# An Authentication and Authorization Architecture for Jini Services

Thomas Marcus Schoch
tmschoch@informatik.tu-darmstadt.de
Distributed Systems Group
Computer Science Department
Darmstadt University of Technical

Advisors:
Dr. Oliver Krone
oliver.krone@swisscom.com
Swisscom

Prof. Dr. Friedemann Mattern
mattern@inf.ethz.ch
ETH Zürich

October 2000

# Contents

# Chapter 1

# Motivation

It is likely that there will be a new paradigm in the field of computer application in the near future. The current paradigm is that workstations and servers are connected to local area networks (LANs), e.g. Ethernet. These LANs can build wide area networks (WANs) by interconnecting them, e.g. the Internet.

Using that infrastructure, the industry currently defines standards for connecting all types of appliances into the Internet. This also includes household appliances, like TVs, refrigerators, lawn sprinklers, vacuum cleaners, lights, etc. Further on, it is imaginable that also clothes and every product inside a refrigerator will be connected to the Internet via smart labels, a technology that is already available.

However, only connecting these items to the Internet does not provide any benefit. The added value of the interconnecting originates if all the appliances provide a well-defined application programming interface (API). To reach a higher degree of transparency - that means to hide proprietary protocols and for easier administration of those appliances - the appliance is represented by a service in the net. The service provides also a well-defined API and carries out the communication with the appliance. A client, which itself is a software component, can use the appliance by using that service API. So the API is an interface which can be used by different firms to communicate in a well-defined manner.

One advantage for the user is that he has a higher degree of flexibility in operating the appliance. The intrinsic value is the interaction between different services. Services can combine some other services to provide a new functionality, e.g. a storage service and a TV service can be combined to a VCR service. It could also be possible that the TV service detects a defect and automatically contacts the manufacturer service. The manufacturer service then instructs a robot in the storagehouse to get the right spare part. Schedule services on both sides negotiate a date for the installation of the spare part, and so on. This also means a reduction of the costs for the maintenance of appliances.

Such services require a distributed communication infrastructure. A communication infrastructure acts as a middleware between the operating system and the application. Its task is to provide service discovery and brokerage, similar to a marketplace, where marketer and customers can meet and trade. The protocol used in the marketplace example is very informal. A customer can express the wish to buy a TV in many different ways which is understood by the marketer, whereas a client and a service in the computer environment need a well-defined protocol of how to communicate. To use the TV service mentioned above, there is exactly one possible way to turn the TV

on or off by calling the corresponding methods.

Jini, a technology from Sun Microsystems, is one of several approaches to provide a middleware for service discovery and brokerage. A key concept of Jini is the self-healing ability that fits very good in spontaneous networks, where appliances appear and disappear without notification, so that the self-healing ability does implicitly the clean up of no longer used resources. The biggest advantage of the Jini technology compared to Universal Plug and Play is current availability. Therefore, Jini is a good candidate for the infrastructure of the new paradigm.

Clearly, security, in particular user authentication and authorization, is very important in an open service environment. Every Jini service can currently be used by anyone in the Internet. For example, the owner of the TV can use the tvOff-method of the TV service, but anyone else in the Internet, especially someone who is malicious, can use this method of the TV service too, while the owner of TV is watching TV. It would be possible to prevent this by using a firewall, but then another problem arose: trying to program the VCR service to record a movie from an Internet-cafe would be impossible. Using a firewall does not include authorization, e.g. only the owner of the TV should be able to watch TV anytime, so that authorization mechanisms are needed. His children maybe restricted to watch TV only in the afternoon.

The thesis is structured as follows:

Chapter 2 is about security concepts in general. It explains the meaning of digital signatures, secure communication and therefore also integrity, authentication and authorization which are needed by this architecture.

A more detailed description of Java and Jini, which is a very promising middleware technique to interconnect all kind of appliances, will be given in chapter 3.

Chapter 4 combines the aspects security and Java. It introduces the Java security concepts which are used in this architecture.

Since the description of the architecture which is given in chapter 6 is very detailed, a general overview is given first in chapter 5 which also includes a case study on a higher level, whereas chapter 8 provides a more detailed case study.

Chapter 7 explains how the reference implementation of this architecture is made by implementing the Java interfaces which describes the architecture.

Finally, conclusions and future work, which are related to the architecture, are given in chapter 9.

# Chapter 2

# Security Concepts

This chapter gives a general overview about security concepts that are used throughout this thesis. They are not Java or Jini specific. The main focus is on digital signatures, since Java provides built-in support for digital signatures. The second part introduces the concept of a secure communication channel. Therefore it also takes a look at integrity and confidentiality. At the end of this chapter authentication and authorization are described in general.

## 2.1 Digital Signatures

The purpose of a digital signature on any kind of data is to verify the identity of the person who digitally signed the code and to verify the integrity of the data.

A digital signature of an input $a$ consists of two phases. In the first phase the input $a$ is transformed to $b$ using a secure hash function, since the result $b$ is much shorter than the input $a$. In the second phase a signing function is applied to the input $b$ using a private signing key $c$ and returns the digital signature $d$ of input $a$ using the private signing key $c$.

$d = signingfunction(hashcode(a), c)$

The complementary operation is the verification of a digital signature. For that, a verification function takes the digital signature $d$ and a public verification key $e$ that corresponds to the private signing key $c$ and calculates an output $f$. Using the same secure hash function applied on the input $a$ returns $g$. Now $g$ and $f$ are tested for equality.

$verified = equals(verificationfunction(d, e), hashcode(a))$

If the verification succeeds, it has been verified that the given input was digitally signed by someone, who has the private signing key corresponding to the public verification key if some assumptions are hold:

- The secure hash function is collision free: $hashfunction(m) = hashfunction(m')$ and $m \neq m'$ is extremely hard to compute. That means it is extremely hard to find an input $m'$ that corresponds to the original input $m$ and both have the same hash code.
- The private signing key and the public verification key must fulfill two requirements. Given one of both keys, it must be extremely hard to compute the other.

Both keys and only these both are corresponding, so that it is extremely hard to find another verification key that corresponds to the signing key and vice versa.

- The signing function must fulfill the same requirements as the hash function and must generate different results for different signing keys. Therefore, the signing function can be seen as a hash function with a parameter: the signing key.
- The verification function that uses the public verification key has to reproduce the same input which was given to the signing function that used the private signing key.

Only if all these assumptions are fulfilled, the conclusion that only someone who had known the private key that corresponds to the used public verification key could have signed the data, is a valid conclusion. And that is the only interpretation which is possible in this model.

These are the theoretical aspects of digital signing, but the expression "signing" suggests one application of that mathematical theory: in real life, a document is signed by a person with his signature. The document matches with the input and the person's hand-written signature matches with the digital signature. The private key corresponds to the ability of the person that the person and only this person can sign the document with his hand-written signature. The interpretation of the public verification key is the ability of a handwriting authenticator to authenticate a signature as the signature of that person who signed it. This is only one kind of interpretation, there could also be other interpretations.

There remains one very important fact to mention. This interpretation only works if the person who uses the private signing key is the only person who knows the key. In real life a signature can be faked. But this can be detected by an expert, whereas publishing a private signing key enables everybody to digitally sign an electronically document without the possibility to detect this by an expert.

Thus, signing code and the verification with a public verification key only implies that someone who has the corresponding private signing key signed the code and that the data has not been altered. That means, it does not imply authenticity, referring to the verification of the identity of someone, although it used for that purpose. As described above, it cannot be used for that purpose. Authenticity can only be achieved by using an indirection, but not on a technical level, only on a logical level, so that a misuse is still possible. This works by assigning trust to the private signing key, whereby two circumstances are trusted. The first is that the owner of the private key takes care that no third party gets the private signing key and that the digital signing with the private signing key implies a semantic that is the same on both sides.

All the following aspects concern legal aspects. The semantic, which is used, must be unique and therefore has to be written down legally binding. This could be a previous contract, or a data structure inside the code. Furthermore, the private signing key must be kept secret. For an unique assignment of the public verifying key with the owner of the private signing key, the public verifying key must be certified by a public authority. This could be done by signing the public signing key with the private signing key of the public authority. In general, an authority which signs public keys is called Certificate Authority (CA). Thus, the public key of the CA must be trusted. Chains of trust can be built using a tree of CAs. The public key of a CA has to be signed by the CA in the hierarchy above. That means there must be a root CA which has to sign its own certificate. Therefore, its self-signed certificate or the fingerprint of it has to be

published by another medium, e.g. an “official” newspaper. A fingerprint is a human readable hash value of the input, e.g. a hash value of a certificate.

## 2.2 Secure Communication Channels

A secure channel is a communication channel, in which data cannot be read or altered. An example of an insecure channel is an Internet link. Every router has access to all the data that he routes. But it is possible to secure an insecure channel. Confidentiality and integrity have to be achieved. Confidentiality means that only the sender and the trusted receiver can read the message, whereas integrity means that the receiver knows that the received data is the same as the data sent by the sender.

Confidentiality is typically achieved by using symmetric encryption. It is also imaginable to use asymmetric encryption, but this is 1000 times slower than symmetric encryption. A message authentication code (MAC) is the way to ensure that integrity is achieved.

A mathematical view on symmetric encryption and decryption is:

- encryption: $cipher = cryptofunction(data, secretkey)$
- decryption: $data = cryptofunction(cipher, secretkey)$

The assumptions are that the secret key is kept secret on both sides, whereas the cryptofunction can be public. Having the cipher, but not the secret key, it must be extremely hard to compute the original data or the secret key. It should also be extremely hard to compute the secret key if data and cipher is known.

One application of encryption and decryption is the confidential transmission of any kind of data over an insecure channel. If an attacker gets the cipher from the insecure channel, he is not able to recover the original information without the secret key which is only known by parties which are entitled.

The MAC is computed by a hash function which is initialized with a secret key. Since sender and receiver know the secret key, both are able to compute the MAC. The receiver can compare the computed and the sent MAC.

Another problem is the replay of networks packets. A network packet which is neither read nor altered can be received by a third party. Then, this network packet can be replayed several times by the third party. To prevent replays, increasing transaction numbers can be used. That means, the replay of a network packet has an old transaction number which is detected by the receiver.

## 2.3 Authentication

Authentication refers to the process to establish trust that an entity is real, e.g. the authentication of a contract is achieved by the verification of the signatures. In the computer environment, authentication means to verify the identity of an entity, on which behalf code is executed among other things. The entity could be a user that enters his user-name and his password for the verification process. It is also imaginable that the entity is a computer program or a device and the verification is done by a challenge-response procedure.

## 2.4 Authorization

In general, authorization is independent of the authentication process. Authorization means to grant permissions to an entity. The permissions can refer to any restricted resource: physical or logical, e.g. accessing a printer or executing code. In most cases, it is important to know the identity of the entity. For that reason, authorization normally builds on authentication. In this thesis, authorization means to grant an authenticated entity permissions to access a restricted physical or logical resource.

# Chapter 3

# Jini Concepts

This chapter consists of two parts. The first and smaller one takes a look at the computer language Java which is used by Jini. Building on that, Jini and its concepts are described in the second part.

## 3.1 Java

A very short overview of Java is given in this section, since Jini utilizes Java. Fur further information on Java there is very much literature available.

### 3.1.1 History

The very beginning of the computer language Java was in 1991 as Sun's "Green Team" should anticipate and plan for the next wave in computing. Their first conclusion was that one trend would be the convergence of digitally consumer devices and computers. So they tried to create the infrastructure for a interactive network. At that time the cable networks were focused but it came out that there was no business in that field. But they found another candidate, the newly popular Internet. The first version of Java came out in 1995 and was a revolution, because it brought motion into the static Internet. Since then, Java has stormed the computer market.

### 3.1.2 Features

Java is a strong typed object oriented language. The syntax is close to the notion of C++, but the concepts of the language differs strongly. The most important features are

- no pointer arithmetic
- exception handling
- garbage collection
- platform independent compiled bytecode
- applets runnable in a www browser
- dynamic code loading over the Internet

- network support
- thread support
- remote method invocation

The absence of pointers and the automatic garbage collection made it possible to very rapidly write down the code. A very important feature for the Internet is that Java uses compiled bytecode, so that the applets can run in every browser that supports Java independently of the underlaying operating system. The Java applets or applications will be executed in a Java Virtual Machine (JVM) which can use a just in time compiler for a faster execution of the code. One disadvantage of using bytecode is the execution time. A native code program is 20 times faster than a bytecode program, but using a just in time compiler (JIT) speeds up the execution dramatically.

## 3.2 Jini

This section only provides a short overview about Jini. The main focus will be on the security architecture. For a more detailed description many books are available. Jini will only be introduced insofar that it is possible to understand the security architecture. As mentioned in the motivation, Jini is a middleware for service discovery and lookup in a distributed environment. It is built on top of Java which is designed to support distributed environments by using remote method invocation (RMI) and by using dynamic downloadable code. But Jini differs from the first paradigm of distributed systems which will be introduced in the next subsection.

### 3.2.1 Classical Networking

The classical approach was to make the network transparent, i.e. to hide the non local effects, e.g. performance, latency and failure mode. There are good reasons for it. A distributed system is much more complex than a local stand alone one. To design, to implement and to test such a system is much harder compared to a local one. If all the complexity can be reduced, so that a distributed system looks like a local one, no new models are required and everything can be done the known way.

But it turned out that this simplification is an oversimplification. Performance and latency are examples that simplification can also be bad. In general, it is not decidable if a network entity has been crashed or if it is slow, whereas in the local case this is deterministic, e.g. the time to access a hard disk drive is given. So it is decidable if the hard disk works or has been crashed. Another important point is the failure mode. In the local case, the system is correct or not, whereas partial failures can occur in the distributed case, e.g. a multicast message is only received by a few receivers, so that not all entities share the same state. Hiding these facts can lead to an undefined state.

### 3.2.2 Jini's Paradigm

Jini introduces a new paradigm to the distributed world. It provides the marketplace mentioned in the first chapter. It is a middleware for services and clients. They can use Jini and its concepts to find each other and to communicate synchronously and asynchronously. The big benefit of Jini is that services can spontaneously join or leave the Jini community and the Jini concepts do the discovery and the cleanup, e.g. the

VCR service uses the TV service to show a movie. If the VCR service crashes, the TV service can release the resources that the VCR service used, so that another service can use the TV service. A Jini service is very flexible, it can communicate with a physical device and provides its feature, e.g. the TV. It can be a service which uses other services to provide a functionality, e.g. the VCR service which uses the TV service and the storage service. Or it can be a ”stand alone” service, e.g. which provides information like the current time.

To achieve the given properties Jini has five key concepts which will be explained in more detail in the next subsections:

- discovery
- lookup
- leasing
- remote events
- transactions

The following subsections give a very short overview about Jini and its concept. Some minor aspects are left out which are not relevant for this thesis.

### 3.2.3 Discovery

The discovery procedure is the bootstrapping process of the Jini infrastructure. It is used to find the lookup services, the core components of every Jini community. The lookup service, which is a regular Jini service, provides access to the services in several groups in its community. A Jini community enfolds the subnet that is reachable by an IP broadcast. This restriction is very useful, because in the majority of cases only local services are relevant, e.g. to use the TV service at home, all the other TV services in all the other households do not matter. But Jini is not bound to this. A lookup service can also provide services from other communities, e.g. it is possible to start the VCR service at the office. A group whereas is a logical restriction. Services can be in multiple groups and lookup services can provide services from several groups. A special group is the public group which is some kind of a default group.

There are three different ways to discover a lookup service which are driven by the service or are driven by the lookup service:

- When a service or an application starts, they find the lookup service in their community using a multicast request protocol (figure 3.1).
- The muliticast announcement protocol goes another way. A lookup service can announce its existence by using that protocol, shown in figure 3.2.
- The unicast discovery protocol is used if a service or an application knows the address of a lookup service in advance (figure 3.3). Using that information it can contact the lookup service directly which can be in another community. The syntax to specify the lookup service is quite simple. It is a URL containing `jini` as protocol name and a hostname where the lookup service is running. The URL can also contain a port, if not, the default lookup service port `4160` is used, e.g. `jini://marzipan.icsi.berkeley.edu`.

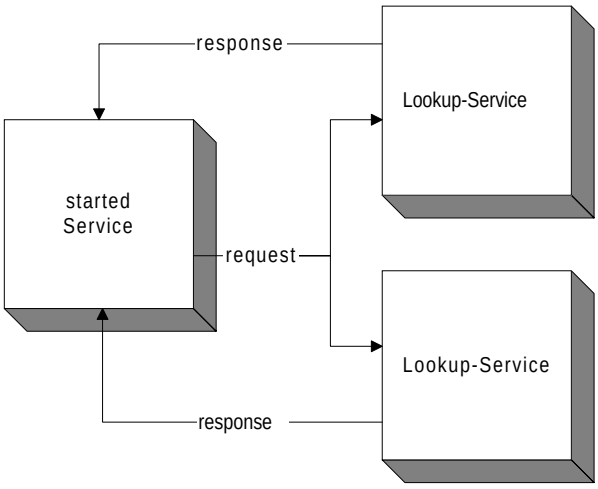


Figure 3.1: Multicast Request Protocol

**Example Code**

First, a service or an application has to instantiate a `LookupDiscovery` object by specifying the groups that it is interested in:

```
LookupDiscovery disco = new LookupDiscovery(groups);
```

Then it registers a `DiscoveryListener` which is only an interface and needs an implementation:

```
disco.addDiscoveryListener(discListener);
```

If new lookup services are discovered or discarded the corresponding methods of the `DiscoveryListener` are called:

```
void discarded(DiscoveryEvent e);
void discovered(DiscoveryEvent e);
```

The `DiscoveryEvent` contains the lookup services.

