lund 1994. The goal was to find a cheap, small and power-efficient radio chip that could create a micro-scale web between mobile phones and their accessories. At that time, the only known and developed technique was infrared links (IrDA), but IrDA have many drawbacks in contrast to radio technique.

- Limited range (approximately one or two meters).
- Require direct line-of-sight.
- Can in principle, only be used between two devices.

Another criteria to make this product widespread, was to make it compatible with as many other electronic devices as possible. Therefore, a special interest group was put together. Its purpose was to urge other electronic manufacturers to support Bluetooth standard and establish a *de facto* standard for the air interface and the software that controls it.

2.3 RADIO INTERFACE

The initial requirements that was defined for the Bluetooth interface were:

- It must operate world-wide.
- The connection must support voice and data.
- The radio transceiver must be small and run on low power.

To be able to operate world-wide the frequency band must be license-free and open to any radio system. The only frequency band that satisfies these requirements is the 2.45 GHz band or the Industrial-Scientific-Medical (ISM) band.

2.4 PHYSICAL LINK

There are two types of physical links defined, which support multimedia applications:

- Synchronous connection-oriented (SCO) link
- Asynchronous connectionless (ACL) link

When transmitting voice, the SCO link is typically used. SCO supports symmetrical, circuit-switched, point-to-point connections.

ACL link handles typically peak-flow transmission and supports symmetrical or asymmetrical, packet-switched, point-to-multipoint connections.

2.5 INTERFERENCE

When choosing an open frequency band, the radio systems must cope with several uncontrolled sources of interference. The following optimisations have been done to prevent those sources of interference:

- Frequencies hopping with high hopping rate and short packet lengths.
- Forward error code.
- Using automatic-retransmission-query (ARQ) to achieve short delays and shorten transmission ratio, i.e. if an error is detected the receiver at once indicates that in the next packet.
- Never retransmit voice. Instead, a robust voice-encoding scheme is used.

2.6 NETWORKING

When two Bluetooth devices are within range, they can set up an *Ad Hoc* connection. To set up a piconet, one of the Bluetooth devices has to take the role as master. The master role is to control the traffic in the piconet. To achieve that, every piconet user uses the master built-in clock and the master's identity to keep track of hopping. The master also allocates capacity for SCO links by reserving slots. For ACL links, they use a polling scheme. The polling scheme states, that a slave only is allowed to send in the slave-to-master slot, when the slave is addressed with the MAC address in the slot header. This also eliminates collision between slave transmissions.

2.7 SCATTERNET

As discussed above, units within range can establish Ad Hoc connections between them. If one member of this newly created piconet also belongs to another piconet an overlapping

pattern will occur. If all members of the two piconets share the same hop-channel that will result in a drastically decrease of throughput in the network. Therefore, another solution was adopted – scatternet. This solution simply states that only devices that wants to communicate with other piconet members should be using the same hop channel. In spite of this approach collision do occur, because the piconets hop independently.

2.8 SUMMARY

Bluetooth is an uprising technology that has all the qualities to become a leading product. Today only one Bluetooth product exists and that product only support point-to-point connection, but a point-to-multipoint connection is under development and will soon be presented.

3 SERVICE DISCOVERY PROTOCOL

3.1 INTRODUCTION

Today exist different applications that discover a service in a network. We have chosen to discuss four of the most promising solutions. The first is the Bluetooth Service Discovery Protocol (SDP) which hardware background was discussed in the previous section. Thereafter, three software approaches will be discussed.

3.1.1 What is a service discovery protocol?

A service discovery protocol is used to find different services within a network. A service can be a device offering some kind of work to be performed. It could for example be a printer offering the service to print pages or it could be a file-server offering storage of files. A service can be nearly anything offered within a network.

3.1.2 Passive and active services

You could say that there are three different types of services. A service can offer some kind of work or it can provide you with information. The third kind of service can even allow changing of states in a device, such as switching on and off a lamp.

3.1.3 Service Discovery Protocol in pre-configured networks

Service discovery can be very useful in pre-configured networks, such as LANs. The service discovery can be used to find for example printers in the network, without the need of any pre-configuration of the computer or the printer. Other shared resources, such as CD and DVD-exchangers can easily be shared between users within the network. The idea with a discovery protocols is that there is not any need to management of new devices added to the network. Therefore, a device can easily be moved from one network to another, without the need of re-configuration. An example of this is a company's projector, used for presentations at the home office, but also at different customer offices.

The possibility to be able to access the company's file-server from a laptop without any pre-configuration could be of big interest for travellers. With use of a discovery protocol, such services can easily be built into the network.

3.1.4 Service Discovery Protocol in Ad Hoc networks

Ad Hoc networks are networks that are not pre configured and where devices are added and removed now and then. In these temporary networks, resources (services and clients) can appear as fast as they can disappear. In these networks, service discovery methods can

be of big help. A discovery protocol gives, for example, a user the possibility to print on the closest printer wherever he might be in the world. It might be at a place he has never been before, and at a printer-model he has never seen. With use of a discovery protocol he can use any printer-model. The service discovery protocol will make sure it works. In Ad Hoc networks, discovery protocols can make devices available to users without pre-configuration. Furthermore, the management of the networks will drastically decrease the use of a discovery protocol. As the world is getting wireless, the discovery protocol is one solution to solve some of the new arising network configuration problems.

3.1.5 What does a service discovery protocol need to provide?

A discovery protocol must provide the ability to share services between devices. Some kind discovery message is usually used to maintain the sharing. To be able to share services without overloading the network with announcement- and discovery-messages, commonly some kind of centralised service-server is used. This will reduce the traffic in the network and allow an administrator to manage the services if preferred. In the following section a deeper description will be given of a few discovery protocols and the security will be examined in each discovery protocol. Finally, a summary will be given where the different protocols will be compared with each other. The following Service Discovery Protocol will be examined:

- Bluetooth Service Discovery Protocol, SDP
- Simple Lookup Protocol, SLP
- Universal Plug and Play, UPnP
- Jini

3.2 BLUETOOTH SERVICE DISCOVERY PROTOCOL

3.2.1 Service Discovery Protocol

Bluetooth is a wireless connection device, which is using radio waves to connect different devices. To be able to connect different Bluetooth devices, it has a built in discovery protocol, the Bluetooth Service Discovery Protocol (SDP) [2]. SDP addresses service discovery specifically for the Bluetooth environment. It is designed to find services available from or through Bluetooth devices. SDP does not define any method for accessing the services. To be able to access the devices, other service discovery methods such as Jini, UPnP or SLP can be used. While SDP can coexist with these other service discovery protocols, it does not require them. Other ways to access the service might also be used depending on the service. In Bluetooth environments, found services can be accessed using other Bluetooth specific protocols.

3.2.2 How SDP works

SDP can be used to find devices without the need of any other service discovery protocols. SDP can for example be used to establish a connection to a remote Bluetooth device [3]. When this is done, the connection is established at the L2CAP-layer, se figure 3.1. The SDP protocol does not have any mechanism for how to utilise the service, such as delivering the service access protocols.

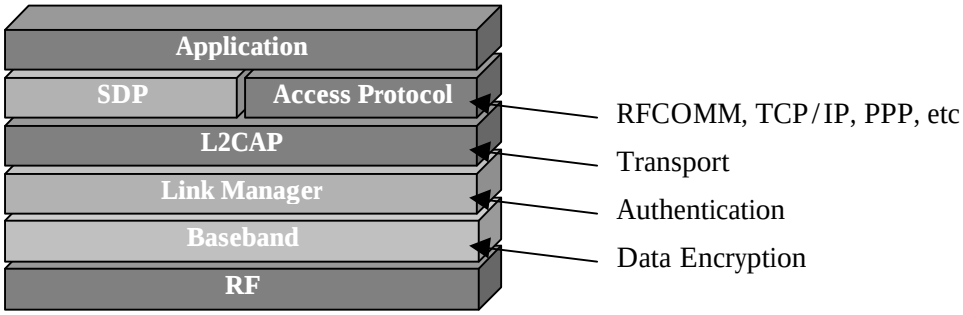

Figure 3.1 *The Bluetooth protocol stack [3].*

SDP servers maintain a database with information about existing services within the Bluetooth network. The server also responds to request on an existing connection. SDP clients can search for services in a specific class or for a specific service. Clients can also provide the ability to browse available services. A SDP service is any feature usable by another device. Services can be searched for as a specific class of services or it can be searched for from browsers.

3.2.3 Advantages and disadvantages

Here follows a short summary of some advantages and disadvantages with SDP.

\+ Fast and hardware cheap.

\- Works only between Bluetooth devices.

3.2.4 Built-in security

SDP does not care about security. It does only allow devices to locate other devices within the Bluetooth environment.

3.3 SERVICE LOCATION PROTOCOL

3.3.1 Service Location Protocol

SLP is defined by IETF in RFC 2165 [4]. The Service Location Protocol was constructed to provide a scalable framework for discovery and selection of network services. When devices use this protocol, the requirements on static configured networks decrease. Or as they describes it in the RFC "Using the protocol, computers using the Internet no longer need so much static configuration of network services for network based applications." [4] This is, as described in the RFC, very important as computers are becoming more and more portable and in the same turn, the network management has to decrease.

3.3.2 How SLP works

Usually a user finds a service by a host name (a URL) which is an alias for the network address [4]. This is not a convenient way to find services in a network when the computer doesn't know where they are located, or maybe not even know which services are present in the current network. The Service Location Protocol eliminates the need for a user to know even these basic things. Instead, the user asks the network for a service by its name and/or some attributes to describe the service. The Service Location Protocol then allows a binding of the description to the network address of the service.

The Service Location Protocol is built to work in local area networks. It is not a global resolution system for services within the entire Internet. All hosts within the multicast radius will be reached when searching for new services. The multicast radius is the same as the TTL (Time To Live) in the multicast advertisement. The default multicast-radius is set to 32. If a site does not support multicast, the request is restricted to that local subnetwork.

The SLP consists of three parties, the User Agent, the Service Agent and the Directory Agent. The User Agent is a process that is working on behalf of the user to receive service attributes and configuration from Service Agents or Directory Agents. A Service Agent is a process that is working on behalf of one or more services to advertise service attributes and configuration of the services. Directory Agents are processes that collect information from different Service Agents to be able to provide a single answer to requests from User Agents. It is used to centralise the information for efficient user access.

A Service is a process or system that is providing a facility to the network. The Service is accessed through its own communication protocol, unspecified in the SLP standard. A Service can be found by multicast requests, but also by unicast requests from User Agents. The multicast address is a service-specific multicast address, that all services, which provides that type of service must listen and answer to. Once a User Agent has found a Directory Agent, it saves its address and all future requests are sent by unicast to that Directory Agent. Answers are always sent by unicast messages to the User Agent.

3.3.3 Advantages and disadvantages with SLP

Some of the advantages and disadvantages with SLP are listed below:

+ Operating system independent

+ Simple protocol, therefor simple to implement

- Doesn't specify anything about how the services are created

- Provides only a simple way to discover services, not how to use them

3.3.4 Requirements

SLP is constructed to work over TCP/IP. To be able to work, it needs that the basic IP-protocols are supported in the network, such as: Multicast, TCP/IP and/or UDP/IP, preferably DHCP (Dynamic Host Configuration Protocol) shall exist in the network.

3.3.5 Built in security

The built in security is not complete, but some security-related considerations have been taken. The authentication problems are taken care of in the SLP protocol. To authenticate services in SLP for instance certificates may be used. There is a field in the SLP specification where the authentication block is located. Observe that this does not provide any kind of access control of services, only a way to make certain that the service comes from the service provider it claimed to be. Furthermore, there is nothing specified on how the communication is supposed to be secured.

3.4 UNIVERSAL PLUG AND PLAY

3.4.1 Universal Plug and Play

The Universal Plug and Play (UPnP) protocol is mainly created by Microsoft. It is supposed to work like Microsoft's Plug and Play, but is actually quite different [5]. The goal with UPnP is that it is supposed to be platform independent. To be able to achieve this, UPnP has been constructed to work at a lower layer than some other discovery protocols. It works directly at the TCP/IP layer. As a matter of fact, it doesn't even have to be at the TCP/IP layer. To be able to reach all kinds of transmission media, UPnP is constructed to work even over non-IP protocols, such as IEEE 1394 (Firewire), USB and several more [6]. This also allows even cheap and simple devices to run UPnP. UPnP is platform and operating system independent, which gives developers the possibility to choose the best platform for their device. The platform independence is provided by using the TCP/IP

protocol. Furthermore, it uses already standardised and reliable Internet-mechanisms. For example, small HTTP servers are used to send information about the device to the user of the service. The service information is transmitted in an already standardised form, as an XML-page (eXtensible Markup Language). To be able to use services at different operating systems, different APIs (Application Programming Interface) are used. Vendors only have to apply different APIs to be able to use their services at different operation-systems without the need of rewriting the devices source code.

UPnP is made to bring easy-to-use to home and office environments. It is to be used in local networks as well as at the global Internet. The UPnP forum describes UPnP as: "Universal Plug and Play embraces the zero-configuration mantra of Plug and Play but is more than just a simple extension of the Plug and Play peripheral model. Universal Plug and Play is an evolving architecture that is designed to extend the zero-configuration mantra to highly dynamic world of many networked devices supplied by many vendors." [5]

3.4.2 How UPnP works

An UPnP community consists of Client Components, Smart Objects and Directories [5]. Directory servers (also called proxy) are able to store object announcement and respond to client discovery requests. Directories are providing the ability to answer on behalf of different objects within the network. The directory server makes UPnP scalable, it works as a co-ordinator at the local network and as a global discovery mechanism which covers the entire Internet.

Smart Objects are devices, which are providing some kind of services within the network. When appearing in the network they are sending out an announcement packet in the network. If a directory is present, the smart object knows it does not need to answer any discovery requests from clients, because it knows the directory server will take care of them and answer them. But if there isn't any directory server within the network it must handle all discovery requests to see if the discovery match the description of its service.

Discovery is made by sending out discovery packets. Discover packets are sent with the Simple Service Discovery Protocol (SSDP) which is constructed to discover devices in IP network. SSDP uses UDP- and TCP-based HTTP to discover services.

UPnP is constructed to work in server-less networks. Therefore, all Smart Objects must be able to provide the capabilities necessary if there is no discovery server within the network. This means that they all have to be able to answer Service Discovery requests and they must have a built in HTTP server to be able to respond to requests.

UPnP does not define any programming model. Therefore, devices are operating system and program language independent. To get devices (Smart Objects) to interact with each other, specific APIs can be used. The usage of APIs allows a device to interact with other devices running at different operating systems. The API also allows the usage of different

transport mediums. This means that the discovery protocols does not have to run over IP based transport mediums.

3.4.3 Advantages and disadvantages

Here follows a summary of some of the advantages and disadvantages with UPnP.

+ Built on reliable, well-known technology.

+ No code is moved around or being downloaded.

+ The ability to use non-IP based networks.

- Device-interaction only through API or XML pages with XSL (Style Sheets).

- Hardware requirements (see Section 3.4.4).

3.4.4 Requirements

The UPnP has quite high demands on hardware. An implementation of Simple Discovery takes 4 Kbytes of code [5]. The handling of HTTP activities requires about 20 Kbytes of code and the device also need to implement the TCP/IP stack to support transportation. The device also have to implement the domain service that allows automated naming and generation of addresses, which takes another 40 Kbytes of code. Totally 64 Kbytes of code is needed only to be able to run as an UPnP device.

3.4.5 Built in security

In UPnP the security consideration is divided into three parts in UPnP [5]. The three key areas are:

- Authorisation: decides who is allowed to use the device and by which restrictions.
- Service deliverables: decisions about who has access to particular information to be able to secure the services for the subscribers. Some mechanism must be used to provide this.
- Encryption of data: to protect data while transmitting it over the network will be of major concern.

The UPnP protocol does not itself provide any of these security issues. It is up to the developer to implement the security. Security is likely to be an issue in the next version of

UPnP [5]. There is no built in security when this report was written, but security issues are in progress.

3.5 JINI

3.5.1 Jini

Jini is another service discovery protocol created by Sun Microsystems Inc [7]. Jini is written in Java and is an extension to the standard Java library. Jini provides a way to find services. Services are stored in centralised servers, called Lookup Servers. If preferred, the Lookup Servers can be managed by an administrator. One of the advantage with Jini is that it is operating system independent, because it is written in Java. The fact that services are written in Java allows them to be very flexible. Services can be any kind of small indicators, such as lamp-switches, to really complex devices, such as printers or copy-machines. Complex devices can provide the user, the client, with a complete control panel with buttons and different modules to control the device. There is no need of device drivers, because everything needed for the device is included in the downloaded service, called a proxy. The device drive isn t always needed to be downloaded, instead the service provides the client with an implementation which communicates back to the actual device in a way that the device understands and then performs the command its own proxy is sending.

Services can be found in different ways. Services can be browsed in Jini browsers or they can be attached to applications as a device drive. The later is done by requesting a specific type of service, or more precisely, a specific implementation of an interface.

3.5.2 How Jini works

The Jini community consists of three parts [7]: the actual server, the client and the Lookup Server. The server consists of the actual server that performs the service and a HTTP server. The part it will transmit is called a proxy. The proxy is a program that is able to perform a service, or it contains a protocol describing how to communicate with the real service (the server). The server is then performing the work that the proxy has told it to. The proxy is downloaded to the client as an implementation to an interface. The usage of interfaces is a standardised way to write programs in Java.

When a client wants to use a service, it sends out a multicast request, asking for an implementation of that specific interface. If the Lookup Server knows about any service implementing that specific interface, it answers the client with a list of addresses to the machines, called a Marshalled object (an instance of an object, containing an URL to the code), where the implementations can be found. The answer from the Lookup Server, the marshalled object, also contains an instantiation of variables to use in the downloaded service-proxy. When a client has decided which service, or more correctly proxy, to use, it downloads the proxy from the service-machine. The proxy code is downloaded with the use of an HTTP server at the server machine. The HTTP-server does not actually have to be

located at the server machine, it can be located at any machine within the network but preferably at the same machine as the service or at the Lookup Server machine. When the client has downloaded the proxy code, it is ready to use the service through the interface of the proxy. Jini will initialise the downloaded proxy with the variables from the marshalled object before the client uses the interface.

Lookup Servers can be run at any machine within the network. But there has to be at least one Lookup Server running in the network to be able to find services without knowing the location of a service in advance.

The Lookup Server, the client, and the services are normally written in Java, but do not actually need to be written in pure Java.

Devices can be written in nearly any language. Other languages for example can be used if the proxy talking to the device is using Corba or any other similar protocol.

The Lookup Server can be written in other languages as well. Clients can be written in any language as long as it has a Java-part that receives the proxy and takes care of the interaction with it.

3.5.3 Advantages and disadvantages

Here follows a summary of some advantages and disadvantages with the Jini technology:

+ Operating system independence through the usage of Java.

+ Any kind of services can be implemented, large flexibility of the service implementation.

- The need of a Lookup Server.

- The requirements of Java, which is hardware expensive.

3.5.4 Requirements

To be able to run a Jini service or client it requires some hardware [7]. A device must implement the TCP/IP and UDP/IP, preferably with DHCP. The network also must support multicast. To be able to use a service a specific client or some kind of service browser must be used. Furthermore, both the client and the server must run Java with the Jini-extension installed.

3.5.5 Built in security

In Jini versions 1.0 and 1.01 (released in December 1999) there are no concern about security. The security is an issue for the future, but is probably included in the next generation of Jini.

3.6 SUMMARY

The choice of Jini was made because Jini has an interesting approach of the ability to share resources in networks. Jini is a powerful competitor when considering the fact that it has a built in way to transfer device-drivers needed to use the services.

4 COMMUNICATION SECURITY

4.1 INTRODUCTION

Implementing security is not an easy task and often set aside. One way to structure different security models is to divide them after the OSI reference model and then compare them with each other. The underlying goal in this section is to motivate the application security model that we used.

4.2 FUNDAMENTALS OF LAYERED PROTOCOL ARCHITECTURES

To be able to understand fundamental security models, it is first crucial to understand the fundamental protocol architecture (readers familiar with network protocols can skip this section).

The main idea is to separate the protocols into independent pieces, so that each piece can be developed independently. The hardware and the software can be seen as a *protocol stack*, where each layer adds a specific service to the communication and each underlying layer is used by the upper layers. The two most common and standardised protocols are the Open System Interconnection (OSI) and the Transmission Control Protocol over the Internet Protocol (TCP/IP). These architectures are described below[8].

4.2.1 OSI Reference Model

The International Organisation for Standardisation (ISO) Open Systems Interconnection (OSI) model consists of seven layers.

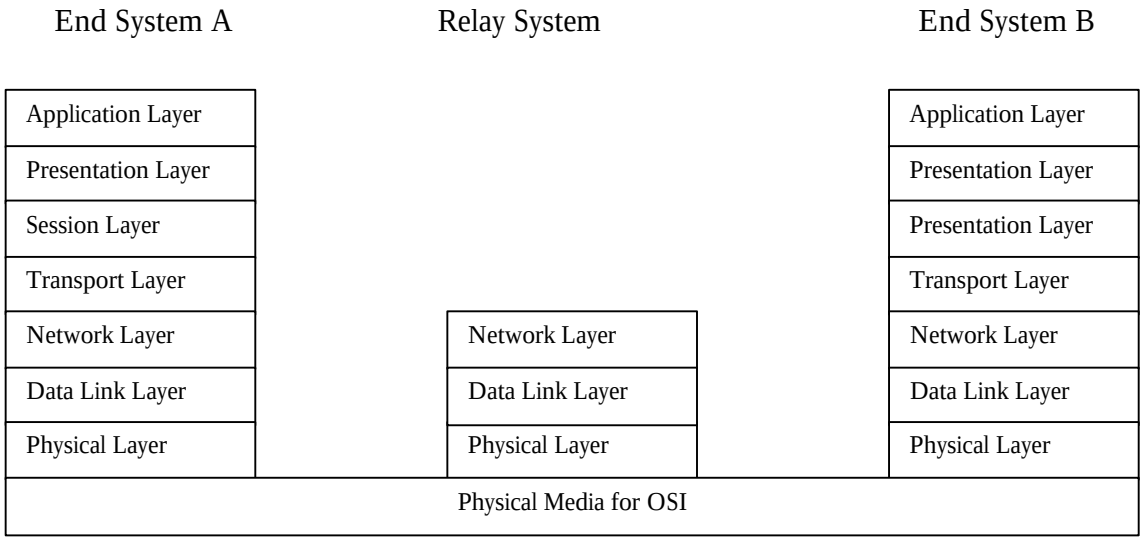

Figure 4.1 *The OSI reference model.*

Each layer in the end system is communicating with the other end system layer, i.e. the receiver side.

Application Layer:	User program handling management to access the OSI environment.
Presentation Layer:	Defines the data format for the communication between applications.
Session Layer:	Control structure for communication between applications.
Transport Layer:	Flow-control and end-to-end error detection and correction.
Network Layer:	Handles data transmissions and switching technologies used to connect systems and manage connection and routing.
Data Link Layer:	Reliable data delivery over physical medium.
Physical Layer:	Transmission of unstructured bit streams over physical medium.

4.2.2 The Transmission Control Protocol and the Internet Protocol (TCP/IP)

The TCP/IP Protocols is the most common protocols used on the Internet today. The protocols, was born in the mid-1970s Defence Advanced Research Project Agency (DARPA) project. There are no official TCP/IP protocol model and the model below is a summarise by William Stallings [8].

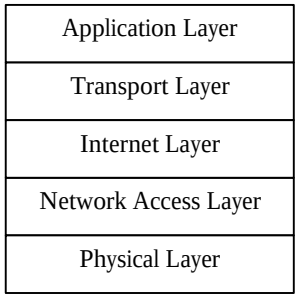

Figure 4.2 *The TCP/IP reference model.*

Application Layer:	Handles communication between application on separate host.
Transport Layer:	Handles end-to-end data-transfer.

Internet Layer: Handles routing of data.

Network Layer: Handles the logical interface between an end system and a subnetwork.

Physical Layer: Transmit bit streams over physical medium

4.3 WHERE SHOULD SECURITY BE IMPLEMENTED?

After having discussed different communication protocols the question is where should security be implemented related to the protocols. The answer to the question depends on what kind of application that is to be implemented.

The OSI Security Architecture (X.800) [9] provides guidance to which layer, security should be applied and what effect it has on the complete picture. Fourteen different security services are specified to the seven architecture layers, but they can be simplified into a four-level model:

Application Level: Security protocol elements that are application dependent.

End-System Level: Protection on an end-system to end-system basis.

Network Level: Protection between two sub-networks.

Link Level: Protection over a link between two nodes in the network.

The different levels are illustrated with bold lines in the figure 4.3.

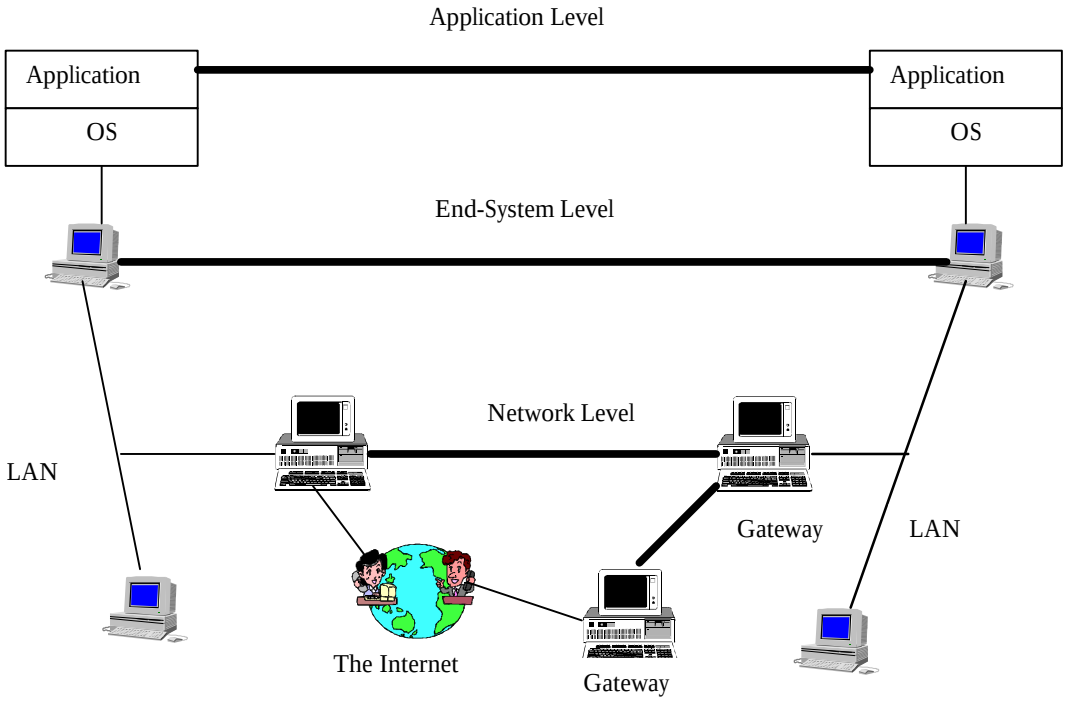

Figure 4.3 *Security implementation in different OSI Layers.*

4.3.1 Link Level Security

Link Level Security secure a single link between typically two gateways and applies security at the Link Layer in the OSI and TCP/IP model. This model is more appropriate in a relatively trusted environment, where the insecure links are few.

The advantage with applying security on the Link Level is that a high level of protection does not require a high equipment cost. It is also easy to make the security transparent to all higher layers, since it is not dependent on any particular network protocol.

If you have an environment, where there are many insecure links, that should be protected, the link level security model is no longer so well suited. Due to this, each link must be managed separately, which leads to high operational costs. Another disadvantage is that it does not protect against attacks from within a sub-network.

4.3.2 Network Level Security

Network level security provides protection across one or more sub-networks. This protection is applied at the network layer in the OSI model. This model is appropriate, when the sub-network closest to the end-system is considered trusted, and the intermediate

network is not. This approach is cheaper than an end-system protection, because the number of gateways is far less compared with the end-system solution.

This security model is often used, when creating a *virtual private network* (VPN). That is when one creates secure tunnels between trusted nodes in a hostile environment.

The Internet Engineering Task Force's (IETF's) IPsec (RFC2411) working group has proposed a standard for VPNs. By this standard, comparability could be achieved between the many different VPN vendors.

4.3.3 End System Level Security

End-system level security provides protection between two communicating end-systems and relates to either Transport Layer or sub-network independent Network Layer protocols. This model is appropriate, when all underlying layers are unsecured. Advantages with this model compared with an application level solution are:

- Transparent to all application.
- Protection for header information. (Transport, Session and Presentation layer headers)
- Better performance, i.e. can handle data of multiple applications in a common way.

Secure Socket Layer (SSL) is a Transport Layer security protocol and is drafted by Netscape Communications. It provides end-system level security. SSL is often used on the Internet today, for example when paying bills to your Internet bank or for credit card transaction.

SSL was never drafted as RFC [10], but version 3 is serving as the basis for the Transport Layer Security (TLS) protocol and is described in RFC 2246.

4.3.4 Application Level Security

Application level security not quite defined, it could involve both Application Layer and Presentation Layer in the OSI model. The advantage with this security model is that it has lower equipment and operational costs. There are two situations when this solution is the only possible way:

- When the security is built into a particular application protocol.
- When application communications must pass application relays.
- The Application Security model is common where there is more than two end-systems involved. Examples of such applications are:

- Privacy Enhanced Mail (PEM) (RFC 1422).
- Pretty Good Privacy (PGP) (RFC 1991).
- Secure Multipurpose Internet Message Extension (S/MIME) (RFC 2632).

4.3.5 Summary

Here follows a short summary of the advantage and the disadvantage of placing the security at the application level, as we have done in our application:

\+ Security models where each user wants to authenticate other users are hard to achieve with low-level security solutions.

\+ If security policy leaves the user to require protection or if different security levels are wanted, then the application security solutions tends to be better.

\- High level security requires that security is implemented in every end-system.

\- High level security does not protect the lower layer headers. This can be used to achieve traffic information.

4.4 SUMMARY

As we have seen implementation of security is quite complex and it depends on the application, the security policy, and the environment as a whole.

5 CRYPTOGRAPHIC TECHNIQUES

5.1 INTRODUCTION

Encryption is the most important tool to secure a communication. Cryptographic methods can be divided into two groups – Symmetric and Asymmetric Encryption [11]. These two techniques can then be combined to various security services. This section will give an introduction to these two techniques and the combination of them. Thereafter will common cryptographic technologies be discussed.

5.2 FUNDAMENTAL ENCRYPTION

Encryption is an algorithm that transforms a data-item, called *cleartext* or *plaintext*, into an unintelligible data item, called *ciphertext*. The cryptographic algorithm must be *invertible*, i.e. there must exist a matching cryptographic algorithm that can reverse the encrypted ciphertext. The encryption-algorithm uses an input parameter – a *key*.

The encrypted data should be kept to a minimum, so that it can easily be distributed.

The most common way to encrypt data is to use a combination of substitution, transposition, and permutation. A newer technique is to use elliptic curves.

5.3 SYMMETRIC ENCRYPTION

Symmetric cryptographic algorithms use the same key for encryption and decryption. These algorithms are also called *secret-key algorithm*. Before the sender sends encrypted data, both parties have to agree on a key. The security of a symmetric algorithm lies in the symmetric key. If one of the secret keys is revealed, then all the encryption is also revealed.

Common symmetric algorithm is *Data Encryption Standard* (DES).

5.3.1 DES, Data Encryption Standard

In a symmetric algorithm, the data is divided into blocks and then each block is encrypted. How data are divided and later encrypted is described in FIPS PUB 81. FIPS PUB 81 draw out four modes:

- Electronic Codebook (ECB) mode: Operates like a "code book", where every block of "plain text" maps to exactly one block of "cipher text". This is usually the highest performance mode, but suffers under the problem that when the same pattern appears, it is always encrypted in the same way.

- Cipher Block Chaining (CBC) mode: CBC is similar to ECB, except for the fact that each encrypted block is XORed with the previous block of ciphertext. Thus, identical patterns in different messages will be encrypted differently, depending upon the difference in previous data.

- Cipher Feedback (CFB) mode: CFB is similar to CBC in the way that following data is combined with previous data so that identical patterns in the plain text result in different patterns in the cipher text. The difference is that data is encrypted one byte at a time. Each byte is encrypted along with the previous 7 bytes of ciphertext. This mode is useful for data streams, such as terminal sessions, where data arrivs one byte at a time.