### 3.2.4 Lookup

Lookup is the process of retrieving a desired service. Since Jini is based on services, this process is done by a service called the Lookup Service. This service has to maintain all the other services. That means, it has to handle registration of new services and requests for services. Lookup services are more powerful than normal name services. They have to store and retrieve serialized Java objects, known as Proxies. A request can consist of a Service ID, a Java interface to be implemented by the proxy or a set of attributes. The Jini package provides a implementation of a Lookup Service (LUS), but it can be implemented by anyone else, since the LUS is just an interface description.

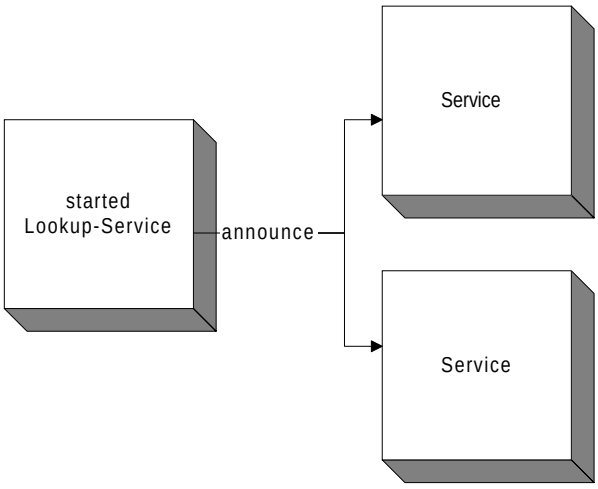


Figure 3.2: Multicast Announcement Protocol

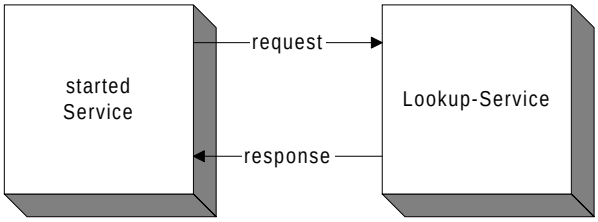


Figure 3.3: Unicast Discovery Protocol

**Example Code**

As shown above, the discovery process finds the lookup services and calls the appropriate `DiscoveryListener` methods. Two things can be done with the lookup services:

- register a service
- search for a service

To register or to search for a service, a communication with the lookup service has to take place. The method `getRegistrars()` of the `DiscoveryEvent` returns an array of `ServiceRegistrar`s. `ServiceRegistrar` is an interface which every lookup service hat to implement, so that different implementations beside Sun's implementation of the lookup service named reggie are possible.

Having a lookup service, the method

```
ServiceRegistration register(ServiceItem item,
                             long leaseDuration);
```

is called to register a service. `leaseDuration` will be explained in the next section. The service itself is located in the `ServeItem` object. Its constructor is:

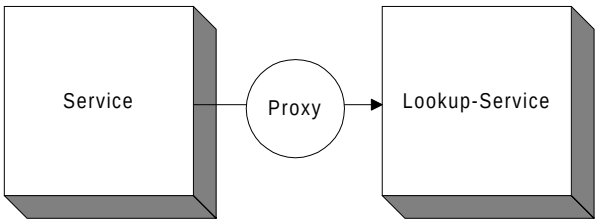


Figure 3.4: Service Registration

```
ServiceItem(ServiceID serviceID, java.lang.Object service,
            Entry[] attrSets)
```

The `ServiceID` uniquely identifies a service in time and place. If a service registers itself the first time, it should store its information persistently, so that it is able to register itself on another lookup service or after a crash with the same `ServiceID`. This can be useful for clients who search exactly the same service. It is also possible for services to re-register themselves, e.g. if a newer version of the service is available. The third argument of the constructor is an array of `Entry`s. `Entry`s are attributes which describe the service, and can be used to search a specific service. A common attribute which most services should provide is the place. In the TV service example, the TV in the living room should be used by the VCR service and not the TV service for the TV in the bedroom if the VCR service is started in the living room.

The second and most important argument is `service` which is declared as an `Object`. This is not the service itself, but a proxy for that service. The proxy must be a serializable object which the service has to specify there (figure 3.4). The use of a proxy is the key idea of Jini. The proxy has to implement a well-defined interface which is known by the client, e.g. the TV service interface which includes methods to turn the TV on or off or to choose the AV source. If a copy of that serialized proxy is downloaded by a client, the client can use the proxy, because it knows the interface which the proxy implements. There are different possibilities to implement the proxy. In most cases it is likely that the proxy contacts the service backend via RMI or any other protocol, calls the corresponding methods and returns the result if the method has to return one. Another possibility is that the proxy implements the whole functionality and does not contact any other entities. It can also directly communicate with an appliance. Using that technique, the problems of not installed drivers are obsolete, because the ”implementation of the driver” is hidden by the proxy, so that a client only has to know the interface. However, using the created `ServiceItem`, the service can register itself on the lookup service.

A service can also be a client if it uses another service, e.g. the VCR service is such a service because it provides the functionality of a VCR. But it is also a client because it uses two other services: the TV service and the storage service. The client also uses a `LookupDiscovery` object to get the lookup services. But a client has not to register a service. It has to find a service using the

```
public java.lang.Object lookup(ServiceTemplate tmpl)
                        throws java.rmi.RemoteException
```

method of the `LookupDiscovery` object (figure 3.5). For that purpose it has to create a `ServiceTemplate` object. Its constructor is:

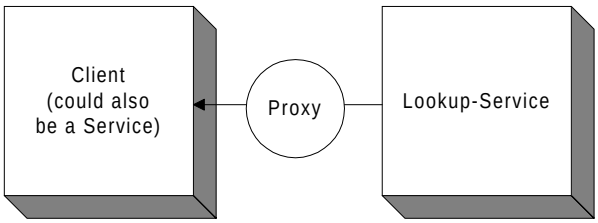


Figure 3.5: Finding Service

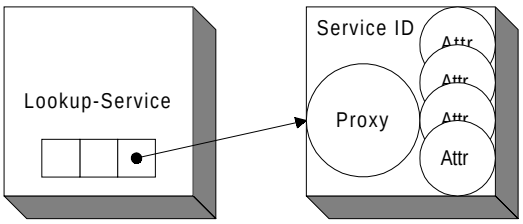


Figure 3.6: Lookup Service Entry

```
ServiceTemplate(ServiceID serviceID,
                java.lang.Class[] serviceTypes,
                Entry[] attrSetTemplates)
```

`ServiceID` and `Entry` have already been introduced. A client can specify the desired ServiceID and the attributes that the services must provide (figure 3.6). `null` can be used as wildcard. The second argument `serviceTypes` is very powerful. It is possible to specify which interfaces the proxy should implement. As mentioned above the interface defines the protocol of how a client can communicate with the service. A client does not care about the implementation. It only expresses that it needs a service, resp. its proxy that can ”speak” the protocol, i.e. that implements the given interface, e.g. a remote control specifies, it needs a service which ”speaks” the TV service protocol. The lookup services then returns a proxy which is able to ”speak” this protocol if one is available. Another important feature is that the proxy does not have to implement it directly, it can also be implemented as an extended interface, e.g. the TV service provides a proxy that implements the TV service interface version 2, but the remote control asks for a proxy that implements the TV service interface version 1. If the version 2 interface extends the version 1 interface, the lookup services returns the version 2 proxy and the remote control uses only a subset of the version 2, i.e. the version 1 interface to communicate with the service. -

Summarizing to publish a service:

- Registration of a `DiscoveryListener` that will be invoked if a lookup service is found.
- Registration of a proxy on a lookup service using a `ServiceRegistration` object.

Summarizing to find a service:

- Registration of a `DiscoveryListener` that will be invoked if a lookup service is found.
- Lookup of a proxy on a lookup service using a `ServiceTemplate` object.

### 3.2.5 Leasing

Leasing is the self-healing capability of Jini that does implicitly the cleanup of no longer used resources, e.g. the TV service stores context information about the VCR service. This could be a GUI that shows the status of the VCR on the TV screen. For various reasons, the VCR service crashes and cannot release the context information which is stored in the TV service. If that happens a few more times, the storage of the TV service will be exhausted. The idea of leasing, which prevents such situations, is not to grant permission to access a resource for a undefined amount of time, but for a explicit amount of time. If there is no request to renew the lease until it expires, all resources that are assigned to that lease will be released, e.g. if the VCR service cannot renew its lease, because it has been crashed, all the context information will be cleared by the TV service after the lease has been expired.

Another important scenario for leasing is the use of mobile devices which could spontaneously join and leave Jini communities. And that is exactly what Jini is built for. A new service can spontaneously join the Jini community by registering itself on a lookup service. The lookup service itself uses leasing to keep track which services are still available. If the service is no longer available for various reasons, the lookup service discards the proxy. It is up to the service how to implement and how to hand out the leases. The lease has to implement the `Lease` interface which only defines methods to cancel or to renew a lease. The lookup service for example hands out the leases via the `ServiceRegistration` and its `getLease()` method. The renewal of leases is a standard task which can be done by a `LeaseRenewalManager`. A client only has to instantiate the `LeaseRenewalManager` using the lease as one of the constructor's arguments.

### 3.2.6 Remote Events

In general, events can be used for asynchronous communication. For example, a user interacts with a GUI asynchronously. If the users clicks on a button, an event will be generated and depending on the implementation, one or more parties will be notified. In the local case, the events are ordered and the notification is guaranteed, whereas in the remote case a partial failure can occur, i.e. a few events get lost or the ordering is not guaranteed. There are also some other problems which can partly be fixed but only with loss of performance. Jini provides a very generic kind of events, i.e. only one type that can be extended: `RemoteEvent`. A client that needs notification of events has to implement the `RemoteEventListener` interface which only defines the `notify()` method which the client has to implement. Then it has to register itself on the service which returns an `EventRegistration` object which contains the lease for the notification among other things. Every time an event occurs, the `notify()` method will be called. For example, the VCR service registers itself on the TV service to be notified if the TV service has been shut down.

### 3.2.7 Transactions

Jini also defines a two phase protocol for transactions. A transaction is a collection of more than one action that are logically performed atomic, i.e. all actions succeed or none. The interface `Transaction` provides the methods `abort()` and `commit()` that must be implemented by every client that performs on a transaction. A centralized transaction manager - a Jini transaction manager service is included in the package - calls the `abort()` method if one of the clients failed, or calls the `commit()` method if all clients succeeded.

# Chapter 4

# Java Security

This approach is based on the security features of Java. Security is a big issue in Java because it supports dynamic loading of code over the Internet. That means Java applets which could run in a client browser, are downloaded by a server in the Internet which could be malicious. Therefore Java has to provide security features to prevent that an unknown applet executes malicious code.

## 4.1 Security in older JDK Versions

With the first version of Java (JDK1.0), Sun introduced the Sandbox model, in which downloaded code has no access to any security relevant resources (figure 4.1). JDK1.1 extended the Sandbox model with the signing of code. A class file and its digital signature will be checked at client side. If the signer of the code is trusted and the signature could be verified, the applet gets the same permissions like a local class. If the signer of the code is not trusted or the signature is not valid, the applet will be executed in the Sandbox (figure 4.2).

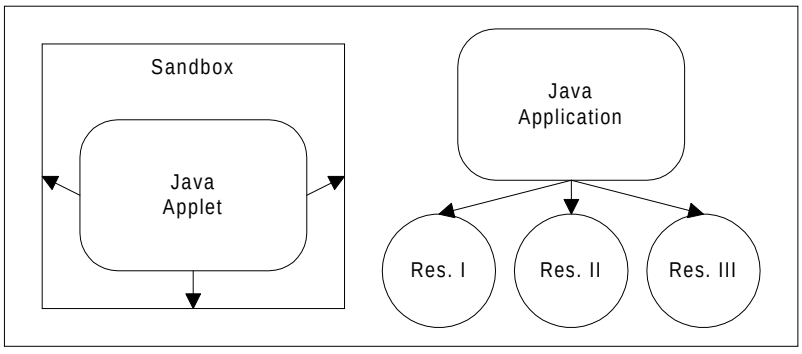


Figure 4.1: Security Model JDK1.0

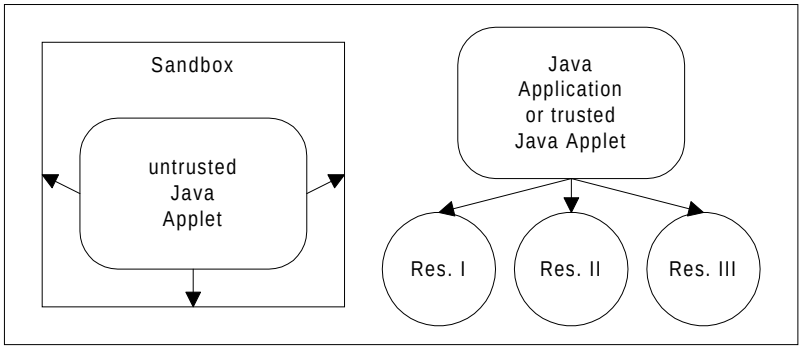


Figure 4.2: Security Model JDK1.1

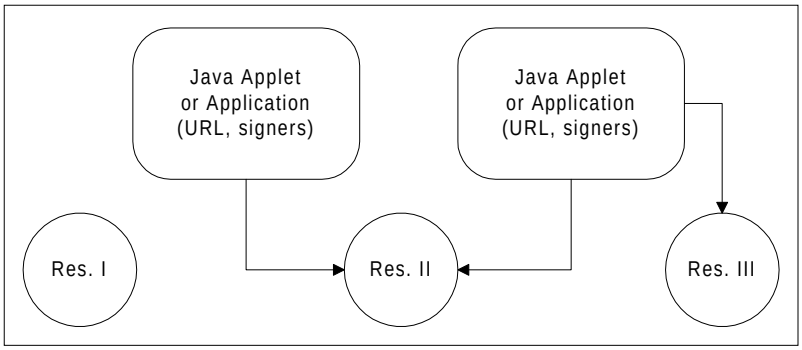


Figure 4.3: Security Model JDK1.2

## 4.2 Security in JDK1.2

JDK1.2 introduced fine grained access control to all security relevant resources. It uses `Permission` classes and subtypes of it to represent the rights to access a resource, e.g. `new FilePermission ("c:\config.sys", "read")` represents the permission to access the file `c:\config.sys` for reading only . These permissions can be granted on the basis of the codebase, by whom the code is signed or without any restrictions (figure 4.3). That means the Sandbox model and the explicit distinction between local and downloaded code is no longer needed because the functionality of the older models is a subset of the new one. A key store is used to store the private signing keys and the certificated public verifying keys for digital signatures. To specify where the key store is located and which permissions are granted, a policy file is used which can be an argument to the JVM or referenced by the default security settings. The policy file itself is an ASCII file which contains the location of one key store and multiple permission granting entries. A simplified grammar for those permission granting entries is:

```
grant [codebase url] [signedby signers]{
   {permission}*
}
```

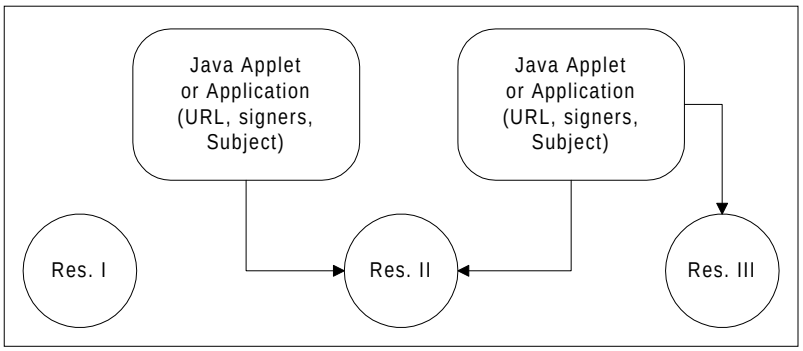


Figure 4.4: Security Model JDK1.3

The analysis of such a policy file is done by the JVM, so that the user only has to specify the policy. The program `policytool` provides a GUI for simple editing of a policy file. Certificates and private signing keys are stored in a key store. Another program `keytool` provides command line access to the key store. A programmatically access is provided by instantiating a `Keystore` object that allows signing objects and verifying digital signatures at runtime. The command line program `jarsigner` is used to sign class files inside a jar file. A jar file is an archive which contains classes and other files.

The internal permission check is very sophisticated, but also transparent to the authentication architecture, so that these internal details are skipped here.

## 4.3 Security in JDK1.3 based on JAAS

Granting permissions in JDK1.2 is based on where the code is from and who signed the code. Java Authentication and Authorization Service (JAAS) is a standard extension to the JDK1.3. JAAS extends the JDK1.2 concept by granting permissions based on which behalf the code is executed (figure 4.4). It also contains modules to log on the user. JAAS is based on the Pluggable Authentication Modules (PAM) framework which defines an API for applications to log on users. Thus, every application can use the PAM framework which can be configured by an administrator. Consequently, PAM is independent of which application uses the authentication mechanisms and configuring the authentication policy can be done separately. Next, the concepts of JAAS will be described more in detail: Subjects and Principals.