- Output Feedback (OFB) mode: This mode is almost identical to CFB, except that 1-bit errors in the ciphertext cause only 1-bit errors in the plain text (instead of ruining the remainder of the stream). However, it is not as secure as CFB.

5.4 ASYMMETRIC ENCRYPTION

An Asymmetric encryption algorithm, usually called *public-key cryptography*, use different keys for encryption and decryption. The two keys are *public key* and *private key*. Either can encrypt and decrypt data. A user gives his or her public key to the other users and keeps the private for him- or herself. Data encrypted with the public key can be decrypted only with the corresponding private key and vice versa.

This technique simplifies the key-distribution, because the other part does not want or does not need to know the corresponding part's private key, which is the opposite case concerning the symmetric encryption.

5.4.1 Public-Key Cryptography Standards (PKCS)

RSA Data Security Inc. has published a number of standards called PKCS. The standards consist of a number of components, called PKCS #1, #3, #5, #6, #7, #8, #9, and #10 [12].

5.4.2 Public-Key Cryptography

RSA is a public-key algorithm invented by Rivest, Shamir and Adleman. The algorithm is as follows.

Public Key

n product of two primes, p and q (p and q must remain secret)

e relatively prime to (p - 1) (q - 1)

Private Key

d $e^{-1} \bmod ((p - 1) (q - 1))$

Encryption

$c = m^e \bmod n$

Decryption

$m = c^d \bmod n$

5.4.3 The PKCS standards

Here follows are short description of the two most important standards that are involved in Java.

PKCS #7 is a widespread standard and used in many programs. It describes a general syntax of how to structure data in a cryptography application. Methods that use this standard are digital signatures and digital envelopes.

PKCS #8 describes syntax for specifying private-key. The private-key information is included in some public-key algorithm. PKCS #8 also describes syntax for encrypted private keys.

5.5 HYBRID TECHNIQUES

The Hybrid method combines both the symmetric and the asymmetric method. The reason for this is performance, since public key encryption and decryption is about 100-1000 times slower than a symmetric algorithm, with comparable security. Due to that, the public-key algorithm involves heavy computation.

The Hybrid techniques take advantage of the simplified asymmetric key-distribution and the asymmetric encryption.

5.6 MESSAGE DIGEST

A message digest is a string with fix length representing a set of data that can be of arbitrary length [13].

It should be computationally infeasible to find two different messages with the same message digest, or to find a message with a given, predetermined message digest.

Two common message digests invented by RSA Data Security, Inc. are *MD2* and *MD5*. They take an arbitrary message and generate a 128-bits message digest. One other common and often used message-digest algorithm or *hash algorithm* is SHA (Secure Hash Algorithm).

5.7 SIGNATURE

A signature is a specific mark for a set of data. The receiver of the data can then verify the signature and by doing that, he can be certain that the data was not altered during transmission and that it was really sent by the claimed sender.

The signature is generated with a *signature algorithm*. Typically, it is first a message digest computed on the message and then the message digest is encrypted with a private key.

5.8 CERTIFICATE

A certificate is a way to establish trust. A certificate binds a public key to a certain user identity and is then signed by a third independent part or by a part trusted by both the other parts. One of the most common certificate-format is X.509 [11], see figure 5.1. It has been recommended for use with the ISO authentication framework. Although no particular algorithms are specified for either security or authentication, the specification recommends RSA. There are provisions however, for multiple algorithms and hash functions. X.509 was initially issued in 1988. After public review and comment, it was revised in 1993 to correct some security problems.

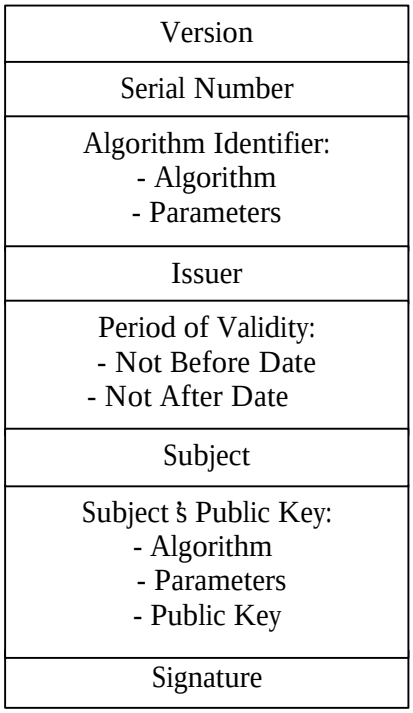

Figure 5.1 *A X509 certificate.*

5.9 CERTIFICATION REVOCATION LIST (CRL)

Normally, a certificate has a validity period, which lasts from a month to a couple of years. After the validity-period ends, the certificate expires [14].

A *revoked* certificate, is a certificate that has lost its validity before the validity period has ended. One reason could be that the key has been compromised and the certificate therefore must be revoked. Another reason might be that the certificate holder changed jobs.

Certificate revocation is an important issue for all public-key implementations. The revocation is often accomplished by posting a *certificate revocation list* (CRL) in some directory.

5.10 ENCODING RULES IN X.509, PKRC, ASN.1 AND BER

There is a notation for expressing the syntax of objects and message called *Abstract Syntax Notation One* (ASN.1) [15] defined in X.208 and the standard ISO/IEC 8824. It describes the notation for abstract types and values, i.e. BIT STRING, INTEGER and NULL. These types are relevant for the PKCS (see Section 5.11) standards and others. How to encode these types into octet strings of bits is defined in *The Basic Encoding Rules* (BER). BER is defined in X.690, ISO/IEC 8825-1. Unfortunately, BER allows multiple representation for some values. Therefore, *Distinguished Encoding Rules* (DER) is defined in X.509, which is a subset of BER [16].

ASN.1 is widely used to describe security protocols, interfaces, and definitions of services.

5.11 KEY-AGREEMENT

Key-agreement is a method whereby two parties, without prior arrangements, exchange messages in such a way that they agree upon a secret key that are known only to them. Key agreement can be achieved with a public-key algorithm or with other methods. A *key-agreement algorithm* is an algorithm for achieving key-agreement.

Diffie-Hellman is a key-agreement algorithm invented by Diffie and Hellman involving exponentiation modulo of large prime number [17]. The difficulty of breaking Diffie-Hellman is generally considered equal to the difficulty of computing discrete logarithms modulo a large prime number. Below follows a short description of the Diffie-Hellman key-exchange algorithm.

5.11.1 Diffie-Hellman

Here follows how Diffie-Hellman works for two users – A and B. A and B agree on two large prime numbers, *n* and *g*. This could be done via an insecure channel or the numbers could be common among a group of different users.

1. A starts with choosing a random large integer *x* and sends B

   $X = g^x \bmod n$

2. B chooses a random large integer *y* and sends A

   $Y = g^y \bmod n$

3. A computes

   $k = Y^x \bmod n$

4. B computes

   $k' = X^y \bmod n$

The fancy thing is that both k and k ’are now equal, i.e. $g^{xy} \bmod n$ and thereafter used as a secret key.

5.12 SUMMARY

Most of the cryptographic technique mentioned above are quite old and there are few new cryptographic algorithms that have been standardised.

In the future, the market will be changed, because it is quite likely that the USA export regulation to Europe will be more liberal and that will make the business climate for the today's supplier of Europe more harder.

6 JAVA 2 SECURITY

One of the advantage with our implementation is that all security issues are resolved with Standard Java. Before describing the case study, the general Java security architecture must be made clear.

6.1 INTRODUCTION

Sun has put a lot of effort into improving the security model that is shipped with JDK Java 2 Standard Edition (SDK). The biggest difference between Java Development Kit (JDK) 1.1 and SDK 1.2, is in the security area. First, the old security model, *The Sandbox Model*, will be discussed and thereafter, only the security model that comes with SDK 1.2 will be focused on. The reasons are not to make the reader confused and the fact that the older security model has been obsolete, after the introduction of Java 2.

One disadvantage for people that does not reside in the United States or Canada is that they are not allowed to access the Java Cryptography Extension (JCE) 1.2 [18]. Therefore, developers are forced to extend SDK with an own cryptographic library.

6.2 THE OLD JAVA SANDBOX MODEL

The first security model was designed to restrict remote code, downloaded from the Internet and let the remote code run "inside" this sandbox. The model distinguished between two types of codes - remote code and native code [19]. The policy was that remote code was recognised as not safe and local code as trusted. Trusted means that code has access to vital system resources (such as the file system), see figure 6.1.

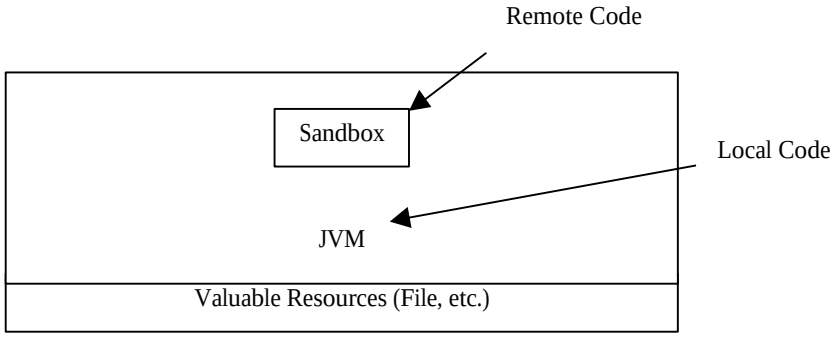

Figure 6.1 *JDK 1.0 Security Model.*

One overall difference between building a security model in Java compared with building one in C or C++ is that Java is designed to be type-safe and easy to use. The goal is to relieve the burden from the programmer, so that the risk of making subtle mistakes is less

than it is in C or C++. Language features, such as automatic memory management, garbage collection, and range checking on strings and arrays are examples of how the language helps the programmer to write safe code.

Second, compilers and a bytecode verifier ensure that only legitimate Java bytecodes are executed. The bytecode verifier, together with the Java Virtual Machine, guarantees language safety at run time.

Moreover, a classloader defines a local name space, which can be used to ensure that a not trusted applet cannot interfere with the running of other programs.

Finally, access to crucial system resources is mediated by the Java Virtual Machine and is checked in advance by a SecurityManager class that restricts the actions of a piece of not trusted code to the bare minimum.

6.3 THE JAVA 2 SANDBOX MODEL

The new model described in Java 2 SDK security specification can be divided into four categories and these new features are illustrated in the figure 6.2.

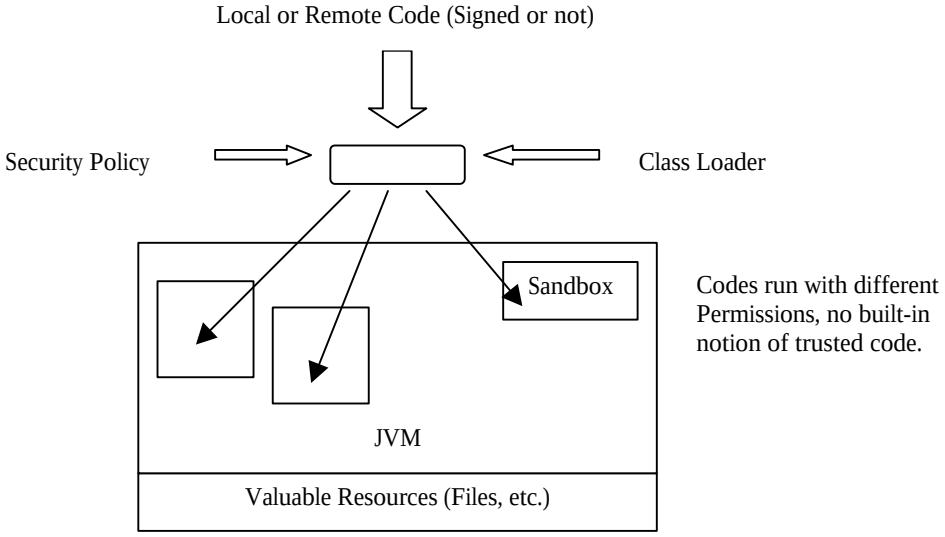

Figure 6.2 *Java platform security model.*

The new features are summarised in Java 2 SDK security specification into four groups:

- Fine-grained access control.

- Easily configurable security policy.
- Easily extensible access control structure.
- Extensions of security check to all Java programs, including applications as well as applets.

The Fine-grained access control already existed in the JDK 1.1, but to use it demanded substantial programming (e.g., by sub-classing and customising the SecurityManager, ClassLoader classes etc.).

To have such an extremely security-sensitive security-model, made it quite insecure and easily let security holes be made. The new architecture will make this more difficult and safer. The simplicity with which one can configure the security policy was already implemented in the old security mode, but it was not easy to use. The old security model was clumsy, when it came to a straightforward structure of the access control. The new architecture allows typed permissions (each representing an access to a system resource) and automatic handling of all permissions (including yet-to-be-defined permissions) of the correct type.

There is no longer a built-in concept which says that all local code is trusted. Instead, local code (e.g., non-system code, application packages installed on the local file system) is subjected to the same security control as applets. The same principle applies to signed applets and any Java application. Thus, the old security model is obsolete and should not be preferred.

6.3.1 Overview of the Protection Mechanism

The most fundamental in the Java security model is the protection domain [20]. A domain can be divided into a set of objects which access is monitored by different permissions. The relationship is illustrated in figure 6.3.

The use of protection domains makes it easy to group and isolate units with different permission.

Protection domains is divided into two camps - *system domain* and *application domain*. The system domain is the only one that has access to important valuable resources, e.g. the file system, network facilities, etc.

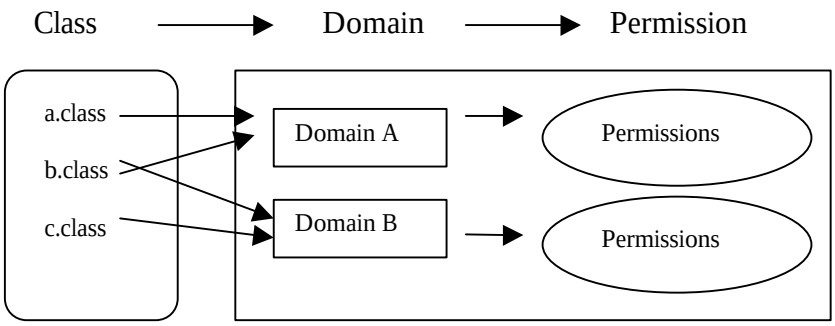

Figure 6.3 *Mapping chain.*

It is crucial, that threads that are executed over one or more protectiondomain, does not receive more permission. Hence, a less "powerful" domain cannot gain additional permissions just by calling or being called by a more powerful domain.

When access to critical recourses is requested, the AccessController [21] (see Section 6.3.4) class is consulted. The AccessController decides, based on the domains permissions, if the request is to be allowed or denied.

6.3.2 Permission

All permissions are defined in different classes and those classes are sub-classed from the abstract java.security.Permission [22] class.

The complete list of all permissions can be viewed at Java 2 SDK Security Guide [23].

Setting proper permission is not an easy task. Suppose that an applet has been granted the permission to write to the entire file system. This will indirect allow the applet to remove all security barriers, i.e. the applet have all permissions.

Other permissions that are "dangerous" to give out include those that allow the setting of system properties, runtime permissions for defining packages and for loading native code libraries.

6.3.3 The Java Policy

The policy file is consulted every time a class is loaded and states which permissions a certain domain has.

The policy file group code by their java.security.CodeSource [24]. A codesource is an absolute path specifying the physical location of the files. The file could be located either

locally or remotely, i.e. the codesource is an URL. Each codesource is then associated with a set of permissions.

All sets of codesources can also be signed. That is done by assigning a certificate attribute to each codesource.

Whenever a Security Manager is installed, the default system-policy file is loaded beside the user-defined policy file. In most cases, this is harmless, because it only to specify which permission Java system files has.

6.3.4 Access Control Mechanism

When code wants access to system resources the java.security.AccessController is invoked. The AccessController takes a "snapshot" of the current calling context and then base the access decision on the context.

The context is the codes *Java.security.ProtectionDomain* [25]. The ProtectionDomain is formed by the domain and optional also a set of certificates [26] containing the public keys that corresponds to the private keys that all code in this domain is signed with.