### 4.3.1 Subjects and Principals

`Subject`s and `Principal`s are used to represent the identity of a user and its login names. The distinction between `Subject`s and `Principal`s is very important because a real life person can have different login names for different services. That means that a real life person has a collection of different login names (figure 4.5). In that manner `Prinicipal` is definded as:

```
public interface Principal{
   String getName();
   ...
```

```
}
```

The name of the `Principal` can be interpreted as a login name. As mentioned above, a `Subject` is a collection of `Prinicipal`s which can also include private and public credentials, e.g. public certificates or Kerberos tickets. Accordingly, a `Subject` contains all the data that a user needs to log on:

```
public final class Subject{
   Set getPrincipals(){};
   Set getPrivateCredentials(){};
   Set getPublicCredentials(){};
   ...
}
```

The private and public credentials can be any Java `Object` and therefore they are not typed. Credentials can be used, but in many situations, they are not needed (figure 4.5).

PAM demands that the login procedure is independent of the authentication technology, i.e. which authentication is used. For that purpose, JAAS defines a `LoginContext` and a `LoginModule`.

A `LoginModule` represents one authentication technology, e.g. username password authentication (figure 4.5). Every authentication technology must implement the two phase protocol similar to the two phase transaction protocol used by Jini's Transaction Manager:

```
public interface LoginModule{
   boolean login();
   boolean commit();
   boolean abort();
   boolean logout();
   ...
}
```

The regular sequence is:

- execution of the authentication technology using `login()`
- commit the result using `commit()`
- cleanup the system after logout using `logout()`

This sequence can be interrupted at any time, calling `abort()`. Calling `abort()` or `logout()` should clear all confidential data that are currently stored. On this account, all confidential data should be stored in primitive data types, e.g. `char[]`, because the values they store can easily be overridden.

The central "transaction manager" in this scenario is the `LoginContext`. It coordinates the login procedure by using multiple `LoginModule`s and calls their appropriate methods:

```
public class LoginContext{
   LoginContext(String, CallbackHandler){};
   Subject getSubject(){};
```

```
   void login(){};
   void logout(){};
    ...
}
```

To perform a login session, the following steps have to be executed:

- instantiation of a `LoginContext`
- calling its `login()` method
- if `login()` returns, `getSubject()` can be called
- in the end `logout()` should be called

If a `Subject` has been returned, i.e. no exception was thrown, e.g. the password was wrong or the user is unknown, the authentication has been succeeded and the `Subject` represents the authenticated user. But a few topics are still unsettled. Which `LoginModule`s are used? How does the `LoginModule`'s interact with the user? And at least, how is authorization achieved?

JAAS provides a stackable authentication framework. That means multiple `LoginModule`s can be specified in a login policy file (figure 4.5). The grammar for one entry called "name" in the policy file is:

```
name{
   {package.LoginModule flag [options]}*
};
```

The name is also given as an argument to the constructor of the `LoginContext` which uses it to lookup the `LoginModules` in the login policy file. Multiple `LoginModule` including their package name can be specified. A flag can be

- REQUIRED - if the `LoginModule` fails, the overall login procedure fails and the remaining `LoginModule`s will be processed,
- REQUISITE - if the `LoginModule` fails, the overall login procedure fails and the remaining `LoginModule`s will not be processed
- SUFFICIENT - if the `LoginModule` succeeds, the overall login succeeds and the remaining `LoginModule`s will not be processed
- OPTIONAL - if the `LoginModule` succeeds or fails, the overall login does not depend on it and the remaining `LoginModule`s will be processed.

If one of the processed REQUIRED or REQUISITE modules fails, the overall login procedure fails. If none of those modules are specified, at least one SUFFICIENT or OPTIONAL must succeed. Combining `LoginModule`'s with these four flags: very sophisticated login schemes can be built. (Note: if only one `LoginModule` is specified, is does not matter which flag is used, but it is recommended to use the REQUIRED flag because if the list maybe extended, no security hole can arise.)

The login policy file can be specified statically in the Java security settings or as a command line argument to the Java interpreter:

```
-Djava.security.auth.login.config=login.config.file
```

Next, the user interaction will be focused, e.g. prompting for a user-name and a password. JAAS uses the callback model. That means a `CallbackHandler` (figure 4.5) has to be instantiated which `handle()`'s `Callback`'s, instantiated in the `LoginModule`. The `CallbackHandler` is the second argument of the `LoginContext`'s constructor.

The `CallbackHandler` defines only one method:

```
public interface CallbackHandler{
   void handle(Callback[] callbacks)
}
```

`Callback` is an interface which defines no methods so that it is only used as marker and for type safety. A `LoginModule` can call the `handle()` method of the `CallbackHandler` with an array of `Callback`'s which will be performed by the `CallbackHandler`. To build a user-name password authentication module, the predefined `NameCallback` and `PasswordCallback` can be used.

As mentioned above, if the authentication procedure succeeds, a `Subject` will be returned. Actions that need authorization must implement the `PrivilegedAction` or `PrivilegedExceptionAction` interface, depending on throwing an exception is required. These interfaces only define the `Object.run()` method. To perform that action, the static method `Subject.doAs()` will be invoked, with the `Subject` and the `Privileged{Exception}Action` as arguments. The `doAs()` method performs a lookup in an extended policy file if the given `Subject` has got the permission, while it is executing the `run()` method of the `PrivilegedAction`. That also means that all the code that need authorization, must be put in the run() method of a `PrivilegedAction` object.

```
grant [Codebase url,]
      [Signedby signers,]
      {Principal package.PrinicipalImpl name,}*{
   {permission_entry}*
}
```

If multiple `Principal`s are specified, the `Subject` must contain all the `Principal`s. This policy file that extends the JDK1.2 policy file, can be statically stated in the Java security settings or given as a command line argument to the Java interpreter:

```
-Djava.security.auth.policy=auth.policy.file
```

### 4.3.2 Summary of the JAAS Concepts

A `Subject` represents a user. It contains `Principal`s which represents the users login names. In an extended policy file can be specified which `Principal`s a user must have at least, so that a `Permission` can be granted.

To get a `Subject`, a `LoginContext` must be instantiated with a `CallbackHandler` and the name of a login policy which is specified in the login configuration file. Depending on the login policy, multiple `LoginModule` which represents an authentication technique will be invoked which in turn invokes the `CallbackHandler` for user interaction. If the login succeeds, a `Subject` will be created, otherwise an exception will be thrown.

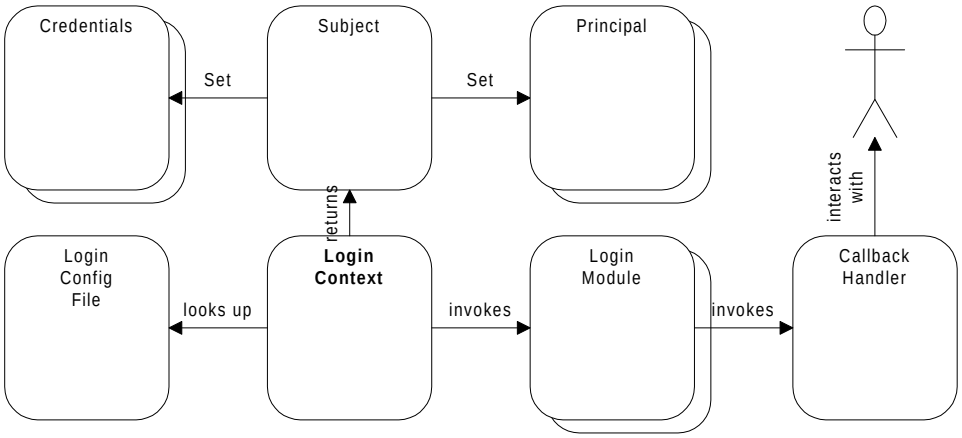


Figure 4.5: JAAS Concepts and Classes

To perform an authorizated action, the `Subject.doAs()` has to be invoked with the `Subject` and the `Privileged{Exception}Action`. The `doAs()` method calls the `run()` method and checks the permissions during the execution of the `run()` method.

## 4.4 Signed Code in Java

A common problem of dynamically loaded code is the unawareness if the code does what it is expected to do, especially does not perform any malicious actions, e.g. formatting the hard disk. For that purpose Java uses digital signature of code.

In this architecture, two assumption are made about signed code:

- the signer of the code keeps its private signing key secret.
- the semantic of signing code is that the signer guarantees that using code signed by itself does not lead to any disadvantages for the user, e.g. transmitting secret data without encryption over insecure channels like the Internet.

These assumptions are also made by Java for downloaded code. This implies that the user has to trust the signer and has to verify the truthfulness of the certificate that the user utilizes to verify the digital signature of the code.

It is up to both parties, how the user gets the certificate from the owner. It is also up to the user if he trusts a self-signed certificate. In that case, the certificate is only used as a container for the public key. It is also possible to ask for that the certificate has to be signed by a trusted CA.

These both assumptions above does not overload the current semantic of signed code. It is exactly the way Java intends signed code to be interpreted: a negative semantic that says what the code does not do. It is not a positive semantic: there is no notion, what the code is exactly doing. This fits perfectly to the architecture, where the authentication procedure is transparent to the client. If authentication is used, and the user trusts the signer, the code is expected to do no malicious actions which could harm the users interests.

The user has to do two things which he also has to do if he uses Jini services without authentication because unsigned code from the Internet is in general not trustable. He has to setup the JDK1.2-style policy file and has to grant appropriate permissions only to signed code which is signed by a trustworthy entity. He also has to put the certificate into the default Java key store. The default location of the key store is `file:/userhome/.keystore`. `keytool` is a command line program for that purpose.

### 4.4.1 Example

Suppose, the file `johncert.cer` contains a certificate from John Smith which the user wants to put into the key store using the alias `john`. First, he has to make sure the certificate can be trusted:

```
> keytool -printcert -file johncert.cer
```

`keytool` prints the fingerprint of the certificate among other things which the user can validate. If the fingerprint is valid, he could import the certificate:

```
> keytool -import -alias john -file johncert.cer
```

But before a certificate can be imported at client side, the certificate and the assigned private signing key must be generated at service side, also using the command line program `keytool`:

```
> keytool -genkey -alias john -keypass 123456
```

The `keypass` argument specifies a password that is only used to check the integrity of the key store. If the program is started, a dialog asks for some information, like name, organization, etc. It also prompts for a password for the key. Whereas this password is used to encrypt the private key. If return is pressed, the same password like for the key store integrity is used. A user should be very careful. If an attacker gets the private key and the associated certificate has been issued by an public authority, the attacker can sign contracts instead of the authoriziated user or can do other malicious actions.

A policy file could look like:

```
keystore "file:/${user.home}/.keystore", "JKS";

grant signedBy "john"{
    permission java.security.AllPermission "", "";
};

grant{
    permission java.net.SocketPermission "*:1024-",
    "connect,accept";
};
```

In this example the default key store is used. Only code that is signed by `john`, i.e. it is the alias for the certificate in the key store, is granted the `AllPermission`. The last granting entry is specified because otherwise the class loader used by RMI cannot

be instantiated. It is not insightful, why this grant entry is needed. Java core classes need no permissions and if all classes, which are used, would be signed by `john`, the last grant entry is still needed.

Using signed code as described above, the JVM detects if the code has been altered during the transmission. It also detects if an entity, who is not trusted registers or re-registers a service on the lookup service.

## 4.5 Java Cryptography Architecture

The Java Cryptography Architecture (JCA) provides several classes which make it possible to encrypt and decrypt Java objects. All the classes that are needed for that purpose are introduced in the next sections. The classes needed for encryption are shown in figure 4.6. Figure 4.7 gives an overview about all signature relevant classes.

### 4.5.1 Cipher Object

The encryption and decryption is done by the Cipher object in the JCA:

```
class Cipher{
   static final Cipher getInstance(String, String){};
   final void init(int, Key){};
   final byte[] doFinal(byte[]){};
   ...
}
```

In general, every object in the Java Cryptography Architecture is instantiated via a factory. That means a cryptography provider has to be registered with the JVM. This can be done statically in the Java security settings or dynamically:

```
Security.addProvider(provider);
```

If an instance of a cryptography class is needed, the most common way is to call the static `getInstance()` method with the algorithm name and the provider name as parameter, e.g.

```
Cipher cipher = Cipher.getInstance(‘‘DES’’, ‘‘SUNJCE’’);
```

The provider is optional. If none is specified, the first provider in the provider hierarchy is taken. The hierarchy is a priority list of providers, whereby it is possible to specify the priority of each provider. The default behaviour is that a new provider will be added to the end of the list.

This architecture does not dictate to use a specific provider. It is recommended to use a non US cryptography provider outside the USA, because cryptography products are export restricted. Algorithms are suggested by the architecture. It is not obligatory to specify the algorithms, they could also be negotiated during runtime. But specifying the algorithms makes it is easier to switch to another provider because the minimum set of to be supported algorithms is known.

The suggestion for the algorithm used by the Cipher class is DES, using the default values for mode and padding. The mode specifies variants of the DES algorithm, e.g. how the blocks are ordered. DES is a block orientated algorithm. That means that

not the whole input is encrypted at once, but block by block. The default value is the Electronic Codebook Mode (ECB). Padding means, how missing bits in the last block are filled. The default is PKCS5Padding.

The next step is to initialize the Cipher object. There are several ways, whereby the most common way is to use the init() method with mode and secret key as parameters. Mode can be `ENCRYPT_MODE` or `DECRYPT_MODE`. To encrypt or decrypt data, which have to be a byte array, the method doFinal() is called with the data as argument. It returns the transformed byte array.

### 4.5.2 SealedObject

Handling byte arrays is not very comfortable if objects are used. For that purpose, JCE provides the class SealedObject:

```
class SealedObject implements Serializable{
   SealedObject(Serializable, Cipher);
   final Object getObject(Key);
   ...
}
```

The constructor expects a serializable objects, because only byte[] can be encrypted and a initialized `Cipher` object for encryption. After instantiation, the object can be transmitted confidentiality over an insecure channel. On the other side, only the method `getObject()` with the common secret key has to be called. The `SealedObject` decrypts and deserializes the data and returns the original object.

### 4.5.3 KeyAgreement

`Cipher` and `SealedObject` provide a very simple way to exchange Java objects over an insecure channel with confidentiality.

But one premise is that both parties share a common secret key. Therefore, the secret key is very often a session key, it must be exchanged over a secure channel or has to encrypted asymmetrically. A better approach is to use a key agreement algorithm. Both sides generate a pair of private and public keys for the key agreement algorithm. They exchange their public keys and generate with their own private key and public key from the other side the same secret key.

```
class KeyAgreement{
   static final KeyAgreement getInstance(String, String){};
   final void init(Key);
   final Key doPhase(Key, boolean);
   ...
}
```

The usage is straight forwarded. First the `getInstance()` method is called to get an instance, then the method `init()` is called, with the own private key. Depending on the algorithm used, the `doPhase()` method is called as often, as the protocol requires, the `boolean` value indicates if the last phase has to be performed.

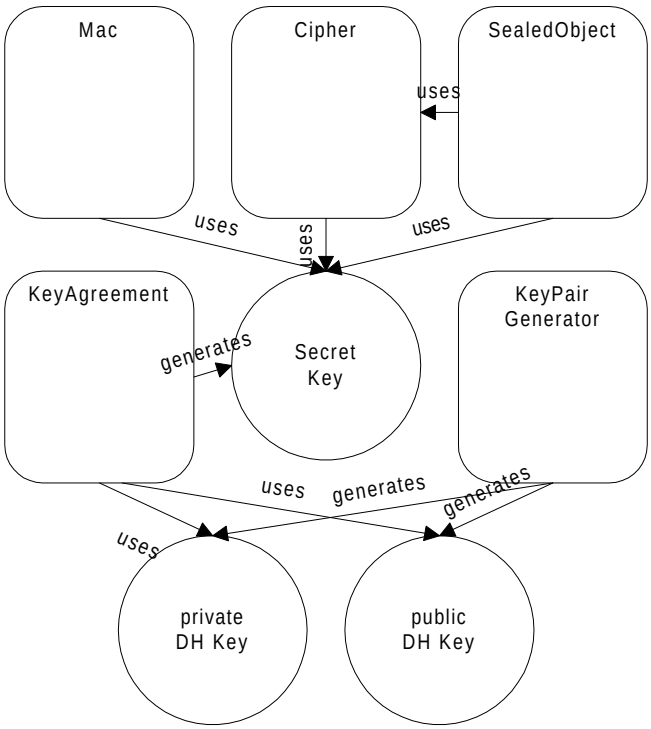


Figure 4.6: Encryption Classes

### 4.5.4 KeyPairGenerator

To create the key pair, the `KeyPairGenerator` is used:

```
abstract class KeyPairGenerator {
   static KeyPairGenerator getInstance(String, String);
   void initialize(AlgorithmParameterSpec);
   final KeyPair genKeyPair();
   ...
}
```

First, the instance is created by calling `getInstance()`, then it has to be initialized with some parameter which the algorithm needs and at least the key pair can be created by calling `genKeyPair()`. `KeyPair` is only a container class that stores the private and public key.

### 4.5.5 Usage of the Diffie Hellman Key Agreement Algorithm

As mentioned above, the architecture suggests a default algorithm. The key agreement algorithm suggestion is the Diffie Hellmann key agreement algorithm (DH) which uses Diffie Hellman key pairs (DH). The parameters needed to generate the keys - exponent and modulus - are pregenerated, public and taken from Suns Diffie Hellman example program. DH is a one phase protocol, so that the `doPhase()` method has only to be called once with true as argument because the first phase is the last phase.

Using DH prevents passive attackers from decrypt the data. However, an active attacker could start a man in the middle attack by creating its own private and public keys, intercepts the public keys of both parties and gives them his public key.

But this could easily prohibited by using the digital signature which was introduced in chapter before. As mentioned above, the service consists of two parts: the service proxy and the service backend. And the service proxy has to make sure that the communication between itself and the backend is confidential and integer. If the public key of the service backend is digital signed, the proxy can detect if there was an active attacker, who altered the public key or exchanged it with its own. In the other direction no digital signature of the public key needed. After the client verified the signature of the public key from the service backend, he is able to generate the secret key. Then he transmits his unsigned public key with already encrypted data to the service backend. If there is an actice attacker between both, he cannot encrypt the data, because he has no notion about the secret key. The public key from service proxy is useless for it. Manipulating the public key or the encrypted data would be detected by the service backend. The only thing is that he is able to generate his own public key, encrypt its own data and send that to the service backend. This situation is not dangerous because that is equivalent for service backend to the situation that the regular service proxy never wanted to log on and the attacker asks for a normal log on that is of course, what the service backend is for.

### 4.5.6 SignedObject

It is very important to notice the difference between the signing of static code and dynamically data. If the service proxy is digitally signed and contains a public key that was generated at runtime, the public key is not signed, only the code is signed. The explanation is very simple: the code was signed, before it was executed the first time using the `jarsigner` command line program. The public key was generated afterwards during runtime. So it has to be signed explicitly during runtime. Therefore, the Java Cryptography Architecture provides a `SignedObject` that is akin to `SealedObject`. The difference is that a `SealedObject` guarantees confidentialty and a `SignedObject` guarantees integrity:

```
final class SignedObject{
   SignedObject(Serializable, PrivateKey, Signature){};
   boolean verify(PublicKey, Signature){};
   Object getObject(){};
}
```

### 4.5.7 Signature

To sign an object, it has to be the first argument of the constructor. The other arguments are the private key and the signing engine that uses the private key and the signing function to compute the signature:

```
abstract class Signature{
   static Signature getInstance(String, String){};
   initSign(PrivateKey){};
   initVerify(PublicKey){};
   final int sign(byte[], int, int){};
```

```
   final boolean verify(byte[]){};
   ...
}
```

The developer only has to instantiate the `Signature` class and use it as argument to the constructor of `SignedObject`, resp. the `verify()` method. The algorithm suggestion is the `SHA1withDSA` algorithm. That means the Secure Hash Algorithm that first hashes the input and Digital Signature Algorithm computes on it the digital signature.

### 4.5.8 Keystore

`jarsigner`, the command line program, uses the key store to retrieve the key needed for signing and the JVM uses it to retrieve the key for verifying. The tool `jarsigner` and the JVM use internally the class `Keystore` to access the proprietary key store file. The same can be done by a developer. That means to sign and verify the public DH key, no extra data is necessary. Only the already given infrastructure is used.

```
class Keystore{
   static Keystore getInstance(String, String){};
   final void load(InputStream, char[]){};
   final Key getKey(String, char[]){};
   final Certificate getCertificate(String);
   ...
}
```

Consequently, the same design pattern is used to get an instance of a `Keystore` class, using the `getInstance()` method with key store type and provider as arguments. The suggestion is to use the standard Java format JKS because it is part of the JDK distribution. To retrieve the key store data, the file has to be loaded via `load()` and an `Inputstream` that reads the file and a key store password. This password is only used to check the integrity of the key store. If no password is given, no check will be performed. At service backend side, the public DH key has to be signed, so the private signing key has to be retrieved from the key store via the `getKey()` method, whereas the alias of the entry and a password for the key are the arguments to that method. The password for the key is required because the key is stored password encrypted in the key store, therefore, to encrypt the key, the password is needed. On service proxy side the `Certificate` and the including public verifying key have to be retrieved. The `getCertificate()` method only needs the alias name of the entry because a certificate is public and therefore does not need to be treated confidential.

### 4.5.9 MAC

The last cryptography concept that is used by the JCA, is the message authentication code. It is important to notice that encryption does not guarantee integrity. It only guarantees confidentiality. If an active attacker manipulates an encrypted message that contains a serialized Java object, it is not likely that the result is a valid class because the attacker has no notion about what is the result of his manipulation. But it is much more likely that a valid class arises after deserilization of the altered encrypted data than finding the secret key. For example, the attacker knows the position of a field in the serialization stream that is encrypted. Only manipulating this location leads

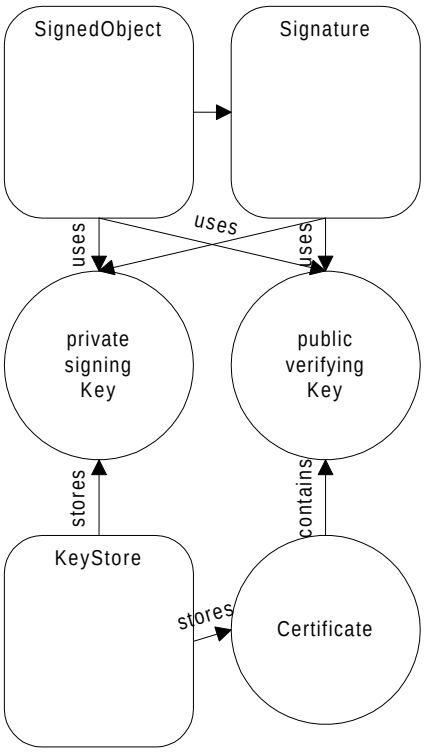


Figure 4.7: Signature Classes

to a different field value. In a bank environment, where the correctness of values is crucial, integrity must be achieved. A simple approach would be to sign the data but then two problems arose. First, asymmetric operations are very time consuming, so this would pervert the use of symmetric encryption. Another lack is that the service backend would also need to store certificates. Using message authentication codes (MAC) solves both problems. They are computing hash values on the input using the secret key as parameter. That means it is a fast symmetric operation and therefore no extra certificates are needed. In addition, the secret key generated for the symmetric encryption can also be used to compute the MAC.

```
class Mac{
   static final Mac getInstance(String, String){};
   final void init (Key);
   final byte[] doFinal(byte[]);
   ...
}
```

As shown many times, the `getInstance()` methods return a new instance of a `Mac` object. Algorithm and provider have to be specified. The suggestion is to use the `HmacMD5` algorithm. Calling `init()` with the existing secret key initializes the `Mac` object. `doFinal()` computes and returns the MAC as byte array. The input must be a byte array, too. In this situation, no wrapper class exists, like `SealedObject` or `SignedObject`, so it is a good idea to develop such a class. The benefits of

those wrapper classes are that the serialization and deserialization process is transparent and that they perform the computation and verification automatically. The verification has to be done by computing the MAC on receivers side and compare it to the MAC attached to the message.

### 4.5.10 Summary

This summary gives an overview of all used keys and classes.

Following keys are used:

- private signing keys, stored in the key store at service backend side, used to sign static code and the dynamically data, created along with the certificate with the command line program `keytool` at service backend side,
- self-signed certificate containing the public verifying key, stored in the key store at client side, used to verify signed code and dynamically signed data, created together with the command line program `keytool` at service backend side,
- private DH key, created dynamically along with the public DH key on both sides, stored locally, needed for creation of secret key,
- public DH key, created dynamically along with the private DH key on both sides, service backend signs it and puts it in the service proxy, service proxy sends it unsigned to the service backend,
- secert key, created dynamically on both sides with the own private DH key and the other public DH key, used to encrypt and decrypt data symmetrically and to compute a MAC.

Following classes of the `java.security` and `javax.crypto packages` are used:

- `Cipher`, used to encrypt and decrypt data, needs the secret key for initialization; input and output are byte arrays,
- `SealedObject`, used to encrypt and decrypt Java objects, needs a serializable object and a `Cipher` object for encryption, needs the secret key for decryption,
- `KeyAgreement`, used to compute a secret key that is only known by two parties over an insecure channel, needs the own private DH key and the other public DH key,
- `KeyPairGenerator`, used to generate a pair of public and private keys, used here to generate the private and the public DH key,
- `Signature`, used to compute and to verify a digital signature, needs the private signing key or the public verifying key for initialization, input and output are byte arrays,
- `SignedObject`, used to sign and verify signatures on Java objects, needs a serializable object, the private signing key and a `Signature` object for signing, needs the public verification key and a Signature object for verification,

- `Keystore`, used to store and retrieve private keys and public certificates persistently, needs an `InputStream` and optional a key store password for initialization, needs an alias to retrieve public certificates and private keys, whereas last one needs a password obligatory, used here to store the private signing key and the public verifying key inside a certificate,
- `Mac`, used to guarantee the integrity of a message, needs the secret key for initialization, used here to guarantee the integrity of the encrypted data.

# Chapter 5

# High Level Overview of the Architecture

This chapter provides a high level overview of the architecture. It starts with the highlights, followed by the goals. Then the architecture and their components are shown in the next two sections.

## 5.1 Highlights

- Built on standard off the shelf Java technology: JAAS 1.0, JDK 1.3, JINI 1.0.1, JCE 1.2.1, JavaCard 2.0
- Transparent to the Client: existing clients do not have to be modified
- Minimal overhead at server side
- Powerful Login Policies allow flexible authentication schemes
- Security Infrastructure built as Jini services
- First prototype up and running, includes smartcard authentication (Java Ring)
- Authorization mechanisms (JAAS 1.0 + Java Security Architecture)

## 5.2 Goals

The system was designed with the following goals in mind:

- Service Authentication and Authorization
- Definition of different Login Policies
- Client Transparency

### 5.2.1 Service Authentication and Authorization

Every service has to authenticate and authorize each client request. The client has to make sure that it is running only signed code from a trusted source.

### 5.2.2 Login Policies

A user should interact as little as possible with the login infrastructure, therefore a single sign on approach is used where appropriate. Especially in the home environment, a user typically does not want to authenticate himself if he turns on the light.

The login policy should be as flexible as possible. Different services in different environments can specify different policies to log on the user. For example, a user at home can turn on the lights without authentication, in his office he has to enter a user-name and password and on the Internet he may have to authenticate using a Java Ring.

### 5.2.3 Client Transparency

The security infrastructure should be transparent to the client implementation. Existing clients do not have to be changed in order to use the security infrastructure. The user does not have to specify the authentication method, this will be determined through the login policy at runtime by the system.

## 5.3 Architecture

Next, it is reported how these goals were achieved. This architecture consists of different Jini services that implement the security infrastructure.

### 5.3.1 Client

In order to achieve client transparency, all security related functionality is packed into the service proxy. The client uses the published interface of the service, as if it were an insecure service.

### 5.3.2 Service Proxy and Service Backend

Upon service discovery, the client loads the appropriate service proxy from the Jini lookup-service that in turn communicates with the service backend. To provide privacy and to prevent replay, we use symmetric encryption of the data, message authentication code and add a transaction number. The symmetric encryption is achieved using the Diffie Hellman-Algorithm: the backend service generates its public and private keys, the private key remains at the backend service, the public key will be transferred to the service proxy. After a proxy has been downloaded by the client, the proxy generates its own private and public keys and with its own private key and the backends' public key, it generates the symmetric secret session key and sends its public key back to the backend, so that the backend can generate the symmetric secret key which will be stored in a session database.

Since the client runs only signed and trusted code, and the communication between service proxy and service backend is secure using the mechanism described above, privacy is achieved and replaying and changing of data between the client and the server is prevented.

## 5.4 The Components of the Security Infrastructure

In the following, it is described how authentication and authorization can be achieved using the above infrastructure.

1. Upon service invocation, the service backend contacts a Jini service called SubjectAuthenticatorService which returns a Subject object introduced in JDK 1.3 / JAAS 1.0. It's up to the SubjectAuthenticatorService, how to generate the Subject. In the reference implementation, we used the JAAS technology, because it is already available and it is easily extendible to be used in a distributed environment.

2. The SubjectAuthenticatorService gets the information to which service the user wants to log on including the number of attempts the user has performed already. With this information the SubjectAuthenticatorService contacts the login policy database: LoginPolicyDB. This can be a separate JINI service or can be built-in the ServiceAuthenticatorService.

3. The LoginPolicyDB returns a LoginPolicy which specifies a policy of how to authenticate the user. Now the SubjectAuthenticatorService contacts a RemoteCallbackHandler which was instantiated by the service proxy at client side. Depending on the login policy, the RemoteCallbackHandler can prompt for a username and password, or initiate even more complex authentication schemes.

4. The RemoteCallbackHandler may contact a JINI RingAuthenticationService which in turn contacts a JINI BlueDotService to run a challenge-response applet on a Java Ring, for example.

5. After the RemoteCallbackHandler has received the result of the authentication procedure, the information is sent back to the SubjectAuthenticatorService which contacts a user database, UserDB, which again can be a JINI service or built-in the SubjectAuthenticatorService. 6. The UserDB checks if all the data is correct and returns the data needed to build a Subject. Based on this data the SubjectAuthenticatorService builds the Subject and returns it to the service.

6. The service finally stores the subject in its context information and returns a token to the service proxy, to be used by the client. This completes the Authentication procedure.

7. To perform actions that need authorization, the service runs the Subject.doAs() methods for the client.

# Chapter 6

# Architecture

Chapter 5 has already introduced the main concepts of this architecture. This chapter provides a very detailed look at the architecture.

Both, interface description and reference implementation of that architecture, are under a GNU public license. Everybody is welcome to extend this architecture and may also contact the authors for that purpose.

## 6.1 Approach

The approach of this thesis is to show how that security architecture can be built on top of the existing Jini and Java environment without changing it. The thesis does not cover denial of service attacks or to secure a lookup service, e.g. hiding services.

It uses Java interfaces to describe the architecture. A reference implementation, which is programed against the interface description, will be explained in chapter 7.

## 6.2 Definition of Authentication and Authorization in the Jini Environment

Authentication and authorization have already been introduced in general. This section additionally explains these terms, how they are used in the the Jini environment.

### 6.2.1 Authentication

Authentication in the Jini services scenario means, on behalf of which entity the service executes code. An entity could be a real life person, a service or an appliance. The authentication of a real life person is clear. It can also be reasonable to authenticate a service or an appliance, e.g. a light switch client only has to authenticate itself to the light service because it does not matter, who switches the light on or off. This does not mean that no authentication is required. Only a light switch in the same room, where the light is, should be able to turn the light on or off, so the light switch has to authenticate itself as the light switch in the same room.

### 6.2.2 Authorization

Authorization in this scenario is built on top of the authentication process. Authorization means to grant an authenticated entity only restricted access to resources, whereby resources can be physical devices or logical permissions, e.g. access the TV screen or the permission to watch TV only weekdays between 3pm to 5pm.

## 6.3 General overview

The general overview covers two aspects. First, it introduces the modules which the architecture is built of. Second, it refers to the sections, where the modules are described in detail. The overview uses the top down approach, whereas the bottom up approach is used in the detailed section.

### 6.3.1 Common Jini schemes

The JoinManager does most of the administration work for a service. The pendant for a client is missing. Therefore, the section 6.4 provides classes that does the administrative work for a client: finding a service and validating if the service is still alive. These classes are independent from the security infrastructure and can be used in all Jini environments.

### 6.3.2 Client Transparency

Jini provides several ways how a service invocation can be done. The most common way is that a client requests a service proxy from the lookup service. Since the service proxy is a regular Jini object and the client knows at least one interface that the service proxy implements, the client could locally invoke methods on the service proxy. Then, the service proxy contacts the service backend using Remote Method Invocation. Client transparency means that there should be no change to existing clients which request service proxies. Therefore, the security infrastructure must be put into the service backend and the downloadable service proxy. Although no change is required at the client, a few security settings have to be done at client side, this is described in section 6.5.

### 6.3.3 Service

Since the clients contain no code of that architecture, all of it has to take place in the service proxy and the service backend. The detailed description of both parts can be found in section 6.11. If the service proxy is called the first time by the client, it has to start the authentication process. Therefore, the service proxy has to communicate with its service backend. Since the communication has to be secure in regard of confidentiality and integrity, a secure communication channel has to be established. More details of how a secure communication channel can be achieved in this environment, can be found in section 6.6. In general, a service proxy has to register itself on its service backend to establish a secure communication. These classes that build on the secure communication channel can be found in section 6.7.

### 6.3.4 Authentication

Summarizing the present facts: the clients regularly calls methods on the service proxy, the service proxy starts the authentication process by calling the service backend on a secure communication channel.

Now, the service backend can continue the authentication process by invoking the SubjectAuthenticator service. This is the central service which coordinates the whole authentication process at service backend side. The description of that service can be found in section 6.10.3.