When the code starts a new process, the parents security context is inherited by the child. The child is allotted its security context when it is created and not when the child first run.

This loading proceeding is a general principal. All code permissions, i.e. their ProtectionDomains are loaded on demand basis.

6.3.5 Secure Class Loading

The class that handles the loading of classes in Java 2 SDK is java.security.SecureClassLoader [27], which is a subclass to the old loader in JDK java.lang.ClassLoader [28].

Another important loader is java.net.URLClassLoader [29] and it is a subclass to SecureClassLoader.

However, the SecureClassLoader must also be loaded and that is handled by a *primordial classloader* and it is bootstrapped to the Java Virtual Machine. The primordial classloader is written in C.

6.4 SUMMARY

Readers that are interested in details of which specific security methods that are implemented in Java 2 SDK can read the Cryptography Specification [30].

A interested reader, that wants to customise their default Policy should look into the following web page [31].

7 THE JINI-PROXYMODEL

7.1 INTRODUCTION

Our implementation consists of two parts – the Jini-proxymodel and our security solution. First the Jini-proxymodel will be discussed and then our security implementation will be discussed.

7.2 THE JINI-PROXYMODEL

The Jini-proxymodel consists of three parties: the service, the client and the Lookup Server [7], se Figure 7.1. Services can offer any kind of work to clients. They can be offered by any kinds of devices. In the Jini community, the client is the actual service user. Clients can be any kind of devices having a built in processor and communication facility. To be able to share services and to be able to find services in the network, a Lookup Server must be running in the network, se also section 3.5. A breath introduction to these three parts will follow.

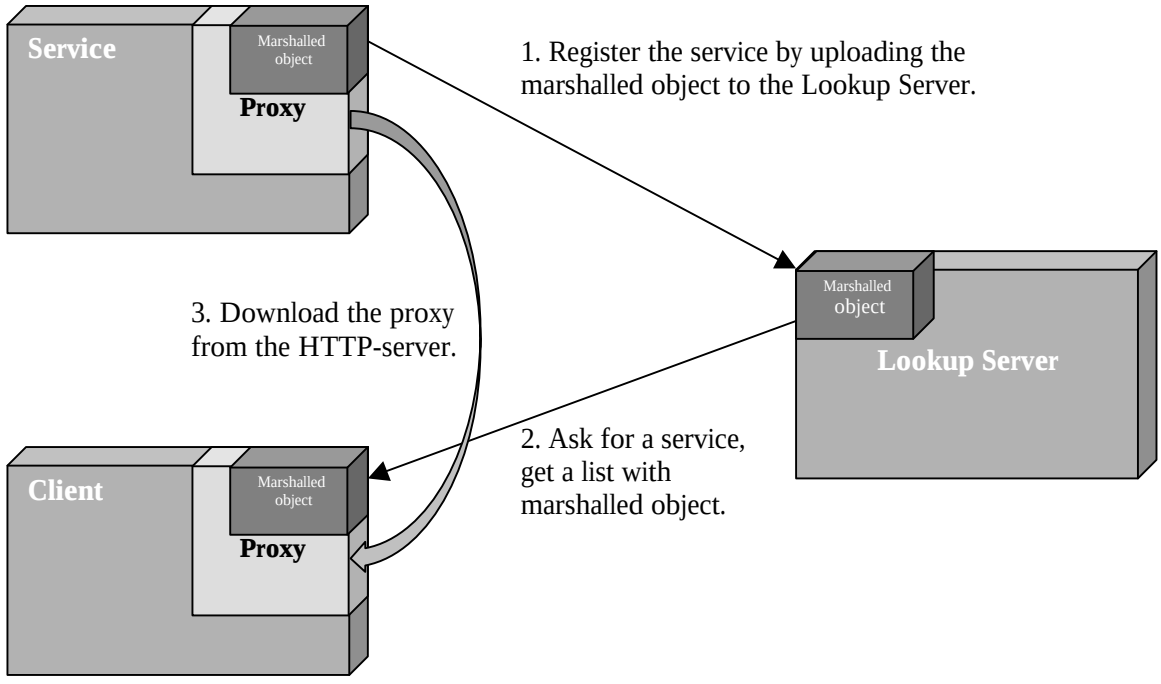

Figure 7.1 *The three Jini parts.*

7.2.1 The service

A service can be offered by any device in the network [7]. The service can be of many different types. Some services offers a work to be performed, others might only offer some kind of information-sharing. A service usually consists of a server and a proxy. The proxy is sent to the client to communicate back to the actual server. To be able to provide a service, the device has to have a Java program, which with help of the Jini-extension can upload the service to the Lookup Server. The service must implement an interface, specified by the service-type. There are no predefined interfaces yet, but this might change when Jini-services are getting more common. The registration of the service consists actually of an interface, along with some other parameters. The registration object consist of:

- the interface (which the service implements)
- optionally attributes (which describes the service and its capabilities with words)
- the marshalled object (the instantiated variables in the proxy and the URL to the HTTP server where the proxy-code can be downloaded from)

A proxy is the code that the client must use to utilise the service. The proxy is downloaded to the client from an HTTP server [7]. The proxy must contain either a program, which consists of the actual service offered, or contain an implementation that communicates back to the actual service machine which provides the service. The communication with the actual service can be done with RMI (Remote Methods Invocation) or some other service defined protocol. The communication can also be done in Corba or some other programming-language independent protocol. This will allow services to be written in any language, using any operating system, as long as the proxy is written in Java.

The proxy implementation must be instantiated at the service provider's machine before the service can be registered at the Lookup Server. This is done in the service by creating an object of the proxy implementation. This object will after it has been created, consist of all instantiated variables required to run the proxy-code at the client. The instantiated variables are saved in a marshalled object along with the address where to find the proxy-code. This marshalled object is then uploaded to the Lookup Server when registering the service.

The service registration is done by sending out an announcement message. The announcement message is sent to the default multicast address 224.0.0.84, port 4160. When some Lookup Server has answered a announcement message, the marshalled object is uploaded and registered at that Lookup Server (no 1 in figure 7.1). The service is now registered in the network.

7.2.2 The client

When a client is looking for a service, it is asking for an implementation of a specific interface [7]. The client must know how to use the interface and which functions that is available through that specific interface. When the client wants to look for a service, it

multicast a discovery message at the multicast address 224.0.0.85 port 4160. The discovery message contains the interface that the service must implement. When a Lookup Server answers, it does so with a list of the matching services available in the network. The client has to decide which service to use, if there are more than one service offered. When the client has decided which service to use, the client selects its corresponding downloaded marshalled object from the Lookup Server (no 2 in the figure 7.1). The marshalled object contains a URL where the implementation code of the proxy can be found. The client then downloads the proxy-code using the HTTP server at the service machine (no 3 in Figure 7.1). The downloaded code is finally initialised with the variables included in the earlier received marshalled object. The service is then ready to be used through its interface. Se figure 7.2.

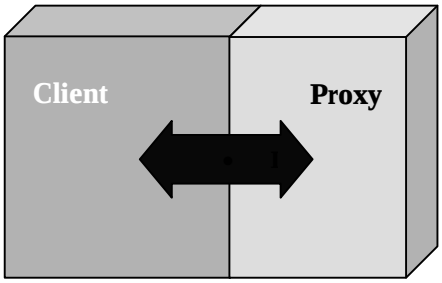

Figure 7.2 *The client talks with the proxy through its interface.*

7.2.3 The Lookup Server

The Lookup Server is used to share information about services and their location within the network to the clients [7]. Before this can be done, the service needs to register its services and at the Lookup Service. The Lookup Server is continuously listening for announcement messages from services. Services are sending announcement messages when they want to register their services in the network. The announcement message is received by the Lookup Server at the multicast address 224.0.0.84, port 4160. When a new announcement message is caught, the Lookup Server stores the information about the service location together with a copy of all instantiated variables in the service. This information is stored as a marshalled object. A marshalled object contains the URL (the address to the service) and variables stored as an instance of a serialised object. A service always has to implement an interface. The type of the interface the service is implementing is combined and stored with the service. Services can also register attributes, which is a description of the service it is offering. Services can therefor also be found by attribute matching, instead of the more common usage of the interface matching.

The Lookup Server also continuously listens for discovery messages from clients that are looking for services. The discovery messages are received at the multicast address 225.0.0.85,

port 4160. When a discovery request is received, the Lookup Server answers the request with a one service or a list of services that implement the specific interface or that match the attributes requested. The list contains information about the location of the service and an instance of the variables to use in the proxy. This is the marshalled object that earlier was uploaded by the service.

7.3 SUMMARY

The Jini-model could be implemented in many ways. Three different implementation methods are presented below.

One option is to push the entire implementation of the service to the lookup service and make the client ask for it by its class. This is a rather stupid method, because the client could have done all the implementation itself.

Another option is to let the client ask for a superclass of the service and then work as a server. This is not ideal, because the client will not be notified during runtime, if there are any updates of the server.

The Jini-proxymodel on the other hand, allows both the service to be updated during runtime and the server to be changed during runtime. Therefore was the Jini-proxymodel best suited for our application.

8 OUR SECURITY SOLUTION

8.1 INTRODUCTION

The next step in our thesis is to add security to the Jini-proxymodel. In figure 8.1 we have showed the Jini-proxymodel together with our security-model.

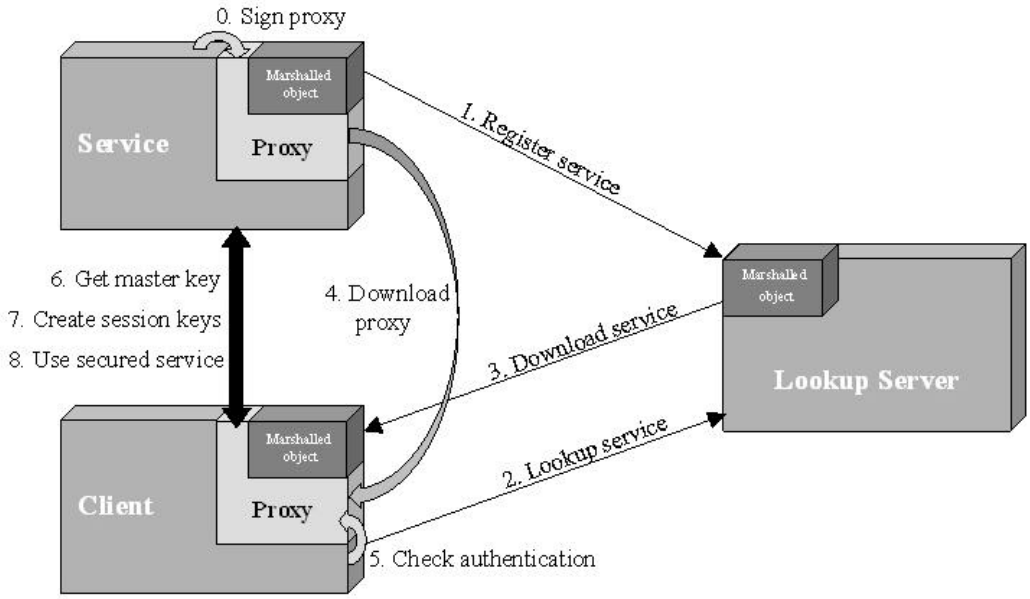

Figure 8.1 *Our security-model applied to the Jini-proxymodel.*

Our security-model and the Jini-proxymodel consist together of 9 steps, where step 1, 2, 3, 4 and 8 in figure 8.1, is the standard Jini-proxymodel (with no security). Our security-model involves steps 0, 5, 6, 7 and 8. The explanation of all the 9 steps is described below.

0. *The server bundles the proxy code in a jar file and signs it.*
1. *The server registers the service at the Jini Lookup Server.*
2. *The client makes a Lookup to find the service.*
3. *The client downloads service from the Lookup Server.*
4. *The client downloads the proxy from the Service Provider's HTTP server.*
5. *The client checks the authentication of the proxy code by checking the certificate's fingerprint, which the code is signed with.*
6. *The client fetches the master key from the server with use of an authenticated key exchange algorithm included in the proxy.*

7. *The proxy creates session keys to use in the secured communication.*
8. *The client uses methods at the service securely through the proxy interface.*

Since we were two persons that implemented the security-solution, our security-model was divided into two parts:

- The service (the server and the proxy) implementation
- The client implementation and the key and fingerprint Management

The first section will concern the server's and the proxy's part of the security-model, i.e. step 0, 1, 4, 6, 7 and 8.

The section after that will concern the client's part of the security model, i.e. step 5 and 6 and how all the necessary keys and fingerprints (see Section 8.2.4), used by the client and the server, are managed.

8.2 THE SERVICE IMPLEMENTATION

8.2.1 Introduction

In this section, it is described how the server is constructed to provide the secured service. The service described in the section is the secure file service. A file-server offers the service to store and share files between clients. The sharing is done through uploading and downloading files to the shared file server. The service is used in the clients through the proxy, which the service provides. The security is extremely important in a service as the file server. Information in the file server must be protected so that no other people than the allowed will get any kind of information about existing files and their content. However, in the user perspective of using the service, it should not add any extra overhead when using it. The sharing of the service therefore has to be extremely carefully implemented. The user has to be able to easily access the service while it has to be completely protected from unwanted access.

8.2.2 Security

When a client wants to use a service, it must be sure that it is a friendly service. The only way to make sure that it is not using a hostile service is to only trust services it *knows* is friendly. This can be really tricky when talking about discovery protocols in Ad Hoc networks. Therefore, a way to decide if the service is a hostile service or if it is a friendly service has to be constructed. If this is not done, there is no way to protect the computer against being harmed. In the same moment as a client has downloaded the service and initialised the downloaded code it can be too late. The client's hard-drive might already be deleted while it is trying to decide whether to trust the service or not.

The Jini extension does not provide any way of finding out if a service is trusted or not. There luckily exists a way to find out if a Jini service is from a friendly person or not. This is done with use of authentication. When downloading a service you can decide if it is a friendly or hostile service by authenticate it.

8.2.3 Step 0 and 1: Authentication of the service

To be able to authenticate a service, it has to be digitally signed. To be able to sign a service, or more exactly its proxy, it has to be bundled in a specific way. This is done in what Java calls a .jar-file. A .jar-file is recognised by its .jar extension of the filename. A .jar-file is one or more .class-files compressed into one single file. A .class-file is the result of a compiled source-code, a .java-file, written in Java. The .jar-file is used in Java for faster transfer of one or more .class-files in network environments.

When the proxy's source code has been compressed and bundled into a jar-file, it can be digitally signed with use of one of Java's standard program, called jarsigner. To be able to sign the .jar-file there has to exist a private key with a corresponding public key, called a public-privat key pair. The private key, the public key and some additional information about the issuer are stored in a keystore in Java. A keystore is a standard way to store and handle keys in Java. The keystore are stored as a file at the local computer and can be managed from Java programs. Java also provides a way of exporting X.509 Certificates to a file. The X.509 Certificate consists of the public key and some information about the owner. This certificate-file can be transferred between computers. The service provider could transfer the certificate-file to the client, so that the client could verify that a signed service is from a trusted service provider. This is a possible way of making sure the authentication is valid. This is not though a convincing way to create the trust relationship. The service provider will have to hand over the certificate-file manually, probably at a diskette, when he meet the service user eye to eye. This must be done to protect the certificate so that nobody can exchange or manipulate the certificate. It is not a likely scenario to have to meet the service provider every time a new service shall be used in an Ad Hoc network. One solution of the problem is to include the certificate in the service, but this requires a way of making sure that the included certificate is a valid certificate. This problem is solved in the next section (see Section 8.2.4).

The digital signature of the proxy is created by the jarsigner and added to the .jar-file. The signature is stored in a specific file called a manifest.mf-file which is included in the .jar-file. The manifest.mf-file contains the .class-files' filename along with a message digest calculated over each .class-file. The message digest algorithm used to create the signature in the file-service case is the SHA-1 algorithm. A typical manifest.mf-entry looks like this:

```
Name: fileServer/server/FileProxy.class
SHA1-Digest: pzgHGYdRZYTQLJEOa/Pw+ZvFrzQ=
```

The content of the manifest.mf-file is repeated in a manifest.sf-file and stored in the .jar-file. There is also an added signature of the whole .jar-file in the manifest.sf-file. The additional lines looks like this:

```
Signature-Version: 1.0
SHA1-Digest-Manifest: LWRErvQe3kBbhXTx8EO+FtoA3qU=
Created-By: 1.3beta (Sun Microsystems Inc.)
```

When jarsigner signs the .jar-file it also includes the signer's Certificate. The Certificate is stored in the .jar-file as a certificate.dsa-file. By doing this, the .jar-file receiver can extract the certificate from the .jar-file.