#### Login Policies

Since several authentication schemes exist and clients can start the authentication process in different domains with different level of trust, it is a good way to specify a policy of how to log on independently of the service. Therefore, the SubjectAuthenticator service consults the LoginPolicyDB service which returns a login policy, see section 6.10.1.

#### Remote Callback

Depending on the login policy, the SubjectAuthenticator service carries out the authentication. In most cases, the SubjectAuthenticator needs user interaction, e.g. prompting for a user-name and password. Therefore, the service proxy has already installed a remote callback handler on client side which is a regular `Remote` object. More details about the remote callback handler can be found in section 6.9. The remote callback handler can perform different authentication schemes, e.g. user-name password dialog.

#### Java Ring Authentication

It is also possible to perform an authentication process using a challenge-response procedure on a Java Ring. Therefore, the remote callback handler has to invoke the RingAuthentication service which itself uses the BlueDot service which in turn does the communication with the Java Ring. Both services are regular Jini services which proxies are requested by a lookup service. The services are described more in detail in the sections 6.10.5 and 6.10.4.

#### Verification

In most cases, the data returned from client side has to be verified, e.g. the password or the response. Therefore, the SubjectAuthenticator consults the UserDB service which stores these informations and returns the information which are needed to complete the authentication process. More about the UserDB service in section 6.10.2.

### 6.3.5 Exchanging Data

The previous sections mentioned that services are invoked and that information are returned. Therefore, different classes are needed to exchange data, e.g. `User`, `Password`, `LoginPolicy` and so on. All these classes are described in section 6.8.

## 6.4 Common Classes

This section provides some useful interfaces which can be used in different contexts.

### 6.4.1 ServiceFinder

```
interface ServiceFinder extends Serializable, DiscoveryListener{
   Object getProxy();
}
```

is the most important interface here. A class that implements this interface is responsible that `getProxy()` returns a valid service proxy. If an already found service proxy is no longer valid, e.g. the service backend is down, it has to find another service proxy that matches the `ServiceTemplate` which can be given as an argument to the constructor. To test if a proxy is valid, the proxy should implement:

### 6.4.2 Pingable

```
interface Pingable extends Serializable{
   void ping();
}
```

so that `ServiceFinder` can do the check by calling the `ping()` method. For communication with the backend, the `RemotePingable` interface should be used. If no proxy is available, an `Exception` should be thrown instead of returning `null`. The semantic of `getProxy()` is that it always return a valid service proxy, otherwise the client could not regularly continue its program, so an exception handling must be started.

## 6.5 Client side

### 6.5.1 Transparency

One goal of this architecture is transparency for the client. A few things have to be done at client side, but this does not affect the client code. That means an existing client that uses an insecure service without authentication and authorization has not to be altered to use a service that needs authentication and authorization. Both services implement the same protocol. That means the extra work, which is needed to perform the authentication procedure, has to be done completely by the service and the infrastructure that is available at service side.

The Jini concepts provide the fundamentals to do this by splitting the service into the service proxy and the service backend. The service proxy can be seen as client code that the client does not know at compilation time, and it is linked dynamically during runtime. So every action needed at client side can be put into the service proxy, because it is executed by the client in its thread (figure 6.1).

### 6.5.2 Signed Code

A dangerous situation arises if the client downloads a malicious service proxy. That means, the service proxies contains malicious code, e.g. deleting files in the users home directory.

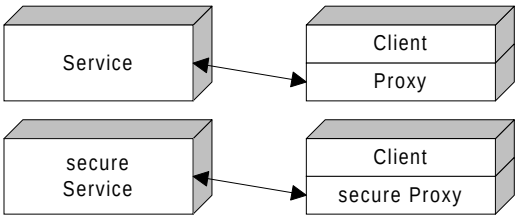


Figure 6.1: Securing a Service

Independently of this security architecture, every time code is downloaded, it is potential malicious. Therefore, only signed code from trusted sources should be executed.

Although the client code does not need to be changed, the user has to add trusted certificates to his key store. Then, he has to edit the policy file that only code that is signed by trusted entities could be executed.

This has always to be done independent from this architecture is used or not, since the download of the service proxy is the problem.

### 6.5.3 Summary

Summarizing the facts about the client side:

- transparency for the client that means no change to existing clients
- editing the policy file: granting only signed and trusted code permissions
- adding trusted certificates to the key store

## 6.6 Secure Communication

### 6.6.1 Goals

First of all, the architectural goals for implementing the authentication module should be given, so that the secure communication can be described with these goals in mind:

- an author of a new service which uses that architecture should interact as less as possible with the authentication module,
- the communication between the service proxy and the service backend should fulfill integrity and confidentiality.

### 6.6.2 Cryptography at Application Level

All these features, excluding transaction numbers, are provided by the Java Cryptography Extension (JCE). Currently, all the encryption is done on application level, so it is not absolutely transparent to the service, i.e. a minimal set of methods have to invoked to do the encryption and decryption of the sent and received data. In the next version of this architecture all the encryption work will be done on socket level by implementing an own Secure RMI Socket Factory, i.e. no special encryption and decryption methods have to be called, since on socket level all the data passed through are encrypted and decrypted implicitly (figure 6.2).

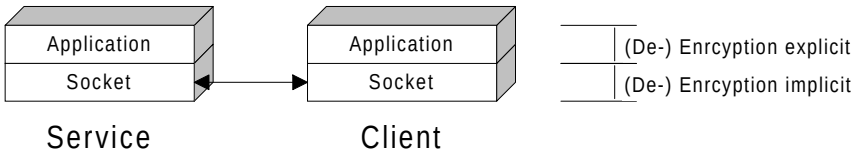


Figure 6.2: Cryptography at Application or Socket Level

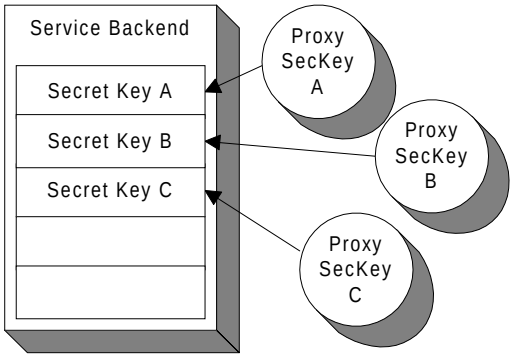


Figure 6.3: Management of Secret Keys

### 6.6.3 Leasing Context Information

Before the interfaces for the secure communication can be introduced, two data objects have to be introduced first. It is likely that one service backend serves more than one service proxy. That means in general, the service backend has to serve multiple service proxies. Therefore, it has to store context information. In this architecture, the service backend has at least to store the secret key used for decryption of incoming data and encryption of outgoing data, i.e. it uses implicitly memory to store data (figure 6.3). The Jini philosophy pretends that any resource should only be leased for a well-defined amount of time. Service proxies may crash or the network goes down, so that the service proxy has not the chance to logout and release its context information. It is only a matter of time, when all the memory is exhausted. Doing the right thing is to lease the context information. If the lease expires, the associated resource will be released.

#### Cookies and Tokens

Therefore, an object is needed to identify the resource to the grantor and an object that represents the right to access the resource. The first is defined as cookie and can be public, whereas the second is defined as token and must be kept secret. Both concepts are very general so they have a common super interface (figure 6.4):

```
interface CookieToken extends Serializable{
   boolean equals(Object);
   int hashCode();
   String toString();
}
```

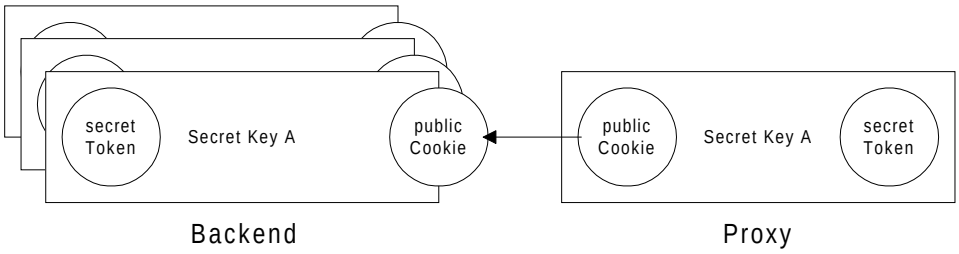


Figure 6.4: Management of Cookies and Tokens

All the methods are already defined by every Java object, because the overall super class is `Object` that provides these three methods. They are explicitly specified to mark their special meaning. Objects of this type should be comparable, e.g. the clients token can be compared to the reference token, they should implement a collision free hash code function, so that they can efficiently mapped, e.g. the cookie can be mapped to the leased resource. The `toString()` method should return a human readable representation. `Cookie` and `Token` themselves do not specify any further methods. They are only used for type safety, because the semantic is despite of the same interface different. Two instances should be different concerning the `equal()` method in space and time. The implementation of `Cookie` and `Token` can be the same, since both interfaces extend `CookieToken` and provide no further methods:

```
interface Cookie extends CookieToken{};
interface Token extends CookieToken{};
```

The difference is only made because of the different semantic. A cookie represents a resource, whereas the token represents the right to use the resource.

### 6.6.4 Cryptography at Socket Level in Future

In the first version of this architecture, the secure communication is handled at application level. In the next version, this should be done at socket level, using an own secure RMI Socket Factory. All the concepts shown in the following can be moved directly to this level. The benefit is the transparency of the secure communication. In this version it has to be done explicitly.

### 6.6.5 Secure Parameters and Return Values

Independent of who are the entities which communicate, methods have to be called remotely. That means one entity is calling the method and the other entity performs the execution of that method. In the following the entity who calls the method is defined as secure client. The entity who executes the method is called secure slot provider. The name for the first entity is clear, the second has been named that way, because in general a service has to serve several clients, so it has to store context information. They are logically stored in a slot. If a client is calling a method, the client has to specify in which slot its context information can be found, whereas the result of an execution does not need to specify a cookie on client side, because it is a $1 : n$ relationship. That means the return value of a secure method invocation is not that complex as the

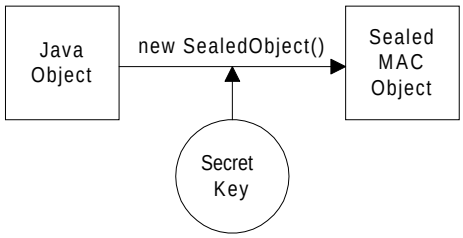


Figure 6.5: Sealing an Object

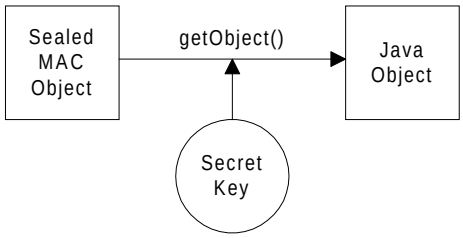


Figure 6.6: Unsealing an Object

arguments given to that method. First, it has to be defined what the interface of a secure return value is and then what the interface of a secure argument is.

**SealedMacObject**

```
interface SealedMacObject extends Serializable{
   Object getObject(SecretKey);
}
```

That means a return value has only to return another Java object by calling the `getObject()` method with a secret key as argument. The name `SealedMacObject` and the use of a secret key implies some aspects, what the implementation is expected to do. The object has to be sealed that means guarantee confidentiality by the use of symmetric encryption and guarantee integrity by the use use of a MAC. Both things have to be done transparently. At instantiation time, `SealedMacObject` has to compute the MAC and to seal the given object (figure 6.5). Whereas `getObject()` has to implement the reverse operation, i.e. it has to unseal the object and to verify the MAC (figure reffig:unsealing)). If the verification does not succeed, an exception has to be thrown. Thus, the semantic of `SealedMACObject` is close to `SealedObject`. In addition, it provides integrity.

**SealedCookieMacObject**

As mentioned above, a parameter has also to include a cookie, so that the secure slot provider can retrieve the context information with the cookie. Of course, the parameter has to guarantee confidentiality and integrity, too. Therefore, a secure parameter has only to extend `SealedMACObject`:

```
interface SealedCookieMacObject extends SealedMacObject{
```

```
   Cookie getCookie();
}
```

and to return the associated Cookie.

### 6.6.6 SecureClient

Next, it will be described how the secure communication can be encapsulated, using one object on client and one on secure slot provider side. The names for those both objects are straight forwarded: `SecureClient` and `SecureSlotProvider`. All the following steps can be found in figure 6.7. First, a look on the `SecureClient` interface:

#### Interface Definition

```
public interface SecureClient extends Serializable
   public final static int FIRSTTRANSNUMB = 0;
   public final static String KEYAGREEALG = "DH";
   public final static String SECRETALG = "DES";
   public final static String MACALG = "HmacMD5";
   public final static String KEYSTOREALG = "JKS";
   public final static String KEYSTORENAME = ".keystore";
   public final static String SIGALG = "SHA1withDSA";
   public static final byte skip1024ModulusBytes[] = {...};
   public static final BigInteger skip1024Modulus =
      new BigInteger(1, skip1024ModulusBytes);
   public static final BigInteger skip1024Base =
      BigInteger.valueOf(2);

   public final static int ACTIVE = 0;
   public final static int PASSIVE = 1;

   SecretKey getSecretKey();
   PublicKey keyGeneration(PublicKey);

   SealedCookieMacObject sealObject(Serializable);
   Object unsealObject(SealedMacObject);

   void setCookie(Cookie);
   void setToken(Token);
}
```

All the constants that are mentioned above, are defined here. This includes the public parameters for the DH algorithm: module and base, too.

#### Active and Passive Mode

`SecureClient` should work in different modes. The regular mode is active, because the client calls methods at the other side, but it is also imaginable that a one to one communication should be established. In that case no secure slot provider is needed

and one of the two parties have to be the passive part of the communication, whereas the other one is the active part.

**Settings in a 1:n Communication**

If the `SecureClient` is instantiated to secure the communication of a service proxy, the key generation cannot be done at instantiation time, because it is the only instantiation which would be available by everyone by regularly contacting the lookup service. Not until the service proxy has been deserialized by the client, the key generation could take place. It would be possible that the service backend put its digital signed public key for the DH key agreement into the service proxy. But then it has to re-register the service proxy every time its signature keys has been changed. Therefore, the most general case is to exchange the keys during runtime of the service proxy at client side. For that reason the `keyGeneration()` method has to use the public DH key given as argument to perform the DH key agreement completely on its side and has to store the secret key. The created own public DH key has to be returned as result. After the method has been executed, the secure client is ready to seal and unseal objects in actice mode. Cookie and Token - that the client has to specify, so that the secure service provider can retrieve the context information and can ensure that the client has the right to access the context information - could not be set, until the lease request has succeeded. For that reason `setCookie()` and `setToken()` are provided to set both after the lease request has succeeded.

**Settings in a 1:1 communication**

The second case is to use the `SecureClient` in an one to one communication, whereby the secret keys are already known on both sides, so that an implementation can be instantiated with the secret key and the mode. The `secretKey()` can be retrieved by an already established secure communication using the `getSecretKey()` method, but it is not restricted to that. Therefore, in that case the method `keyGeneration()`, `setCookie()` and `setToken()` are not needed, because in an one to one communication no context information are necessary.

**Replay**

Confidentiality and integrity are guaranteed by `SealedMacObject` and `SealedCookieMacObject`, but a third security issue has not been explained in detail: replays. Replays are an attack that does not effect integrity and confidentiality. That means the data is not changed, and the context will not be public readable. An active attacker gets the encrypted data on the insecure channel and replays it more than once to the receiver. If the service is idempotent, there must not be taken notion about that fact. But if the service is not idempotent something bad can happen, e.g. a credit transfer will be executed twice, so that one party is aggrieved and the other party that is likely direct or indirect the attacker, gets illegal an advantage.

A problem that is similar to this, is to delete some messages on the insecure channel, so that only some actions are taking place and some not.

To prevent this, transaction numbers must be used, so that a missing message or a replayed message can be recognized. The start transaction number is defined as an suggestion, like the cryptography algorithms, in the interface. There still remains one security hole with the transaction numbers, a message from an older session can be

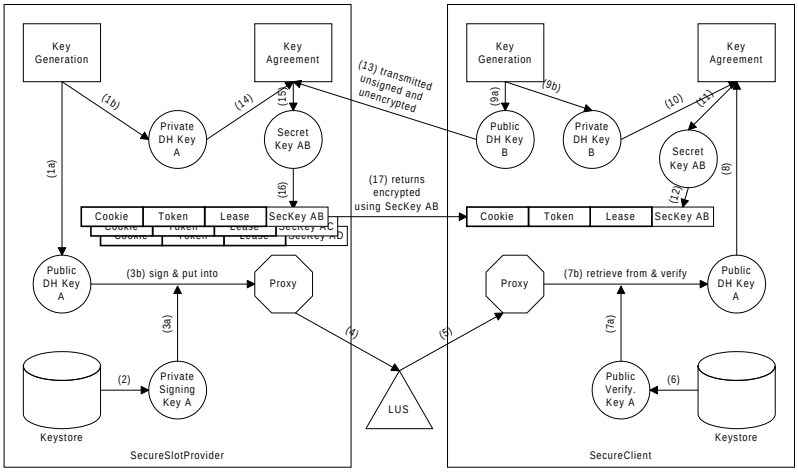


Figure 6.7: Key Agreement between Service and Client

replayed and the original could be deleted. To prevent this, the secret key has to be a session key that is valid for exactly one session in space and time.