When a user receives the .jar-file, it can easily check who has signed the code. This is done by the standard Java class-loader. When the .jar-file is received, the class-loader checks the all the signatures in the .jar-file's manifest.mf-file and in the manifest.sf-file. If all the signatures are correct, the .jar-file will get an assigned protection domain. This protection domain has a belonging certificate. This gives the .jar-file receiver a possibility to check which certificate that belongs to the downloaded .jar-file.

This section has described how the proxy is bundled in a .jar-file and how it is digitally signed. This represents the step 0 in Figure 8.1. The next step for the service provider is to register the service at the Lookup Server, step 1 in Figure 8.1. This is done with the standard Jini function .register( ) at the Lookup Server, which is found with use of Jini's standard LookupDiscovery class. The signed .jar-file must also be placed at an HTTP-server so that the client can download the service.

8.2.4 Fingerprint (Used in step 5 and 6)

To be able to make sure that a certificate really is from the person it claims to be, some additional technique has to be used. One possible way to solve the validity problem of a certificate is to use a Certificate Authority (CA). The Certificate Authority is an authorised organisation that stores certificates and by signing the certificates with their own signature, can guarantee that a certificate belongs to whom it claims to. This is unfortunately not a usable way to check the certificate's validity in an Ad Hoc network. There might not exist any way to connect to the CA. Therefore some other alternative must be used.

There are two ways of creating certificates in Java. Either the certificate can be signed with help of a CA, or it can be self-signed. In this application self signed certificates are used. This method gives the receiver no way to check the validity of the certificate, at least not without any additional technology.

The check of a validity of a certificate can however be done by calculating a unique checksum over the certificate. The service-provider creates the checksum by calculating a message digest over the certificate. The message digest can only be calculated over byte arrays. Therefore, the certificate has to be translated to a byte-array. This is done with use of

one of Java's standard function, java.security.cert.certifcate.getEncoded, which creates a byte-array of an existing certificate. The checksum is then created with use of any message digest algorithm. In the Java standard library, there exist some different algorithms. In this application, the SHA-1 message digest algorithm is used. It creates a 160 bit unique checksum of any byte-array. This means that there exists 160^2 combinations of checksums, while only one combination correctly corresponds to the input certificate.

Now this checksum must be represented in a fancy way and transferred to the service user, the client. To minimise the service user's interaction and to ease the use of the service, a simple mechanism for transfer and entering the checksum was constructed. First, the length of the checksum was shortened to be the first 126 bits of the SHA-1 message digest. To avoid that some of the bits got wrong when transferred, a checksum was added to the first 126 bits. The checksum is a 16-bit CRC-checksum, which tells if some of the first 126 bits are wrong. This gives in total a checksum of 142 bits. These 142 bits are finally represented as characters with a 6 bits character-encoding. This character combination, which represents the correctness of the certificate, is called a fingerprint. The fingerprint is represented as 24 characters. See figure 8.2.

All that is needed when checking an unknown certificates validity is to calculate the fingerprint and check if it matches that person's real fingerprint. This is done when the client receives the proxy and takes out the certificate (how this is done is described in detail later in Section 8.3). It is important to check the validity of the code before it is instantiated. If it is done in this order, even hostile code is harmless. It never gets any chance to run. The fingerprint is also checked when the server authenticates a client, which are trying to connect, see section 8.2.5.

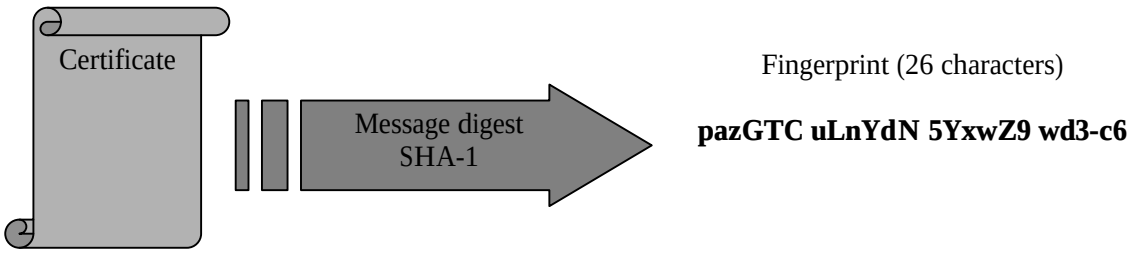

Figure 8.2 *The Creation of a fingerprint.*

To authenticate each other, the server and the client need to have the correct fingerprint. This method requires that the fingerprint have been exchanged between the service user and the service provider before the service can be used. The client has to have the service providers fingerprint and the service provider has to have the clients fingerprint. To be able to easy share the fingerprint, it was printed at the backside of the service provider's business card. The service user now only needs to have the service provider's business card to be able

to make sure that the service is from a trusted person. Fingerprints are read into the server and the client with help of a C-Pen, which transfers the fingerprint from the business card to the application by only moving the C-Pen over the fingerprint printed at the business card.

A C-Pen is a Personal Digital Assistant that reads, stores, processes and transfers printed text cordlessly to your PC.

8.2.5 Step 6: Authenticated Diffie-Hellman key exchange algorithm

If it is the first time the service is used, the master key K (see Section 8.3.3) used to create session keys for encryption and decryption, must be fetched from the service provider. This is done by the proxy. To be able to easily use the key-exchange mechanism in other applications, e.g. services, and to be able to easily exchange it, the key-exchange algorithm was created as its own proxy, see Figure 8.3. This also allows the key exchange proxy to be used as a separate Jini service. It would need some small additions to be shared as a Jini service though. In the file-server service's proxy, the key proxy is included as a part of the downloadable proxy code.

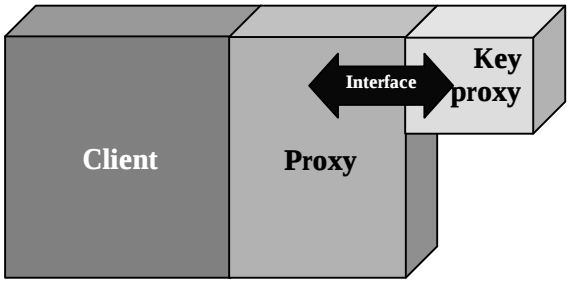

Figure 8.3 *The service proxy talking to the key proxy through an interface.*

To be able to connect to the server to get the master key K, the server requires that the client authenticate it by signing a piece of data, and to send its certificate to the service. The data must be provided from the proxy to prevent that a hostile client has stored some other, already trusted client's signature, of the data. The server can then check that it trusts the client by first check that the signed data really is signed by who it claims to be, e.g. to check that the signature matches the certificate. Then the server verifies the certificate by checking if its corresponding fingerprint is a valid and trusted fingerprint. The fingerprint must have been added to the server in advance. Otherwise, the server will be prompted to add the correct fingerprint before the client can use the service. If the fingerprint is correct, the client is trusted and the master key K can be fetched. If it is not, the client will simply be disconnected and no further communication will be permitted.

When the key proxy has authenticated the client, it will be able to fetch the master key. This key is fetched from a key server running at the service provider's machine. The key is securely transferred with help of a key exchange protocol. In this application, the standardised Diffie-Hellman key exchange protocol is used. The Diffie-Hellman protocol is constructed so that no listener can calculate, or even guess, the shared secret after a completed transaction.

To get the master key K, the authenticated Diffie-Hellman key exchange algorithm is used, see step 6 in Figure 8.1. The Diffie-Hellman key exchange protocol was created to be able to share secret keys without having a pre-shared secret. To use the Diffie-Hellman protocol in an Ad Hoc environment, some kind of authentication has to be added. The earlier described authentication model, to check the certificates fingerprint, is used in this application. The client authenticates the server as described earlier, when the proxy is downloaded. The server has though to add authentication of the client before the key can be exchanged. The key exchange algorithm consists of only two messages, one sent from the proxy to the server, one sent form the server to the proxy. After the messages have been sent, the master key K has been securely transferred with help of the shared secret. The two messages are shown in figure 8.4.

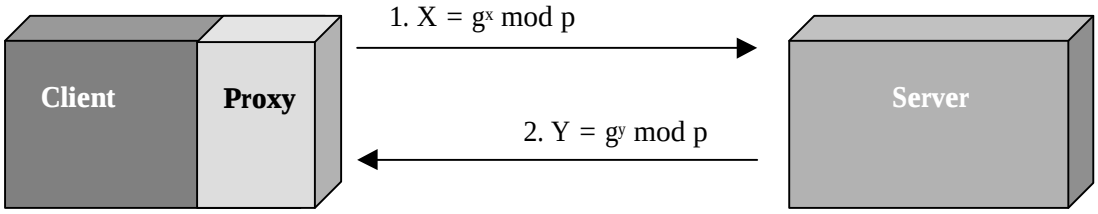

Figure 8.4 *The Diffie-Hellman key exchange algorithm.*

Before the two messages can be sent, both parties has to agreed over two numbers, g and p. The agreement is really simple in Jini services, because it is with the service's own proxy that the service has to agree. Because the proxy is downloaded from the server, the two numbers g and p can be included in the proxy from the beginning. The two numbers g and p are non-secret numbers:

- g is a number specified by the Diffie-Hellman algorithm
- p is a prime number of 768 bits defined in the Diffie-Hellman algorithm

The X is calculated at the client-side by first chose a secret number x and then use the formula $X = g^x \bmod p$. The server-side should also chose a secret number y and calculate the Y with the formula $Y = g^{xy} \bmod p$. But because the Y isn't a secret number, it can be included in the proxy code from the beginning. The y must though be secretly stored at the server

machine, so that it can be fetched when needed, but invisible for other users. Because these simplifications, only the X needs to be send from the proxy. The server does not have to send the Y. When the proxy sends the X it must be signed by the client so that the server has a way to authenticate the service user. The client's certificate has also to be sent to the server. It must be sent because the server does not have the client's certificate if it is the first time the client uses the service. When the server receives the signature, it is checked that it matches the included certificate. This is done by the Java function java.security.cert.Certificate.verify(). If the signature is correct, the server checks if the fingerprint of the certificate exists in the server as a trusted fingerprint. If it does not, the server will be prompted to add the correct fingerprint for the certificate with use of the C-Pen as explained in section 8.2.4. If the signature is incorrect or if the fingerprint does not exist as a trusted fingerprint and it is not entered correctly when prompted, the server will disconnect the client without transferring the secret master key K.

If the authentication passes, the next step is to create the shared secret so that the master key K can be securely transferred. The shared secret is calculated as $g^{xy} \bmod p$, which is easily calculated as $Y^x \bmod p = g^{xy} \bmod p$ in the proxy and as $X^y = g^{xy} \bmod p$ at the server. Now both sides have the same shared secret.

The secret master key K is finally encrypted by XOR:ing the shared secret with the master key K. The result is transferred to the proxy where it is decrypted. The decryption is done by XOR:ing the shared secret with the received encrypted key. The master key K is stored in the client together with an identifier of the service-provider. If the master key K already exists for the service the client has chosen to use, this phase (step 6) does not need to be accomplished. This master key K is later used in the proxy and in the server to create temporary session keys used for en- and decryption of packets and to create and check signatures at sent and received packets.

8.2.6 Step 7: The Fileserver's authentication of the client

A Jini-service does not prevent anyone from getting and using the service's proxy. Therefore, the service-provider has to have a way to authenticate the service user, the client, so that no unauthorised user can get any information out of the service. In the file-service case, all unauthorised users are denied access if they do not pass the authentication phase. This means that every client who wants to use the service must authenticate it to the server to be able to use the service. The server in this section is the program running at the service-provider's machine which is serving proxies who wants to connect and use the service's methods, see figure 8.5.

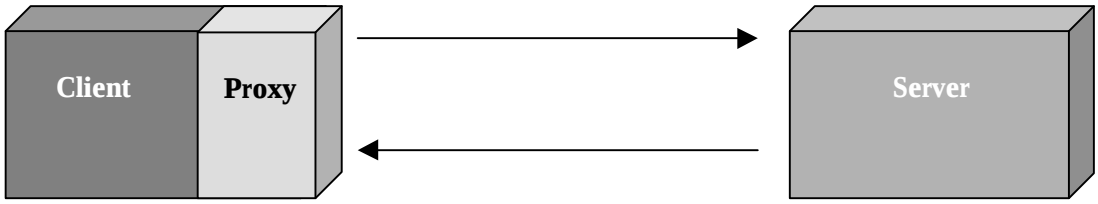

Figure 8.5 *The client communicates with the server through the service-proxy.*

When a client tries to connect to the server through the proxy, the server starts the secureConnection-phase. This phase consists of sending and receiving messages, as will be described below. If the client has the service specific master key K, it will be able to successfully connect to the server. If not, the client will be rejected and unable to use the service. Now follows the messages, which are sent between the client and the server when the client wants to connect to the server with the proper master key K: [11]

(The client, or actually the proxy, is called A and the server is called B in this scenario.)

(a) A selects a random number rA and sends it to B along with A's hostname.

(b) B selects a random number rB and sends a packet with A's hostname to A a packet with A's hostname, B's hostname, the random numbers rA and rB and an HMAC calculated over A's hostname, B's hostname and the random numbers rA and rB. The master key K is used as the input key in the HMAC-calculation.

(c) A receives rB and the HMAC over rA, rB, A and B. A checks that A, B, rA and the HMAC are correct.

(d) A sends A's hostname, the random number rB and a HMAC over A's hostname and rB to B. The master key K is used as the input key in the HMAC-calculation.

(e) B receives the packet from A. B checks that A, rB and that the HMAC over A and rB are correct.

(f) A and B can now create their session keys:
Whmac = HMAC (rA+rB, K)
WblowfishKey = HMAC(Whmac, K)

If (a) - (f) are successfully completed, a secure tunnel between the proxy and the server is created. The secure tunnel is created by encrypting all data sent between the proxy and the

server. The encryption and decryption at the client side is made by the proxy. This is a really neat thing, because the client that uses the service doesn't have to care the least about how the communication is encrypted. The other neat thing with having the proxy providing the encryption algorithm is that the algorithm easily can be exchanged to another algorithms, without the client's knowledge.

To be able to connect the proxy, the server actually only care about that the service-user has the right master key K. This is because the master key only can be fetched if the authentication phase, described earlier in section 8.2.5, passes.

The master key is used in the service to create the session keys used for encryption and decryption. The master key K is also used to create a unique checksum, called a HMAC, at each sent packet. The session keys are calculated in step (f) as:

- Session key for HMAC calculations : Whmac = HMAC (rA+rB, master key K)
- Session key for encryption and decryption : WblowfishKey = HMAC (Session key for HMAC calculations : Whmac, master key K)

The numbers rA and rB in the creation of the HMAC session key is the random numbers exchanged in step (a) and (b) above. The input data when creating the encryption and decryption session key is the session key Whmac used for HMAC calculations.

The Whmac key is used to calculate a HMAC for all transferred packets between the proxy and the server. The HMAC is calculated over the data that shall be transmitted over the network before it has been encrypted. When the data is received, the HMAC is checked and if it is incorrect, the packet is silently discarded. All data, which is sent over the network, is encryption with use of the session key. The algorithm used for encryption is included in the proxy. This enables the service to change the cryptographic algorithm without having to concern about incompatibility with the client. The client does not needs to have any cryptographic algorithm implemented, because it is always included in the proxy-code. When a packet is received, it is decrypted using the session key WblowfishKey in the cryptographic algorithm. In this service, the Blowfish cryptographic algorithm is used. It is combined with a CBC (Code Block Chipper) algorithm to enable large amounts of data to be encrypted. The CBC algorithm is also included in the proxy-code.

When the secured tunnel between the proxy and the service has been set up, the client can securely use the service by invoking methods at the service interface. The time to find the service and to create the secured tunnel is, luckily, short. The total time form the client's lookup until the service is ready to use is only a parts of a second, about $0{,}5 - 1$ s depending on the speed of the computers and the speed and load in the network.

8.3 THE CLIENT IMPLEMENTATION AND THE KEY AND FINGERPRINT MANAGEMENT

8.3.1 Introduction

The client's implementation of our security-model involves step 5 and 6 in figure 8.1. All the necessary keys, used by client and server, must somehow be managed and that is, as well step the 5 and 6 in our security-model, discussed below.

8.3.2 Step 5: Authentication

The service that the client has requested is remote code and is a security risk. Therefore, the client must make certain, that no one has altered the code during transmission or that the code does not come from a hostile server.

The way the client does that is illustrated in the figure 8.6.

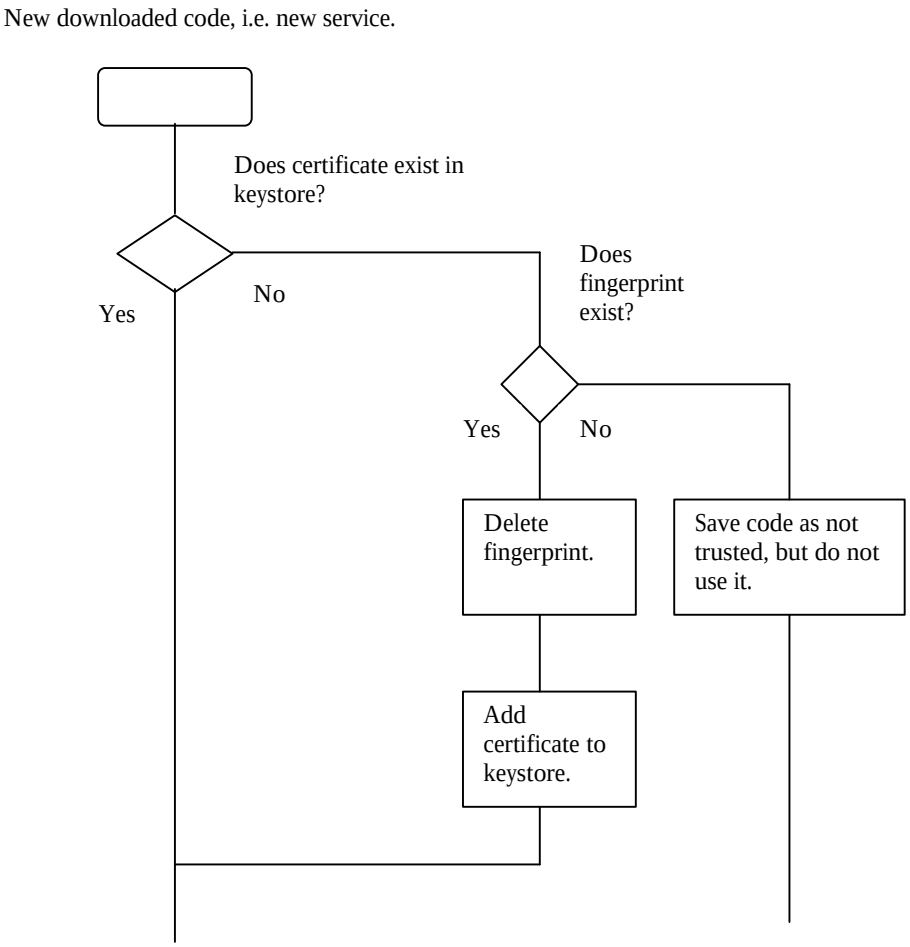

Figure 8.6 *Flowchart of authentication model.*

When the client receives the implementation from the server, it is delivered as a signed compressed jar-file. The jar-file beside the implementation classes also contains a certificate and a signature file. The client now retrieves the certificate and by doing that, he makes certain that the code was not altered or the sender was faked.

If the service has been used earlier, then the certificate already exists in the keystore and the downloaded implementation is immediately marked as trusted. If not, the user might already have trusted the server by accepting the fingerprint (see Section 8.2.4) generated from the server's certificate. In that case, the code can also be marked as trusted. When the matching fingerprint exists, it is then removed from the fingerprint list and the certificate is saved in the keystore.

If none of the criteria above is met, the remote code is marked as not trusted. If the user wishes to use this code, he must enter a valid fingerprint for the code. Otherwise, the code will be stored in the temporary directory until someone removes it from the hard disk.

8.3.3 Step 6: The Master Key

Before the client can use the service, he has to supply the proxy with the master key. The master key is necessary for the proxy to be able to generate HMAC and Blowfish keys. It is also necessary that the client handle the distribution of the master key, because the generation also demands that the request is signed. To be able to sign code the private key is needed and that access should not be granted to remote code. Therefore is the master key distribution rather lengthy.

The process of supplying the proxy with the master key is illustrated in the figure 8.7.

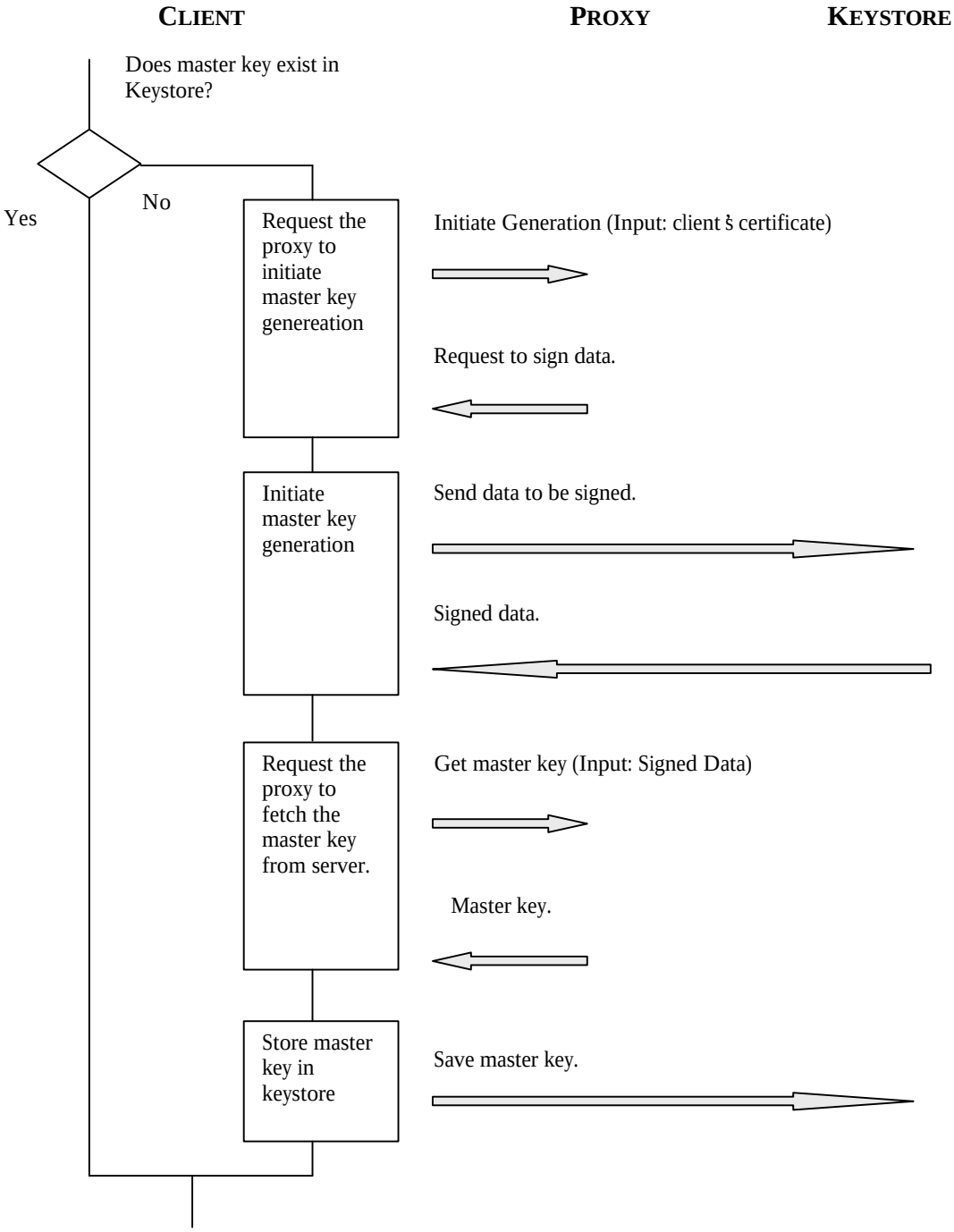

Figure 8.7 *Flowchart of master key agreement.*

If service has been used earlier the wanted master key is stored in the keystore and can be used directly. When the master key is not stored in the keystore it must be generated. The master key is generated with a simplified Diffie-Hellman key-agreement algorithm.

The client starts the key-algorithm with calling the proxy, but before the proxy can send this request to the server, it needs to sign his request. Therefore, the proxy replies to the client that it needs data to be signed. The client forwards this request to the KeyStoreProvider class that handles the signature process. Our KeyStoreProvider class will be discussed in the next section.

When the KeyStoreProvider has signed the data, the client is ready to call the proxy to fetch the master key. After the proxy has received the master key from the server, the client calls the KeyStoreProvider again to store the master key locally.

NOTE: The word *keystore* used above is not quite right, because we had problems to store an arbitrary key in the standard keystore (see Section 8.3.4), which comes with SDK 1.2. The master key is here stored in a separate file, but the difference is insignificant. In order not to confuse the reader and to keep syntax equal, the word *keystore* used.

8.3.4 The Key and Fingerprint Management

The KeyStoreProvider is a general manager for the standard keystore that comes with Java 2 SDK. The KeyStoreProvider is used both by the client and by the server.

The KeyStoreProvider class has three purposes:

- manage the .keystore-file
- manage the .master key-file
- manage the .fingerprint-file

The simplest way to create a keystore-file in Java 2 SDK is to use the keytool program [32]. The keytool-program generates a private and public key-pair.

The private key can be retrieved with the java.security.KeyStore class and the private key itself is retrieved with the interface java.security.Key [33]. The settings we have used are:

- the key algorithm: DSA
- the format of the encoded key: PKCS#8

There are no options to chose when using keytool, which can be seen as quite bad. To verify which algorithm and format that was used, it is necessary to write a separate program, that retrieves the private key and prints out the setting that are in use

We had rather big problems to store an arbitrary key in the keystore, so we had to store the master key encrypted in a text-file. The reason for this is that, when loading the keystore a provider is needed as input value and here we are set aside to use the standard provider *jks.*

The standard provider that comes with the European version of Java 2 SDK only supports a certain sets of key-implementation and not an arbitrary key [34]. It does not matter if the key is in a proper *EncryptedPrivateKeyInfo* format, as is defined in the PKCS #8 [35].

This limitation of the number of key algorithm, is of course a big disadvantage when building a more flexible security model, because it is desirable to only use the standard Java implementation, when building a new own security model. One can imagine what happens when specific system calls are used in an application and this application is then exported to an another system. It will not work and this is in direct conflict with Java basic principal. Java application should run on any system, e.g. Win95, Solaris, and MacOS.

The damage could easily be repaired by building a own keystore and encrypt all key manually to achieve a safe key storage. We did not encrypt the master key or encrypt the file itself, when storing file, but in our implementation is it easy to expand our program with a separate module that takes care of the master-key management. Key management is something that is well known and developed, and not hard to achieve.

The fingerprint is also stored as cleartext, but it is implemented so that a standalone module that handles the storage in a secure fashion easily can be imported.

8.4 ALTERNATIVE WAYS TO IMPLEMENT OUR SECURITY-MODEL

Along the road to the final implementation we did try different ways and beneath follows a discussion of alternative ways that we discarded.

8.4.1 Dynamically update the Policy File

One approach that we tried was to dynamically update the policy file. This solution enables more security levels for different user and this is normal situation in a multi-user environment.

In the API for the java.security.Policy [36] is it clearly explained that this should be possible. But during development we understood, that this was only the case when the code was not already loaded by the classloader. Once the class had been loaded by the classloader, it is not possible to change its permission.

That is a drawback, because the scenario of changing the policy during runtime is quite useful.

8.4.2 Explicitly Verify Signature

For a long time, we had a big problem with a security hole that enabled downloaded code to connect back to its originating server (see Section 8.4.3). Due to this fact, we did try to explicitly verify the downloaded code, i.e. generate a new signature based on the downloaded code and the enclosing certificate, and then compare the newly generated signature with the downloaded signature. We were able to retrieve the certificate, but the code was not retrievable.

The reason that the code was not available is that the remote code is reflected. Reflecting code means that the class is temporarily loaded to the client's classpath or the classes' methods are made accessible to the client.

A misgiving that we had was that reflecting code should be given the same codesource as the code have, that requested the reflection, i.e. the client's codesource and thereby receive the same rights as the client code. This is a total misunderstanding, because code is given its codesource when loaded and in this case it is the server's codebase. The Java approach with reflecting code made it impossible to verify the downloaded code manually. This problem was later solved when we realised that by retrieving a proper certificate, we also made certain that the code was not altered during transmission.

8.4.3 Security Flaw

One problem that puzzled us for a long time was that downloaded code had the right to connect back to its server, without permission being given in the client's policy file. In the beginning, we thought that we had foreseen something, but this is probably an inheritance from the old Java security model, where this was allowed. The idea is that a firewall should protect the local Ethernet from outside attacks which is in contrary to what the Java Security Guide says.

All code is running under security restrains which is also quite logical, because downloaded code should not be able to connect back to its server and download miscellaneous code to the client.

This is of course a security flaw, because a Java programmer should not have to be concerned about this problem.

8.5 SUMMARY

8.5.1 Constructions demands

The demand when implementing the security solution was that the standard Jini code should not be changed, if possible. There should not be any need of extra Java libraries, such

as JCE (Java Cryptographic Environment) at the server or client. Therefor only standard Java code could be used.

8.5.2 Easy to use

The service should be easy to use for the clients and the server should only need minimal administration. Therefore the fingerprint was constructed and used at both the client and the server side. To ease the reading of the fingerprint, a C-Pen was used to read it into the computer. This made the application easy to use.

8.5.3 As little overhead as possible

To minimise the traffic and the overhead at each packet, only minimal extra information was added. See Appendix B.

8.5.4 Speed considerations

The encryption and decryption of sent data should mean as little extra delay as possible. Therefore, the Blowfish algorithm was used. It is one of the fastest algorithms and it is secure enough. In fact, a file transfer never took more than twice the time it should have taken if no encryption and decryption should have been done.

9 CONCLUSION

This work shows that it is possible to construct secure services in Ad Hoc environments, at a low-cost. The development cost has been low, because this solution is built completely on the Java standard library and Jini 1.0.

One problem with security implementation is that both parts must first agree on a common method, before the real work can be done. Therefore, it is not addressing to let the user handle the security issues around communication. In our application the encryption-algorithm is included in the service, without the client even notices that. This makes changes and updates very easy for the administrator. All that is needed is to update the proxy.

In our application is first, an asymmetric-key generated, which is later used to create the symmetric session-keys. The reason for this approach is that the asymmetric key-algorithm is slow but strong and easily distributed, while the symmetric key-algorithm is faster but not as strong and more complex to distribute. This is a good combination of the two different cryptographic-algorithms, which provides both a fast and strong security with an easy key-distribution.

All security models are depending of an initial trust that is "completely" secure. The management of the initial trust is always hard to make invisible and will more or less affect the user-friendliness of the program. To solve this problem as smoothly as possible, did we try to integrate this trust, into something that we do daily. It was from this approach the business card born. When we demonstrated the application at Ericsson Research in Kista, we let a fingerprint be printed of the back of the business card. When someone gave his card to someone else, this symbolised that he trusted the receiver. The fingerprint can even be used to find a specific service provider.

The summary of this application is that it is possible to create secure services at a low cost and still be able to uphold a high-level security.

10 FUTURE WORK

In our work we have been able to present a complete secure Jini-application. The application can serve as a base to other arbitrary services, e.g. printer service.

Implementing cryptographic-algorithms is difficult and even more harder is to make them efficient. We have made some effort to make the algorithm faster, with help of parallel running threads, but optimising the algorithms also involves a mathematical side and that is not within the scope of this thesis.

Using Java is a very nice way of building Object-Orientated programs, but Java has one disadvantage, it calculate floating-point slowly. That was noticeable when we used the java.security.SecureRandom class to generate randomised number, but on the other side is the generation of "randomised" numbers also very hard and complex.

During the work to speed up the generation of randomised number, Mats Näslund (employee at Ericsson Research, Communication Security Lab) did propose that the audio port should be used to record background noise. The idea was that this background noise should be similar white noise that is stochastic to its nature. We did not have time to fulfil the work or verify that the recorded noise was stochastic, but this is also not within the scope of this thesis.