A more general approach would be to specify only `Key` as argument, so that asymmetric operations would potentially be possible. Asymmetric keys are normally not used as session keys, because asymmetric encryption is very slow. That means in general, a symmetric session key is the most reasonable decision.

The `SecureClient` has to use `FIRSTTRANSNUMB` as first transaction number, so that the other side knows what to expect. In active mode the secure client has to increase the number after an object has been unsealed. In passive mode, the secure client has to increase the transaction number after sealing an object. In both modes, the transaction number has to be checked right after unsealing, and an exception has to be thrown if they do not correspond.

In addition, the transaction number and the token which must be sealed together with the token need a container object, because the sealing objects only accept one object to be sealed. For that reason, `TransactionData` and `TokenData` are defined:

```
interface TransactionData extends Serializable{
   Object getObject();
   int getTransactioNumber();
}

interface TokenData extends TransactionData{
   Token getToken();
}
```

### 6.6.7 Secure Slot Provider

The secure counterpart of a secure client is the secure slot provider which serves several secure clients. A common super class for a secure and not secure slot provider is `AbstractSlotProvider` which is explained next. There are also some data containers which are needed by the a slot provider.

**AbstractSlotProvider**

```
interface AbstractSlotProvider extends Landlord{
   Object getData(Cookie);
   void setData(Cookie, Object);
}
```

Every resource has to be leased, so `AbstractSlotProvider` has to extend `Landlord`. Another interface is also imaginable, because Landlord does not belong to the core Jini classes, but the functionality of the `Landlord` interface fits, so extending it, is a good choice.

Given a `Cookie`, the `getData()` method returns the associated `Object`, whereas the `setData()` method stores an `Object` in the slot which is identified by a `Cookie`.

**SlotRes**

But an object not only has to be stored in the slot and not all of the data stored in the slot could be given to the client, so that two interfaces are needed which data is stored for the client and which extra data is needed to maintain the slot:

```
interface SlotRes extends Serializable{
   Cookie getCookie();
   Lease getLease();
   Token getToken();
}
```

As mentioned above, `Cookie` and `Token` are needed by the secure client to specify the slot and the right to access it. The secure client also needs the `Lease` for canceling or renewing it.

**SlotData**

```
interface SlotData extends SlotRes{
   Object getData();
   void setData(Object);

   long getExpiration();
   void setExpiration(long);

   int hashcode();
}
```

The effective data can be accessed by `getData()` and `setData()`. Since every `Lease` expires, the expiration date has to be stored. It is also be stored in the `Lease`, but the data in the Lease can be altered, so the expiration date management can be done by calling `getExpiration()` and `setExpiration()` for a slot. For reasons of efficient storage, `SlotData` should return a collision free hash code by calling the method `hashcode()`.

### SecureSlotData

These both data structures are very general. For a secure slot provider, the `SlotData` interface has to be extended a second time:

```
interface SecureSlotData extends SlotData{
   SecretKey getSecretKey();
   int getTransactionNumber();
   void setTransactionNumber(int);
}
```

Consequently, the extension handles the storage of the secret key and the transaction number that is needed for a secure communication.

### Interface definition

On that basis the secure slot provider can be introduced at last:

```
interface SecureSlotProvider extends AbstractSlotProvider{
   PublicKey getPublicKey();
   SealedMacObject sealObject(Cookie,Serializable);
   Object unsealObject(SealedCookieMacObject);
   Object unsealFirst(Cookie, SealedCookieMacObject);
   SlotRes leaseSlot(PublicKey, SealedCookieMacObject);
   SealedMacObject leaseEncryptedSlot(PublicKey,
      SealedCookieMacObject);
}
```

Since a `SecureSlotProvider` is normally used in a service backend, it has to provide the public DH key which is needed for the key agreement algorithm. That means the `SecureSlotProvider` has to create at instantiation time the private and public DH key. The public DH key can be retrieved by calling `getPublicKey()`.

To seal an object, it has to use the transaction number and the secret key which can be found using the `Cookie`. The serializable object, which is given to `sealObject()`, has to be sealed together with the transaction number which has to be increased afterwards.

Vice versa: to unseal a `SealedCookieMacObject`, which contains the cookie to get the context information, `unsealObject()` has to be used. It has to check three facts:

- the transaction number
- the token
- expiration of the lease

If one of the checks fail, an exception has to be thrown. Otherwise the unsealed object has to be returned.

To lease a slot, the client sends a `SealedCookieMacObject`, without a cookie, since the cookie does not exist yet. So the `unsealObject()` method would fail. For that reason `unsealFirst()` gets the cookie of the new created slot and does not perform any security checks.

The most important method of that interface is `leaseSlot()`. It has to do several things:

- complete the key generation, with the given public DH key,
- store the secret key in the slot
- store the data inside the `SealedCookieMacObject` in the slot
- generate a token and store it in the slot
- generate a cookie and store it in the slot
- generate a lease and store it in the slot
- store the expiration of the lease in the slot.
- return the token, cookie, lease as a `SlotRes`

A combination of leasing a slot and the encryption of the returned `SlotRes` has to be done by the method `leaseEncryptedSlot()`.

### 6.6.8 Summary

`SecureClient` and `SecureSlotProvider` together provide a secure communication over an insecure channel. All the security checks and used data structures are transparent to the tier which builds on this tier. Currently, they only have to call the methods `sealObject()` and `unsealObject()`. In the next version of this architecture, i.e. if all the security is done on socket level, `sealObject()` and `unsealObject()` are implicitly called, so that the secure communication is totally transparent to the tier above.

## 6.7 Common Secure Classes

Common secure classes provide a secure communication channel and can be used in multiple domains. They use the `SecureClient` and the `SecureSlotProvider` to establish a secure communication link.

### 6.7.1 Refreshable

The Java permissions are granted in a policy file which can be altered during the runtime of a program that read the policy file at start time. If the changes should take effect, there must be a possibility to update the state by rereading the file or do another appropriate action. Therefore, an interface has to be introduced which provides a method that can do the update:

```
interface Refreshable extends Serializable{
   void refresh();
}

interface RemoteRefreshable extends Remote{
   void refresh(SealedCookieMacObject);
}
```

The first interface can be implemented by a service proxy or a regular local object, whereas the second interface is intended to be implemented by a service backend. In that case the service proxy has to do encryption of the parameters and calls the service backend method.

Since `refresh()` has no parameters, `null` has to be used as parameter. It is important to do it that way, because there are many things done transparently, e.g. sending the cookie and the token.

### 6.7.2 Register

Very often a service uses other services as client to perform its task. To do so, the client should register and unregister itself, so that the service can create or remove context information about the client. In this architecture, the context information at least consists of the secret key, the token among other things. The Jini philosophy pretends that all resources should be leased. Context information are resources, because they use memory which can be exhausted. To register and unregister only two methods are needed:

```
interface Register extends Serializable{
   void register();
   void unregister();
}

interface RemoteRegister extends Remote{
   SealedMacObject register(PublicKey);
   void unregister(SealedCookieMac);
}
```

The first interface is intended to be implemented by a service proxy. The two methods have to call the methods of the second interface that has to be implemented by a service backend. On proxy side, both methods have no arguments, so they have to retrieve the information by themselves to call the associated methods of the backend.

The `PublicKey` is the public DH key which is needed by the service backend to complete the DH key agreement algorithm, therefore the proxy has to do the DH key agreement first. A public key from and signed by the service backend has already been put into the proxy. Since the verification of the signed code is done by the JVM, only the dynamic data has to be checked explicitly.

#### SignedRegisterData

The `PublicKey` and the `DomainName` which is used to lookup the public verification key in the key store are bundled in one interface:

```
interface SignedRegisterData extends Serializable{
   DomainName getDomainName();
   PublicKey getPublicKey();
}
```

An implementation of it can be put into a `SignedObject` which also contains the signature of the `SignedRegisterData`.

**Register Methods**

The proxy has to call `getDomainName()` to retrieve the `DomainName` which it uses as the alias to lookup the public verification key in the key store. This verification key can be used to verify the signature of the `SignedRegisterData` that can be stored together in a `SignedObject`. Only if the verification succeeds, the public DH key can be used to perform the DH key agreement in the proxy, otherwise an exception should be thrown.

The verification has to be done if `register()` has been called on the proxy. After the verification has been succeeded, the proxy generates its own pair of the public and private DH keys to perform the DH key agreement at its side. Using its public key, it has to call the `register()` method of the service backend. The service backend does not need to perform a integrity check, because every interested party is allowed to connect to the service backend.

The `register()` method on this side also has to do the DH key agreement with the given public DH key of the proxy and its own stored private DH key. The context information has to be stored in a secure slot provider. The return value of the leasing request can be returned directly to the client. It is a `SealedMacObject` containing a `SlotRes` which includes the cookie, token and the lease which are used by the proxy, to identify the slot, the right to access the slot and to renew or cancel the lease, whereas the renewing of leases can easily be done by a `LeaseRenewalManager`.

The `unregister()` method of the proxy only has to call the `unregister()` method of the service backend using `null` as argument, since no arguments are required, but the underlaying sealing tier is expecting an argument. The `unregister()` method on backend side should release the resource that stores the context information. If the service proxy does not, or maybe could not call `unregister()`, this has to be done implicitly if the lease expires. That means calling the `unregister()` method and the expiration of the associated lease are equivalent.

## 6.8 Objects for Authentication

This section describes the objects which have to be exchanged between client side and service side to exchange authentication data.

### 6.8.1 User

The goal is to log on a user. That means the user has to be defined by an interface:

```
interface User extends Serializable{
   boolean equals (Object);
   int hashcode();
   String toString();
}
```

Therefore, a `User` has the same interface definition like a `CookieToken`, but another semantic is associated. The `equals()` method is used to compare two users. Only if the other object represents the same user, `equals()` has to return true, otherwise false. For an efficient storage of users, `hashcode()` should return a collision free hash code. The `toString()` method should only be used for a human readable representation of the object. All comparisons should be done via `equals()` and `hashcode()`.

### 6.8.2 Password

A common login procedure is to use a user-name password dialog. Therefore, the password has to be modeled using an interface, too:

```
interface Password extends Serializable{
   boolean equals (Object);
   int hashcode();
   String toString();
   void clearPassword();
   char[] getPassword();
}
```

The first three methods: `equals()`, `hashcode()` and `toString()` have to be implemented the same way described above. Since a password is very sensitive, the representation in the memory should be cleared by calling `clearPassword()`.

Currently, the method `getPassword()` is only provided to support the JAAS `PasswordCallback` class. The next version should use an own `PasswordCallback`, so that `getPassword()` is needed no longer. The role of `getPassword()` is to return the password as a char array representation, which is also the recommended implementation, because it is very simple to clear a char array.

### 6.8.3 Challenge-Response

Another common login procedure is to use a challenge-response procedure. Therefore the challenge and the response has to be introduced as interface definition:

```
interface Challenge extends Serializable{
   boolean equals (Object);
   int hashcode();
   String toString();
   byte[] toByteArray();
}

interface Response extends Serializable{
   boolean equals (Object);
   int hashcode();
   String toString();
   byte[] toByteArray();
}
```

The methods `equals()`, `hashcode()` and `toString()` have to be implemented as `User` suggests it.

The method `toByteArray()` has to return the byte array representation of the `Challenge` or the response. This method is needed by the Java Card Api that only allows to exchange `byte[]`. That means executing a challenge-response procedure on a Java Card enabled device demands the `Challenge` in byte array representation and the response has to be retrieved from a byte array.

### 6.8.4 DomainName

Assuming that one Java Card enabled device should be used in different domains, whereby every domain requires another variant of the challenge-response procedure, the domains and therefore the different challenge-response procedures have to be distinguished. This leads to the following interface:

```
interface DomainName extends Serializable{
   boolean equals (Object);
   int hashcode();
   String toString();
   byte[] toByteArray();
}
```

The semantic of all four methods is the same as explained above. Notice: `DomainName` does not deal with DNS or other naming schemes. It only is the representation of different login domains. It is up to every domain to choose its `DomainName` representation. The string representation of the `DomainName` is also used to determine the alias name on client side to retrieve the associated certificate and the public verifying key in it. Two other uses of the `DomainName` will be shown later on.

### 6.8.5 DefaultUser

This concept is very crucial to the architecture. One aspect to determine different login schemes can depend on where the client is executed. A possible procedure could be to use the IP numbers of the client host. In general, it is not a good decision to use IP numbers for the purpose of authentication, because there exists several ways to fake the IP number.

Therefore, the concept of a default user is utilized, which provides this feature among other things, without extending the architecture with a new concept. A default user does not represent a real life person, but the location from where the users log in.

#### Example

At first, default user sounds a bit confusing, because a client host is meant and not a user, as the name suggests. But default user is not only used to determine the location, where a user wants to log on. It is also used that a user can log on anonymously, so that no authentication is needed. This also sounds a little bit confusing, but an example can easily motivate the use: the TV service in a child's bedroom, can be turned on between 2pm and 5pm without any authentication if the remote control in this room is used. After 5pm only persons, who have the permission to watch TV, e.g. the parents, are allowed to use the TV service. In the last case authentication is needed. That means between 2pm and 5pm the remote control logs on as a default user to the TV service. That means the TV service does not know, who is watching TV, it only knows that someone is watching TV, who used the remote control in the child's bedroom. That means the remote control is the default, because no other user information is needed.

#### Response Time

Another benefit can be the response time. For every log on request a session key has to be generated that currently takes 3 seconds on a Sun Ultra 5. This respond time is too

long, e.g. if the light should be turned on. So the light switch logs on the light service as default user. Only at the first time, the session key has to be calculated. Afterwards, everyone can turn on or turn off the light as default user of that light switch service without any delay.

#### Interface Definition

The interface of the `DefaultUser` is very simple because it builds on the existing concepts:

```
interface DefaultUser extends User{
}
```

That means a `DefaultUser` is a normal user and the interface is only used for type safety.

`DefaultUser` fulfills two purposes: it defines the location, from where the user logs on and can be used if the user can be anonymous, because the decision only depends from where the users wants to use the service. The location is not a physical, but a logical location. That means on one client host with a multi user environment, there can be one account only for a default user. All other accounts have no default user. For example: the VCR service should be used at the office without authentication, because if the user has logged on his account, he has already been authenticated. Therefore, the user can put a default user in his home directory which will be used to log on the VCR service, whereas the colleagues, which can also log on the same host, have their own home directory on that client, but they do not have the default user.

The use of the default user is not obligatory. It is up to the login policy if the default user should be used, and which permissions are granted to a default user. In general, if a client host cannot provide the default user, the most restrictive policy should be used.

#### Handling the DefaultUser

Currently, there is no notion how the default user can be retrieved on client side. The `DomainName` has to be used in that situation, too. A client host account can be used to log on different domains, so the `DomainName` is a parameter to retrieve the correct default user. The authentication of the default user can be done by a challenge-response procedure. How this works is up to the implementation, e.g. the default user is serialized and written to

```
file:/${user.home}/.jiniauth/{domainName}/defaultuser.ser.
```

Also the object needed for the challenge can be written to:

```
file:/${user.home}/.jiniauth/{domainName}/challresp.ser.
```

## 6.9 Remote Callback

In general, to authenticate a user, the service backend, i.e. the login infrastructure has to request data from the user, e.g. user-name and password. So, there must be a communication back to client side.

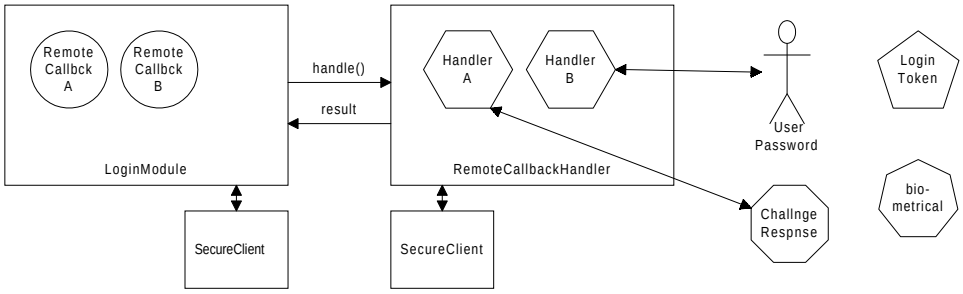


Figure 6.8: Remote Callback Handler

### 6.9.1 Interface Definition

Therefore, a remote callback handler is used:

```
interface RemoteCallbackHandler extends Remote{
   SealedCookieMacObject handle(SealedCookieMacObject)
}
```

The `RemoteCallbackHandler` has to be a `Remote` object, since the `handle()` method is called remotely from service side. Therefore, the communication between the service side and the `RemoteCallbackhandler` has to be secured using the `SecureClient` on both sides.

The remote callback is a one to one communication, whereas the service side is the active part, because it is calling the method and the `RemoteCallbackHandler` is the passive part, because it acts as a service, but with no context information.

Return value as well as parameter are `SealedCookieMacObject`, although no cookies are required. This has to be done, because two `SecureClient`s communicate. In both cases the cookie argument should set to null, resp. it should be ignored.