APPENDIX A: THE FILE INTERFACE

```
/*
 * FileInterface.java 99/12/09
 */

package fileServer.common;

public interface FileInterface extends java.io.Serializable {
    public byte[] read(String path) throws java.rmi.RemoteException;
    public boolean write(String path, byte[] file) throws java.rmi.RemoteException;
    public boolean delete(String path) throws java.rmi.RemoteException;
    public boolean rename(String oldPath, String newPath) throws java.rmi.RemoteException;

    public String[] list() throws java.rmi.RemoteException;
    public String getLastModified(String path) throws java.rmi.RemoteException;
    public int getSize(String path) throws java.rmi.RemoteException;

    public byte[] initGetMasterKey(java.security.cert.Certificate clientCertificate) throws
        java.rmi.RemoteException;
    public byte[] getMasterKey(byte[] signedData) throws java.rmi.RemoteException;
    public void putMasterKey(byte[] masterKey) throws java.rmi.RemoteException;
}
```

APPENDIX B: THE FILE PACKET

This is the packet sent between the file-proxy and the file-server. Every packet is encrypted before it is sent (the first five fields). There is also a HMAC added to each packet:

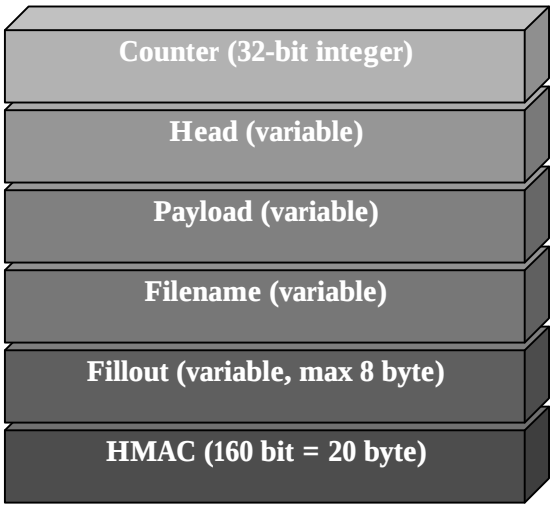

Figure B.1 *The file-packet.*

The different fields in the packet:

- Counter, an integer increased every time a packet is sent. The proxy will always send packets with even counter numbers while the server always sends packets with odd numbers. If the counter is not correct the packet will be discarded. In other words, if the counter is less than or equal than the last received packet or it is larger then the last correctly received packet plus one. This prevents a listener from storing a packet and sending it at later time. This adds extra security to the communication, because a hostile listener can not send packets with its own requests. They would simply be silently be discarded. This is though not a requirement for the security, but it adds extra security to the communication.

- Head, a string with the request-command or the answer-type in the packet.

- Payload, the content of the request or answer.

- Filename, the name of the file requested or the name of the file in the answer.

- Fillout, the extra bytes of the encryption. This is usually called padding. When the first part (the fields above) of the packet is encrypted, it is usually larger than the original size. These extra bytes are stored in the fillout field so that the packet can be correctly decrypted when received. The extra bytes exist because the encryption algorithm (Blowfish) only works at whole 8 byte blocks. Therefor, before the data can be encrypted, some extra bytes (nonsense bytes) must be added to make a multiple of 8 bytes. Those extra nonsense bytes are thrown away when the packet's content is decrypted at the receiver. In the file server application the nonsense bytes are the numbers from 0 to 8.

- HMAC, this field is calculated over the encrypted packet and added to every sent packet. When the receiver receives the packet, it calculates the HMAC and checks its validity. If the HMAC is not correct, the packet is silently discarded. This makes sure that the packet really is from the correct sender.

APPENDIX C: CLASSDIAGRAM

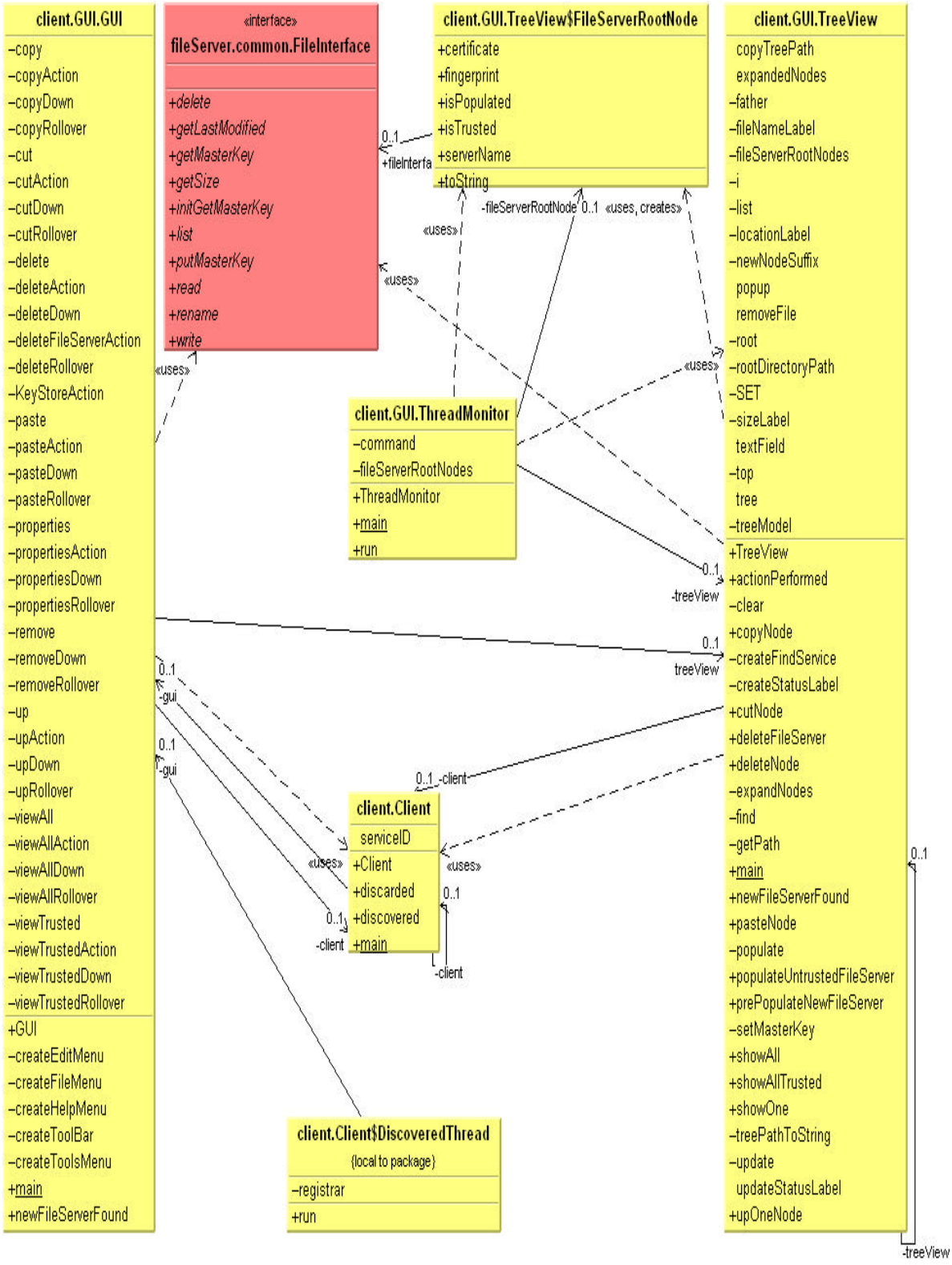

Figure C.1 *Client's classdiagram*

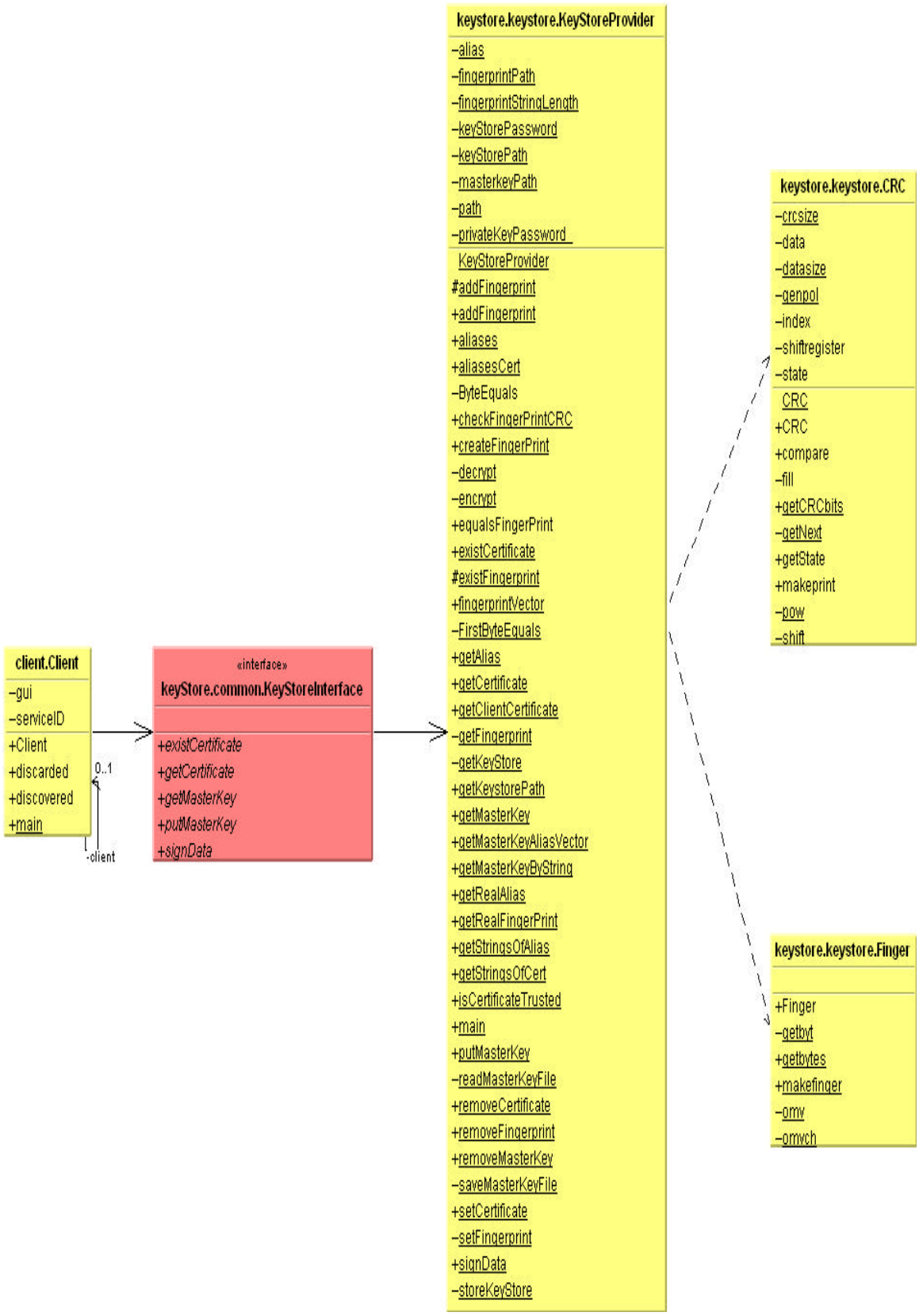

Figure C.2 *KeyStoreProvider's classdiagram*

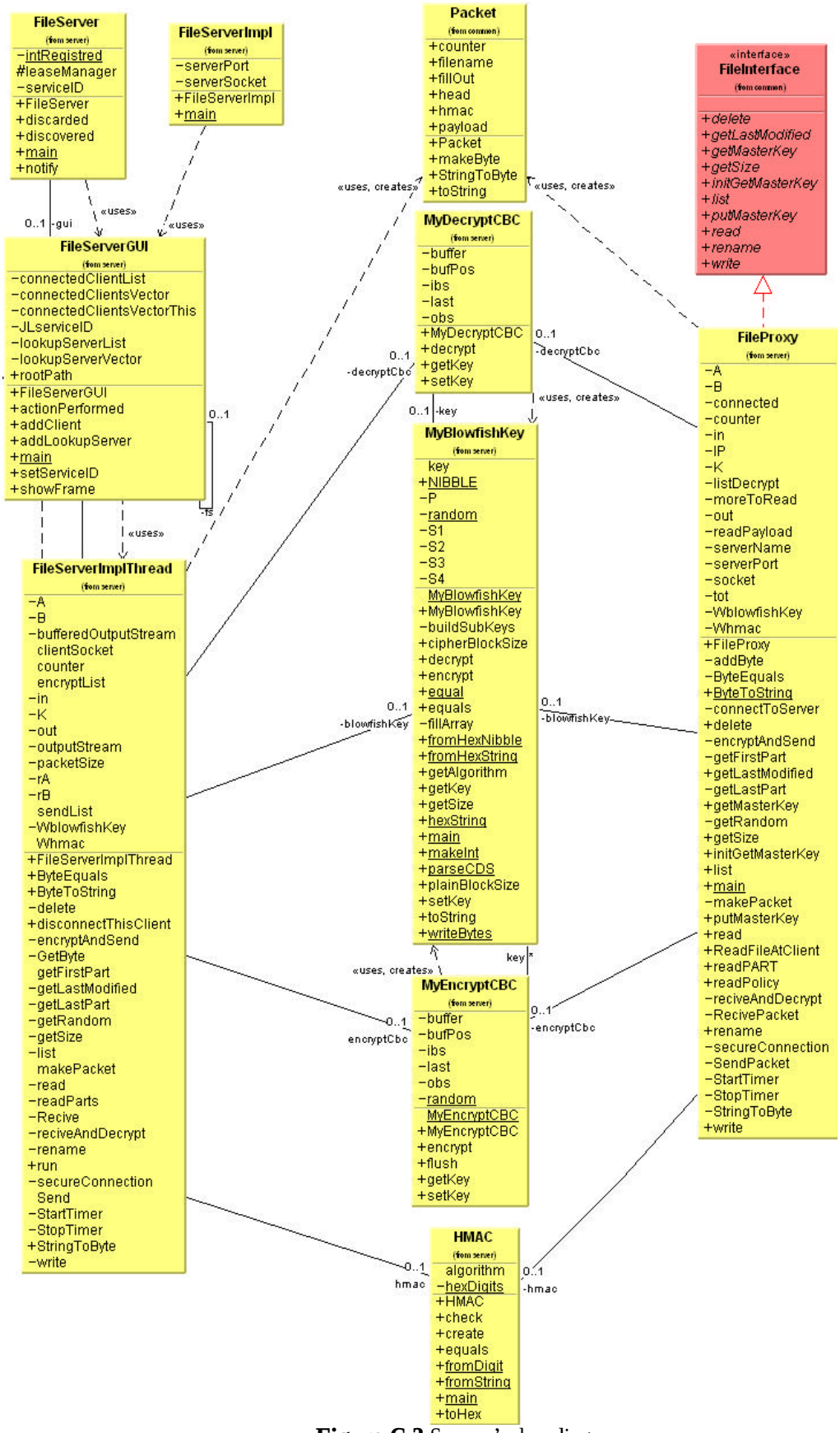

Figure C.3 Server's classdiagram

APPENDIX D: THE CLIENT'S GUI

The client's GUI looks and works very like the Explorer in Windows. The client's GUI is shown in Figure D.1.