This also means that no key generation is needed, because a secret key has already been created that can be used for the remote callback communication, too (figure 6.8).

### 6.9.2 RemoteCallbackHandlerParam

The object behind the sealed parameter is:

```
interface RemoteCallbackHandlerParam extends Serializable{
   RemoteCallback[] getRemoteCallbacks();
}
```

### 6.9.3 RemoteCallback

Thus, `RemoteCallbackHandlerParam` only is a container for an array of `RemoteCallback`s:

```
interface RemoteCallback extends Callback, Serializable{
}
```

Since `RemoteCallback` defines no methods, it is only used as marker, like `Callback`, an interface of JAAS, is used. The extension of `Callback` is not obligatory. It should express that a `RemoteCallback` is a special kind of `Callback`. If the JAAS package is not used, `Callback` can be removed silently.

Every authentication technology which needs remote callbacks must extend or implement the `RemoteCallback` interface.

### 6.9.4 Common Callbacks

A few common remote callbacks are already defined in this architecture.

#### user-name Password Callback

To perform a user-name password dialog, three interfaces are provided:

```
interface RemoteNameCallback extends RemoteCallback{
   String getPrompt();
}

interface RemotePasswordCallback extends RemoteCallback{
   String getPrompt();
   boolean isEchoOn();
}

interface RemoteNamePasswordCallback extends RemoteCallback{
   RemoteNameCallback getRemoteNameCallback();
   RemotePasswordCallback getPasswordCallback();
}
```

If a `RemoteCallbackHandler` has to handle a `RemoteNameCallback`, it has to print the prompt and it has to return a `User` object.

If a `RemotePasswordCallback` is called, it has to print the prompt and it has to display the entered password in clear or not according to `isEchoOn()`. The return value is a `Password`, whereas the password has to be cleared on client side.

The `RemoteCallbackHandler` can use the console to prompt for user-name and password. It can also use a GUI. A better approach is to use a GUI which resides on client side and can be retrieved, e.g.:

```
file:/${user.home}/.jiniauth/userpassdialog.ser.
```

There are several benefits using that approach: every appliance knows for itself which GUI fits perfectly and firms can fit out their appliance, e.g. room panels, with their corporate identity styled dialogs.

In general, the return value of the `RemoteCallbackHandler` has to be a sealed `Object` array that corresponds to the `RemoteCallback` array.

For reasons of performance, the both remote callbacks are bundled to a `RemoteNamePasswordCallback` which should return an `AuthData` object that extends `UserData`:

```
interface UserData extends Serializable{
   User getUser();
}

interface AuthData extends UserData{
   Object getSecretData();
}
```

The semantic of both interfaces is straight forward. `UserData` is a container for an `User` which can be retrieved by calling `getUser()`.

`AuthData` extends the container by adding a data object which should be handled secretly. This object can be retrieved by calling `getSecretData()`. In this case, it is a `Password`.

#### Challenge-Response Callback

This architecture also provides built-in support for challenge-response procedures. Therefore it defines:

```
interface RemoteChallengeCallback extends RemoteCallback{
   Challenge getChallenge();
}
```

If the `RemoteCallbackHandler` receives this callback, it should start a challenge-response procedure and it should return its result: the `Response`.

In general, there are two cases, a challenge-response procedure is needed. As mentioned above to check if the `DefaultUser` can authenticate himself and in combination with the Java Card API. To determine which challenge-response procedure is meant, `RemoteChallengeCallback` should be extended by marker interfaces.

To check the authentication of the `DefaultUser`, an object that contains the key needed for the challenge-response procedure, can be deserialized and therefore, a procedure can be executed.

If a challenge-response procedure is desired, e.g. a challenge-response Jini service can be contacted which itself contacts another Jini services which provide the Java Card API of a smart card or a Java Ring.

The return value of a challenge-response procedure is a `ChallengeData`:

```
interface ChallengeData extends UserData{
   Challenge getChallenge();
   Response getResponse();
}
```

It extends `UserData` and provides the `Challenge` and the `Response`, whereas the `Response` only has to be set, the `Challenge` is set later on by another entity, otherwise an old challenge and response could be used.

#### Login Token Callback

The `RemoteCallbackHandler` can also be used to put or retrieve a login token on client side. That can also be stored together with a cookie and a lease as `SlotRes` as serialized object in the

```
file:/${user.home}/.jiniauth/{domainname}.
```

This is useful for multiple logins with only one authentication, e.g. the login token is valid for ten minutes after the user has authenticated himself correctly. Then, the user calls another services in the same domain which - depending on the policy - can check if at client side exists a valid login token. If a valid login token exists and the login policy of the first service implies the login policy of the second service, the user will be logged on without further interaction.

### 6.9.5 Remarks

The given concept is extremely flexible, so that any possible authentication scheme can be modeled with it. A big benefit is that the login policy at service side decides which of the given techniques should be used. That means a malicious user cannot get more rights by faking the system at client side.

It is also imaginable that biometric authentication schemes will be used by implementing new `RemoteCallback`s.

## 6.10 Infrastructure Services

The architecture enfolds several Jini services which have already been mentioned. A service named `SubjectAuthenticator` does the login procedure by contacting some other services:

- `LoginPolicy` service which decides, how to logon
- `RemoteCallbackHandler` which returns the authentication data
- `UserDB` service which verifies the returned data

In addition, to support Java Ring authentication by processing a challenge-response procedure, two more services are necessary:

- `BlueDot` service which provides the functionality of a Blue Dot Receptor (the Java Ring reader)
- `RingAuthentication` which contacts the BlueDot service, to run a challenge-response procedure

The `UserDB` service, the `LoginPolicy` service and the `SubjectAuthenticator` service could also be combined in one service. But all three services are very modular and provide a well-defined API, with a well-defined semantic, so it is good design decision, to separate them.

Next, all services are described in depth.

### 6.10.1 Login Policy DB Service

The general task of a login service is to determine depending on one or more parameters, what login policy should be used to log on a user.

#### Interface Definition

So, the interface is very simple:

```
interface LoginPolicyDB extends Refreshable, Register, Pingable{
   public final static int FIRSTATTEMPT = 1;
   LoginPolicy getLoginPolicy(LoginData);
}
```

The interfaces `Refreshable`, `Register` and `Pingable` have already been introduced. Another service has to register with the `LoginPolicy` to establish a secure communication link. `Refreshable` is used to refresh the `LoginPolicy`s state and `Pingable` is used to determine if the LoginPolicyDB service is still alive.

The intrinsic service only has one method: `getLoginPolicy()` which gets a `LoginData` to determine which `LoginPolicy` should be returned. The general approach is that this method encrypts the data, calls the backend service and decrypt its result, but first a look at the `LoginData`.

**LoginData**

```
interface LoginData extends Serializable{
   int getAttempt();
   String getService();
   DefaultUser getDefaultUser();
   ...
}
```

The `LoginPolicyDB` service only uses the given three methods of `LoginData`, the remaining are used by other services and therefore described, when needed (figure 6.9).

**Login Policy Parameters**

Login policies therefore depend on three parameters.

As explained above, the `DefaultUser` can be interpreted logically as the client location, or the identity of the client host. This information can be retrieved by calling `getDefaultUser()`.

The second parameter is the service used. Its `String` representation is returned by `getService()`. The naming scheme is an implementation fact, but a good and simple approach is to use the `getClass().getName()` method of the service to retrieve the class name.

The use of counting the attempts should be motivated first. For example the TV service in the childs bedroom: the first attempt is to log on as default user, because between 3pm and 5pm exits no access restriction. That means after 5pm the default user could not log on. Therefore, a second attempt has to be started, e.g. to log on a real user using a user-name password dialog.

It is only up to the login policy service which login policy should be used by a given combination of attempt, service and `DefaultUser`. It can be very restrictive, for example only Java Ring authentication is allowed, or it can be more user friendly: log on as `DefaultUser` or to log on with a existing login token.

**LoginPolicy**

The `LoginPolicy` which is returned contains only two methods:

```
interface LoginPolicyDB extends Serializable{
   String getName();
   boolean implies(LoginPolicy);
}
```

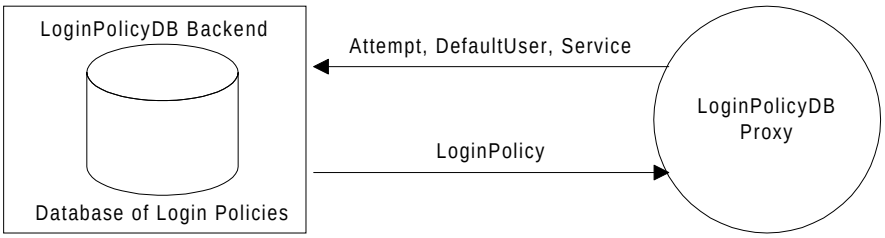


Figure 6.9: Login Policy DB Service

The `LoginPolicy` does not return a login policy, but the name of the login policy which can be used to lookup the authentication technology, e.g. in a database.

The `implies()` method is very important for a login token authentication, otherwise the login policy strength could be weaken by services which need a weak authentication and give out the login token. A service that needs stronger authentication, but also accepts login tokens, could be used, without the strong authentication.

A login policy A should imply login policy B if the authentication strength of the login policy A is stronger than or equal to the login policy B, whereas no semantic of the strength is given in this architecture. It is up to the login policy administrator to decide about it. The `imply()` method should return true at least if the same login policy is given. The implementation also has to ensure that no endless recursion occurs.

#### First attempt

The interface also defines a constant that has to be used to determine the number of the first attempt. Two intuitive possibilities are to use one or zero. This architecture defines one that corresponds to ”first attempt”.

#### Remote Interface

If the `LoginPolicyDB` is implemented as Jini service or at least as RMI Server Object, the `LoginPolicyDB` is a service proxy and has to do only the sealing and unsealing of the data and it calls the appropriate method of the backend, whereby the backend has to do the work described above.

```
interface RemoteLoginPolicyDB extends RemoteRegister,
   RemoteRefreshable, RemotePingable{
   SealedMacObject getLoginPolicy(SealedCookieMacObject);
}
```

### 6.10.2 User DB Service

The task of the User DB service is to verify the results of the `RemoteCallbackHandler`. That means, it has to authenticate the user by the given extra data, e.g. password or challenge and response.

#### Interface Definition

Therefore, it currently provides two methods:

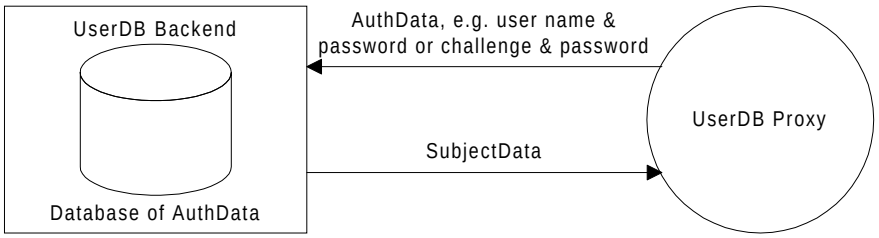


Figure 6.10: User DB Service

```
interface UserDB extends Refreshable, Register, Pingable{
   SubjectData authenticate(AuthData);
   SubjectData authenticate(ChallengeData);
}
```

As explained above, `AuthData` could be the result of a user-name password dialog and `ChallengeData` is the result of a challenge-response procedure.

In general, `AuthData` could be used in different ways, e.g. it could contain an audio file instead of the password, to verify the user by a voice recognition system. Therefore, these both methods span a wide range of possible authentication methods for verifying.

The semantic of `Refreshable`, `Register` and `Pingable` which `UserDB` extends is the same as in `LonginPolicyDB`.

Both `authenticate()` methods should check if the user has authenticated himself correctly. In that case, a `SubjectData` should be returned which contains all the information to build a `Subject` (figure 6.10). If the authentication fails, an exception should be thrown. The semantic is that in general, only authenticatable users try to log on and they give the right information to log on. Every other case is an exception and has to be handled like an exception.

#### SubjectAuthResult

`SubjectAuthResult` is defined as:

```
interface SubjectAuthResult extends Serializable{
   Set getPrincipals();
   Set getPrivateCredentials();
   Set getPublicCredentials();
}
```

The sets of `Principal`s, private and public credentials have already been explained above. They are used to build a `Subject` object, on which the authorization decisions can be made.

#### Remote Interface

The most common way is that these two authentication methods do the sealing and unsealing and that they call the method of a backend which implements the following interface:

```
interface RemoteUserDB extends RemoteRegister,
   RemoteRefreshable, RemotePingable{
   SealedMacObject authenticate(SealedCookieMacObject);
}
```

Since the data has to be sealed, the backend only has to provide one method which has to dispatch accordingly to the unsealed data which authentication verification has to be performed.

### 6.10.3 SubjectAuthenticator Service

This is the central entity of the whole architecture. It contacts the `LoginPolicyDB`, the `RemoteCallbackHandler` and the `UserDB` to handle the authentication process.

#### Interface Definitions

```
interface SubjectAuthenticator extends Register, Refreshable,
   Pingable{
   SubjectAuthResult login(LoginData, RemoteCallbackHandler);
   void logout(SlotRes);
}
```

The `RemoteSubjectAuthenticator` is defined accordingly as:

```
interface RemoteSubjectAuthenticator{
   SealedMacObject login(SealedCookieMacObject,
   RemoteCallbackhandler);
   void logout(SealedCookieMacObject);
}
```

The `SubjectAuthenticator` interface has to be implemented by a service proxy and the `RemoteSubjectAuthenticator` must be implemented by a service backend.

Again, the service proxy has to do the sealing and unsealing of the data. The `RemoteCallbackHandler` which implements `RemoteRef` cannot be encrypted, because the serialization and deserialization process, which has to be done by the encryption, resp. decryption would lead to a wrong reference, so that the RemoteCallbackHandler could not be contacted. Since the communication between the `SubjectAuthenticator` and the `RemoteCallbackHandler` is secure, no security problems arises.

The `login()` method gets the `LoginData` and a `RemoteCallbackHandler` and has to return a `SubjectAuthResult`. First, it has to contact the `LoginPolicyDB` using the `LoginData` as parameter. Depending on the `LoginPolicy`, it has to invoke the appropriate `RemoteCallbacks`, resp. check if the given login token is valid and implies the `LoginPolicy` (figure 6.11).

#### LoginData

```
interface LoginData extends Serializable{
   ...
   SecretKey getSecretKey();
```

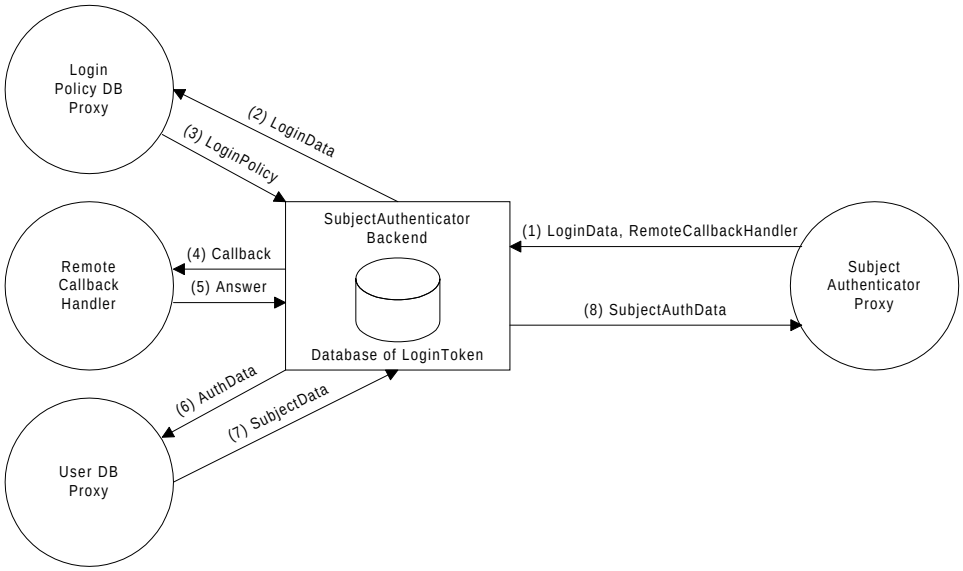


Figure 6.11: Subject Authenticator Service

```
   SlotRes getSlotRes();
}
```

Therefore, two other methods of `LoginData` are needed. To secure the communication with the `RemoteCallbackhandler`, the `getSecretKey()` method retrieves the secret key used to encrypt and decrypt the data. If the `LoginPolicy` specifies that a login token could be used, `getSlotRes()` returns the slot result that contains the cookie and the token that identifies the slot and the right of the user to retrieve the information. Therefore, after a successful authentication, the `SubjectAuthenticator` has to store the data needed in a separate slot provider. Since the data stored in that slot need no extra secure communication, because the data is sent to and received from secure channels, a non secure slot provider is the right choice:

**SlotProvider**

```
interface SlotProvider extends AbstractSlotProvider{
   leaseSlot(Object);
}
```

The only method has to do a few things:

- generate `Token`, `Cookie` and `Lease`
- store `Token`, `Cookie`, `Lease`, expiration time and the given object in the slot, e.g. using a `SlotData`
- return a `SlotRes`, containing the `Token`, `Cookie` and the `Lease`

#### JassSlotResult

In that case, the data that has to be stored, is well-defined:

```
interface JaasSlotResult extends Serializable{
   LoginPolicy getLoginPolicy();
   Subject getSubject();
}
```

First, the `LoginPolicy` itself has to be stored, therewith the implication check can be done. If the check succeeds, the stored `Subject` can be used to built the return value of the `login()` method.