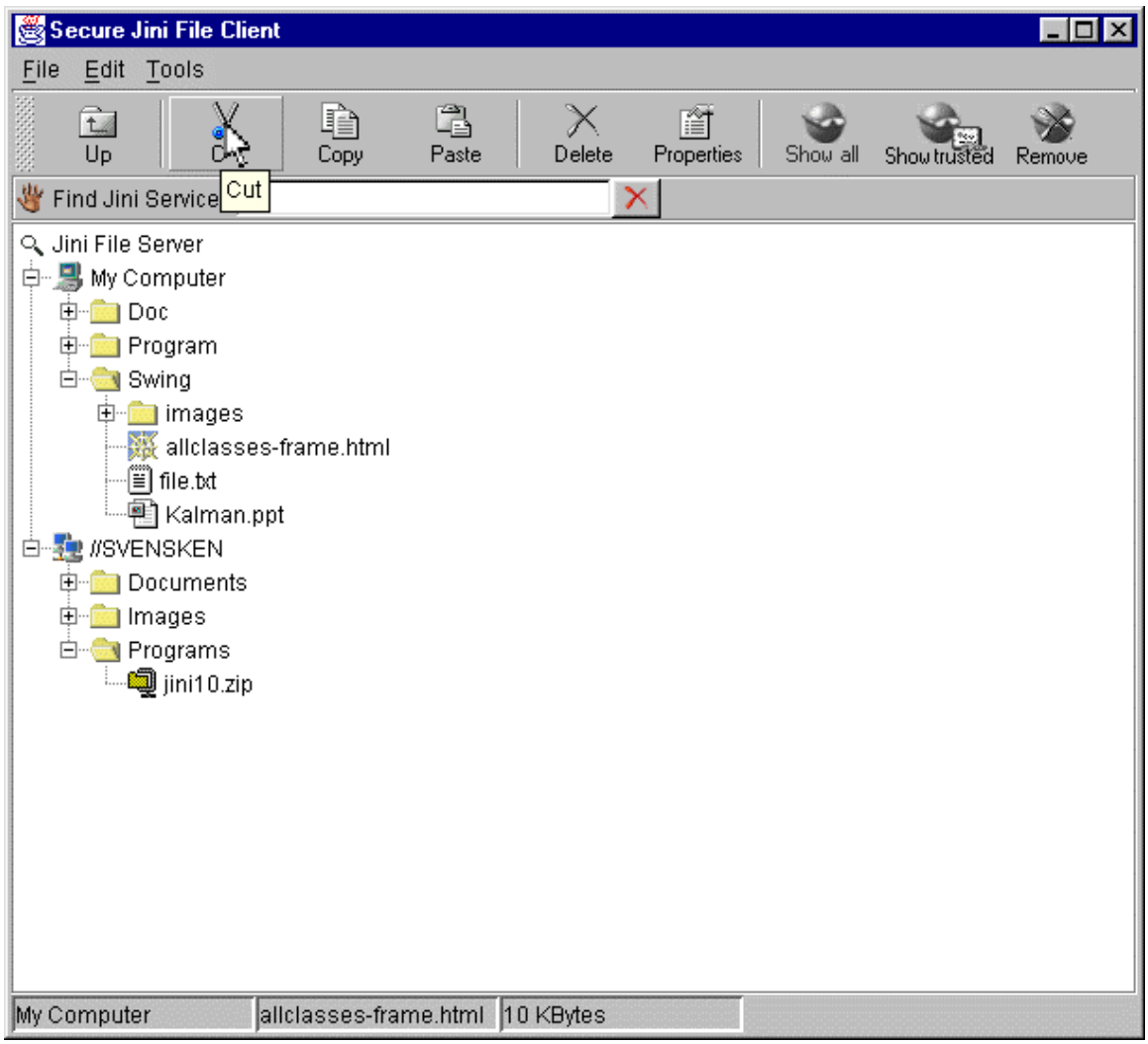

Figure D.1 *A snapshot of the client's GUI*

In the figure above is two rootdirectories shown - the local, "My Computer", and the remote, Svensken. The remote rootdirectory is named after the remote machine name.

Beside all the normal function as cut, copy, paste, etc. are also the Jini function available from the GUI. They are represented with the buttons: Show all, Show trusted and remove.

- The Show All button displays all the available file-servers in the network.

- The Show Trusted button displays all trusted file-servers in the network.
- The Remove button removes one displayed file-server.

The client has also a GUI for its keystore. The GUI is in figure D.1 shown.

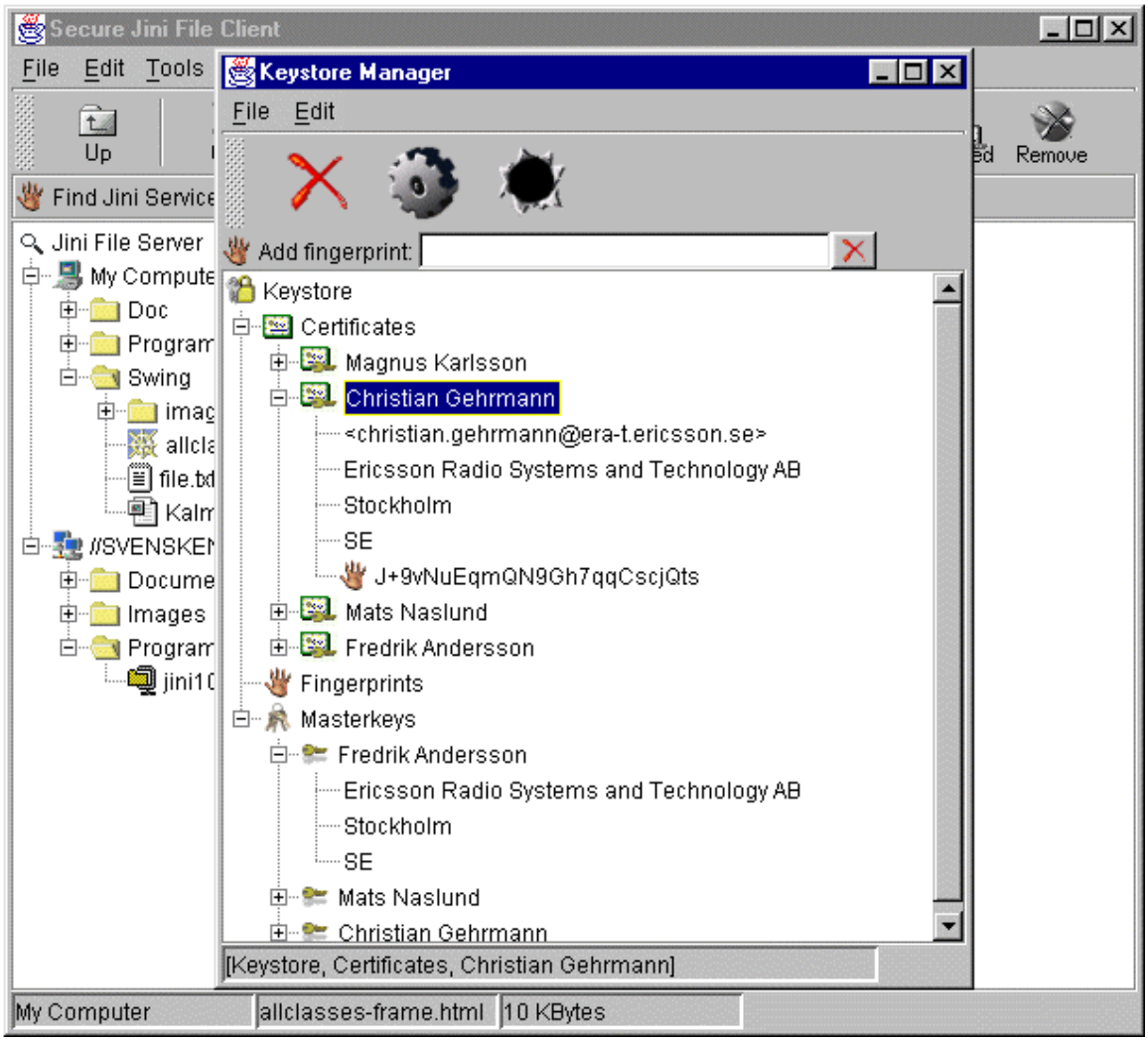

Figure D.2 *The client's Keystore Manager.*

In this GUI is all certificates, fingerprints and master keys shown. The certificate is from persons that trust the client and possible matching fingerprint.

The primarily methods in keystore manager is to add fingerprint and delete certificate, fingerprint and master keys.

APPENDIX E: INSTALLING AND RUNNING THE SERVICE

Install Java 2 SDK version 1.2. Win98 user recommends installing Java 2 SDK version 1.3.

E.1 PREPARATION

Open jini.zip.

Figure E.1 *Directory hierarchy.*

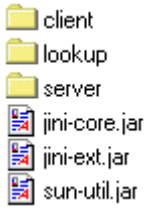

Figure E.2 *Files in jini directory.*

If the root directory looks otherwise the file is corrupt.

E.1.1 Client Preparation

Paste \jini\client\- to C:\jini\client machine and the three jar-files, which are shown in figure C.2.

E.1.2 Lookup Server Preparation

Paste \jini\lookup\- to C:\jini\lookup\ on the lookup machine.

Open C:\jini\lookup\lookup.bat. On line 28 must the machine name be changed to the present Lookup Server machine name.

E.1.3 Server Preparation

Paste the \jini\server\- to C:\jini\lookup\ on the server machine and the three jar-files, which are shown in figure C.2.

On line 16 in C:\jini\server\fileServer.bat must the machine name be changed to the present server machine name.

E.2 STARTING THE APPLICATION

1. Start C.\jini\lookup\lookup.bat on the lookup machine. On a slow machine (Pentium 166 MHz, 32 Mbytes) must user wait up to 4-5 s, before he can press any key to continue. Other user presses directly any key. Waits until the bat-file has self terminated.
2. Start C:\jini\server\fileServer.bat on the server machine.
3. Start C:\jini\client\client.bat on the client machine.

E.3 USEFUL BAT-FILE

The C:\jini\server\fileServerSign compress needed server files and signs these. The keystore is located in C:\jini\server\keystore.

E.4 USEFUL LINKS

Java 2 SDK security tutorial:
http://java.sun.com/docs/books/tutorial/security1.2/index.html

Java 2 SDK tool summery:
http://java.sun.com/products/jdk/1.3/docs/tooldocs/tools.html

APPENDIX F: DICTIONARY

HMAC	An calculated unique checksum over a piece of data. Described in RFC 2165 [37].
Blowfish	An cryptographic algorithm for encryption and decryption.
API	Application Programming Interface. Functions that the programmer uses when programming.
Marshalled Object	An initiated class' variables including an URL to the host where the code was downloaded from.
DHCP	Dynamic Host Configuration Protocol. Atomised host configuration such as IP-address allocation.

REFERENS:

[1] Jaap Haartsen. *Bluetooth – The universal radio interface for Ad Hoc, wireless connectivity*. Ericsson Review No. 3 1998. <http://www.ericsson.com/review/pdf/1998031.pdf>

[2] Farnsworth, Dale and Miller, Brent. 11 June 1999. *Service Discovery Protocol.* <www.bluetooth.com/link/spec/bluetooth_e.pdf>

[3] 29 November 1999. *Bluetooth Specifications version 1.0 B.* <www.bluetooth.com>

[4] Veixades et. al.. June 1999. *Service Location Protocol.* <www.ietf.org/rfc/rfc2165.txt>

[5] Microsoft Corporation. 1999. *Universal Plug and Play: Background.* <www.upnp.org/resorces/UPnPbkgnd.htm>

[6] Christensson, Bengt and Larssen, Olof. 19 December 1999. *Universal Plug and Play Connects Smart Devices.* <www.axis.com/products/documentation/UpnP.doc>

[7] Keith Edwards, W. 1999. *Core Jini.* ISBN 0-13-014469-X. Upper Saddle River: Prentice Hall

[8] Stallings, William. 1997. *Data and Computer Communications 5 edition.* ISBN 0-13-571274-2. New Jersy: Prentice-Hall, Inc.

[9] ITU. *Data Communication Networks; Open Systems Interconnection (OSI); Security, Structure and Applications X.800*. 1991. <http://www.incoma.ru/cdrom3/ccitt/1992/x/x800.ps>

[10] Miyata, Leonard. International Message Association Forum. <http://mlarchive.ima.com/firewalls/1999/0363.html>. Last Modified: 27 Jan 1999.

[11] Schneier, Bruce. 1996. *Applied Cryptography*. ISBN 0-471-11709-9, John Wiley & Sons, Inc.

[12] Kaliski, Burton S. Jr. 1993. *An Overview of the PKCS Standards.* An RSA Laboratories Technical Not. <http://ftp.rsasecurity.com/pub/pkcs/ps/overview.ps>

[13] Kaliski, Burton S. Jr. 1993. *An Overview of the PKCS Standards.* RSA Laboratories Technical Note. <ftp://ftp.rsasecurity.com/pub/pkcs/ps/overview.ps>

[14] Java API. *X.509 Certificates and Certificate Revocation Lists (CRLs)* <http://web2.java.sun.com/products//jdk/1.3/docs/guide/security/cert3.html>. Last Modified: 20 May, 1998.

[15] Freesoft. *ASN.1 Base Definitions.* <http://www.freesoft.org/CIE/RFC/1510/50.htm> Last Modified: 29 September, 1998.

[16] Dr. Cassel, Lillian. *Comments about BER* <http://renoir.vill.edu/~cassel/netbook/ber/node7.html#SECTION00013000000000000000>. Last Modified: 27 February 1996.

[17] NIST. *Secret Key Distribution.* <http://csrc.nist.gov/nistpubs/800-7/node209.html> Last Modified: 7 October 1994.

[18] Sun Microsystems, Inc. *Binary Code License Agreement.* <http://www.java.sun.com/products/jce/jce12_license.txt>. Last Modified: 29 November, 1999.

[19] Java 2 Platform Security Architecture. <http://web2.java.sun.com/products//jdk/1.3/docs/guide/security/spec/security-spec.doc1.html>. Last modified: Unknown.

[20] Java 2 Platform Security Architecture. <http://web2.java.sun.com/products//jdk/1.3/docs/guide/security/spec/security-spec.doc2.html>. Last modified: Unknown.

[21] Java 2 SDK API. <http://java.sun.com/products/jdk/1.3/docs/api/java/security/AccessController.html>. Last modified: Unknown.

[22] Java 2 SDK API. <http://java.sun.com/products/jdk/1.3/docs/api/java/security/Permission.html>. Last Modified: Unknown.

[23] Java 2 SDK Security Guide. *Permissions in the Java 2 SDK.* <http://web2.java.sun.com/products//jdk/1.3/docs/guide/security/permissions.html>. Last Modified: 30 October, 1998.

[24] Java 2 SDK API. <http://java.sun.com/products/jdk/1.3/docs/api/java/security/CodeSource.html>. Last Modified: Unknown

[25] Java 2 SDK API. <http://java.sun.com/products/jdk/1.3/docs/api/java/security/ProtectionDomain.html>. Last Modified: Unknown.

[26] Java 2 SDK API. <http://java.sun.com/products/jdk/1.3/docs/api/java/security/cert/Certificate.html>. Last Modified: Unknown.

[27] Java 2 SDK API. <http://java.sun.com/products/jdk/1.3/docs/api/java/security/SecureClassLoader.html>. Last Modified: Unknown.

[28] Java 2 SDK API. <http://java.sun.com/products/jdk/1.3/docs/api/java/lang/ClassLoader.html>. Last Modified: Unknown.

[29] Java 2 SDK API.
<http://java.sun.com/products/jdk/1.3/docs/api/java/net/URLClassLoader.html>. Last Modified: Unknown.

[30] Java 2 SDK Cryptographic Specification.
<http://web2.java.sun.com/products//jdk/1.3/docs/guide/security/CryptoSpec.html>. Last Modified: 6 December 1999.

[31] Sun Microsystems, Inc. Java 2 SDK Security Guide. *Default Policy Implementation and Policy File Syntax*.
<http://web2.java.sun.com/products//jdk/1.3/docs/guide/security/PolicyFiles.html>. Last Modified: 30 October, 1998.

[32] Java 2 SDK API.
<http://java.sun.com/docs/books/tutorial/security1.2/summary/tools.html>. Last modified: Unknown.

[33] Java 2 SDK API.
<http://java.sun.com/products/jdk/1.3/docs/api/java/security/Key.html>. Last modified: Unknown.

[34] Java 2 SDK API.
<http://java.sun.com/products/jdk/1.3/docs/api/java/security/KeyStore.html#getInstance(java.lang.String)>. Last modified: Unknown.

[35] Java 2 SDK API.
<http://java.sun.com/products/jdk/1.3/docs/api/java/security/KeyStore.html#setKeyEntry(java.lang.String, byte[], java.security.cert.Certificate[])>. Last modified: Unknown.

[36] Java 2 SDK API.
<http://java.sun.com/products/jdk/1.3/docs/api/java/security/Policy.html>. Last modified: Unknown.

[37] Krawczyk, et. al. *RFC 2165 HMAC.* <http://www.ietf.org/rfc/rfc2104.txt>. February 1997