If no login token exists, the login token is not valid or it should not be used, the return of the `RemoteCallbackHandler` must be checked by calling the `UserDB authenticate()` methods. They return a `SubjectData` object, which is used to build the Subject object, as mentioned above. This `Subject` and the corresponding `LoginPolicy` should be stored in a non secure slot provider.

#### SubjectAuthResult

Both, the `Subject` and the `SlotRes`, which is returned from the slot provider, are returned to the caller as a `SubjectAuthResult`:

```
interface SubjectAuthResult extends Serializable{
   SlotRes getSlotResult();
   Subject getSubject();
}
```

### 6.10.4 BlueDotService

The `BlueDot` service is named that way, because the Java Ring reader is called Blue Dot Receptor“. Its task is to dispatch multiple request for the Blue Dot Receptor and start the communication that is needed from the outside to communicate with the Java Ring, resp. the Java Card API.

#### Java Card API

First a few words about the Java Card API and the classes provided by Dallas Semiconductor - the manufacturer of the Java Ring - to access the Ring.

On smart cards or on Java Rings that implement the Java Card API, runs a reduced JVM that only contains the following classes:

- Applet, AID
- APDU
- ISO
- Util
- several PIN classes
- several Exceptions classes

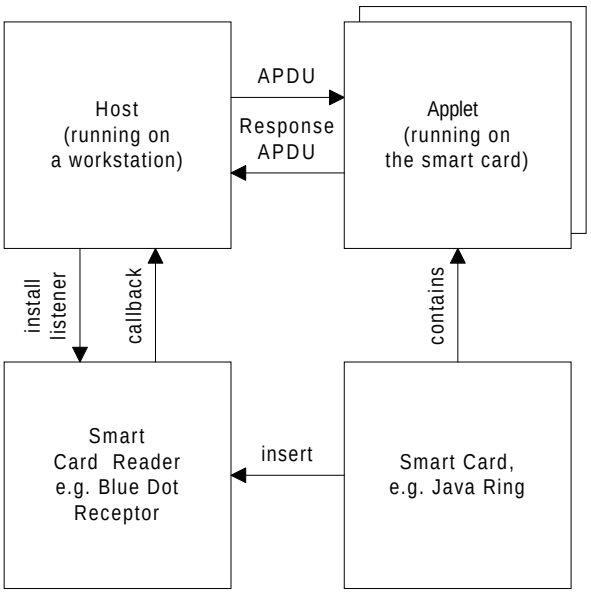


Figure 6.12: Java Card Interaction

- eventually crypto classes, e.g. `DES3_Key`
- eventually several File classes

Even, a very common class `String` is missing in that listing. The basic principle is that the Java Card API executes extended `Applet`s, whereas the communication between an outside application and an `Applet` is achieved by using an `APDU` object which can only exchange byte arrays. That means it is not possible to exchange Java objects. But even if it would be allowed, it could not work, because the Java Card API provides no class loader.

`Applet`s have to be loaded on the smart card or the Java Ring via a proprietary protocol. If they are on the ring, they run in a passive mode until they are deleted. The applets are waiting for an incoming request. Then they process the request and return the result, whereas request and result are byte arrays encapsulated in an `APDU` (figure 6.12).

The class `ISO` contains some constants and `Util` some methods for byte array copying. The PIN classes are used to protect the smart card or an application.

Two extension packages provide cryptography support which the Java Ring includes. So the Java Ring can be used to sign data, calculate hash codes and it can asymmetrically and symmetrically decrypt and encrypt data.

Another extension package, which is not included in the Java Ring, defines several messages for `File` handling.

But the Java Ring contains some proprietary extensions by Dallas Semiconductors: `Clock` and `Coproccesor` that provide access to a clock and fast computation of asymmetric operations.

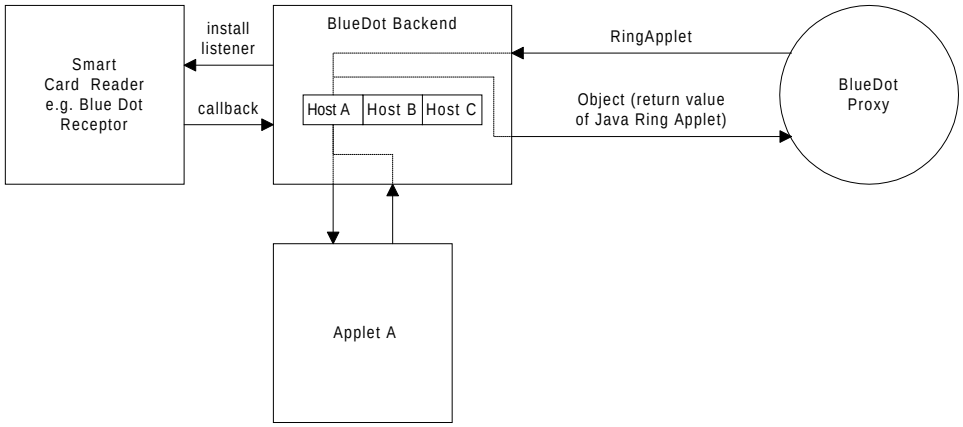


Figure 6.13: Blue Dot Service

**Interface Definitions**

Now, the `BlueDot` service has to run outside `RingApplet`s that select the `Applet` needed on the Java Ring and return the result of the `Applet` as `Object` (figure 6.13):

```
interface BlueDot extends Pingable{
   Object runApplet(RingApplet);
}

interface RemoteBlueDot extends RemotePingable{
   Object runApplet(RingApplet);
}

interface RingApplet extends Serializable{
   RingData getRingData();
   Object getData();
   void run(JibMultiFactory, String);
}
```

The `RingApplet` provides three methods which are called by the `BlueDot` implementation.

First, if a Java Ring is inserted, `getRingData()` has to be called to retrieve the information about the applet and how it can be selected. Then, the `run()` method is called using the proprietary `JibMultiFactory` which provides the connection to the Java Ring and at least `getData()` has to return the data which is the result of the execution of the `Applet`. The String argument should be printed out after the execution has been completed.

At last, the `RingData` interface:

```
interface RingData extends Serializable{
   String getAppletName();
   String getMessage();
   String getPassword();
   URL getURL();
```

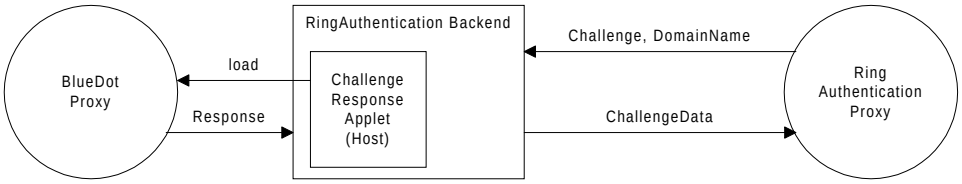


Figure 6.14: Ring Authentication Service

```
}
```

`RingData` is just another data container. `getAppletName()` is used to select the right `Applet` on the ring. `getPassword()` returns the overall Ring password which is needed to access the ring. `URL` specifies the file which contains proprietary information of how to access the `Applet` which has to be downloaded first by the `BlueDot` implementation. `getMessage()` should return a message that can be printed out, so that the user knows what applet is running.

The `BlueDot` interface has to be implemented by the service proxy, whereas the `RemoteBlueDot` interface has to be implemented by the service backend. The most important difference is that the remote interface extends indirectly `Remote`, whereas the local one extends indirectly `Serializable`. The communication between the proxy and the backend has not to be secured, since no critical data has to be exchanged, otherwise the service could be extended easily to provide a secure communication as mentioned above.

### 6.10.5 RingAuthenticaton Service

The `RingAuthentication` has a very simple interface like the most other services consisting of only one method that gets a `Challenge` and a `DomainName` and should return a `ChallengeData` which contains the `Response`:

```
interface RingAuthentication extends Pingable{
   ChallengeData challengeResponse(Challenge, DomainName);
}

interface RemoteRingAuthentication extends RemotePingable{
   ChallengeData challengeResponse(Challenge, DomainName);
}
```

The service proxy, which implements `RingAuthentication`, calls the method of the service backend which implements `RemoteRingAuthentication`.

To do a Java Ring authentication, the service backend has to find a `BlueDot` service and has to call its `runApplet()` method with an appropriate `RingApplet` that communicates with the challenge-response applet on the Java Ring (figure 6.14).

## 6.11 Service

One goal is to simplify the development of new services, so that the underlaying authentication is as transparent as possible. In general, there are two different ways to achieve this:

- provide a class which can be extended
- provide a class which can be used as a field

The advantage of the first approach is that more work can be done by the super class. Since Java does not support multiple inheritage, the class hierarchy is very static, whereas this is exactly the advantage of the second approach. But the second approach lacks of not having that much work be done by the class.

This architecture provides both on both sides: a class that encapsulates the whole authentication functionality and also classes that can be extended and act as wrapper.

### 6.11.1 Interface Definitions

Consequently, the service proxy contains a class that encapsulates the authentication functionality. Its interface description consists of two parts:

```
interface Authenticable extends Serializable{
   void login();
   void logout();
}

interface Authenticator extends Authenticable{
   SecureClient getSecureClient();
   SlotRes getSlotRes();
}
```

The remote counterpart of `Authenticator` is `AuthenticatorBackend` which also provides some methods needed for an authenticated session:

```
interface AuthenticatorBackend{
   SealedMacObject login(PublicKey, RemoteCallbackHandler,
   SealedCookieMacObject);
   void logout(SealedCookieMacObject);
   SecureSlotProvider getSessionSlotProvider();
   getSignedObject(String);
}
```

Since the communication has to be secured between the service proxy and the service backend, results and parameters are sealed objects. `AuthenticatorBackend` does not need to extend `Remote` or `Serializable`, because it is intended to be a field in the service backend (figure 6.15).

### 6.11.2 Proxy

The first interface only describes in general, what a basic authentication functionality is: starting the authentication by calling `login()` and close the session with `logout()`.

The second interface extends the first and provides two methods, in order to access the underlaying secure communication tier. The role of `getSecureClient()` and `getSlotRes()` is simple. They only have to return the `SecureClient` and the `SlotRes` which contain the cookie, token and the lease. Currently, `getSecureClient()`

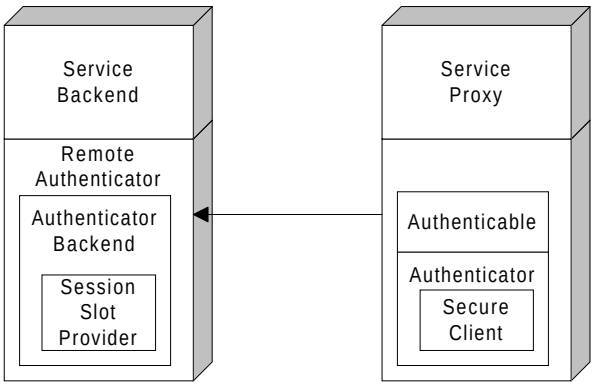


Figure 6.15: Service Backend and Service Proxy

is needed, since the `sealObject()` and `unsealObject()` has to be used by methods on service proxy side to decrypt results and to encrypt parameters.

The intention of `getSlotRes()` is to provide the lease, so that the lease can be canceled or renewed manually. The second interface can be removed in the next version. Sealing and unsealing will be done transparently. Canceling the lease is done implicitly by calling `logout()` and the renewal can be done by a `LeaseRenewalManager`. That means the whole interface would consist of two methods: login() and logout() which will be described next. If `getSecureClient()` or `getSlotResult()` is called, without calling `login()` first, the `login()` method has to be called implicitly. In general, every method in the proxy that calls methods of the service backend has to ensure that the proxy, resp. the user, has already been logged on, otherwise the method has to start the log on process by calling `login()`.

To log out, `logout()` has to call the `logout()` method of the `RemoteAuthenticator` which will be implemented by an object inside the service backend. If the service proxy has not been logged in, the method should return silently.

The most important method of `Authenicator` is `login()` which handles the whole login process at client side encapsulated in the service proxy. If `login()` is called, and a login session already exists, `logout()` should be called implicitly, so that a second login session can be started. An alternative would be to return silently, like `logout()` it does if no login session exists, but this is not the recommended way.

To be able to create a secure communication link, the `login()` procedure of the service proxy has to ensure that a cryptographic provider is installed that provides all the used algorithms, if not, it has to install an appropriate provider.

**SignedLoginData**

The next step is to check the signature of the dynamic data that the service proxy contains. `Register` and `RemoteRegister` use a `SignedObject` to exchange `SignedRegisterData`. For the reasons of log on, `SignedRegisterData` has to be extended to provide further information. It needs the service name from the service backend to built the `LoginData` object which is also given to the login policy service and it uses the name to determine the login policy:

```
interface SignedLoginData extends SignedRegisterData{
```

```
   String getService();
}
```

Having checked the signature, the DH key agreement can be executed, so that the secret communication key is available. The secret key and the `DomainName` have to be used to instantiate the `RemoteCallbackHandler`.

Two information are still needed to instantiate the `LoginData` for the purpose of calling the remote `login()` method. These are the `DefaultUser` and the `LoginToken`. Both can be `null` if they do not exist at client side.

The other objects needed to instantiate the `LoginData` are: the attempt count, the service name and the secret key. The secret key and the service name are also available at client side, so that it is also possible to set these both values in the service backend. If it is done that way, `SignedLoginData` has not be used, because it only contains the service name.

However, now, the remote `login()` method can be called with the public DH key, the `RemoteCallbackHandler` and the sealed `LoginData`.

### 6.11.3 Backend

First, the service backend has to generate the secret key using the given public DH key and its private DH key. This is done by the secure slot provider, so that the result is a `SlotRes`. The `SlotRes` contains the secret key which has to be used to unseal the `LoginData`.

Having the `LoginData` and the `RemoteCallbackHandler`, the `SubjectAuthenticator` service can be called. Its `login()` method returns a `SubjectAuthResult` if the `login()` succeeds. The `SubjectAuthResult`, which contains the `Subject`, is stored in the secure slot provider.

#### LoginResult

Last, a `LoginResult` has to be created. It contains the information that the service proxy needs for the login session: the `SlotRes` to communicate with the service backend and the `SlotRes` which contains the leased `LoginToken`:

```
interface LoginResult extends Serializable{
   SlotRes getLoginSlotResult();
   SlotRes getSlotResult();
}
```

This `LoginResult` has to be sealed and returned as result of the service proxy `login()` invocation.

The service proxy unseals the result and has to store both `SlotRes`s. At least, it has to set the `Cookie` and `Token`, so that its `SecureClient` can do the communication with the service backend.

Furthermore, it is up to the implementation if the service proxy writes the `LoginToken` to disk or do some equivalent action, so that the `LoginToken` can be retrieved by another login process from the same domain. From now on, the user is authenticated and the service backend can use the `Subject` to perform authorizated actions.

The `getSignedObject()` has to return the `SignedObject` containing `SignedLoginData`, whereas the `String` argument is the service name. To sign the object, the private signing key has to be retrieved from the key store. The `DomainName` is used as alias.

The `getSessionSlotProvider()` method has to return the `SessionSlotProvider`, so that the sealing and unsealing can be done. In the next version of this architecture, this should be done transparently, so that this method is needed no longer.

### 6.11.4 Remote Interface

The implementation of the interfaces above can be used as fields in the service proxy and the service backend. The implementation also provides abstract classes which act as wrapper to call the appropriate methods of the fields. The service backend has to implement at least the following interface, whereby the methods only have to act as a wrapper to call the appropriate methods of the `AuthenticatorBackend`:

```
interface RemoteAuthenticator extends Remote{
   SealedMacObject login(PublicKey, RemoteCallbackHandler,
   SealedCookieMacObject);
   void logout(SealedCookieMacObject);
}
```

The service proxy does not need to implement such an interface, because the client is using the service proxy and the whole process is transparent to the client. The login process is started implicitly: if the client calls a method on the service proxy that requires a login session, the method has to check if the user is logged on and if not, it has to start the login process.

# Chapter 7

# Implementation

This chapter only provides implementation facts about the reference implementation. All the concepts have already been shown in the chapters about the architecture. So this chapter is especially for those, who want to alter or expand the architecture. It is also possible to extract some parts. A recommendation is to read the source code listing while studying this chapter.

## 7.1 Package Structure and Conventions

The whole architecture including the reference implementation consists of 26 packages and 145 classes. One half is the interface definition and the other half is the implementation. The next sections give an overview of the package structure and the conventions used, provide some common infrastructure that can be used and describe the architecture starting from the client side.

Another distinction is made between security and non security packages, so that there are 4 portions of packages (”-” means a whole subtree):

- non security interfaces: schoch.jini.interfaces.-, depends on no other of the three packages,
- non security implementation: schoch.jini.reference.-, only depends on the non security interfaces,
- security interfaces: schoch.jini.security.interfaces.-, only depends on the the non security interfaces,
- security implementation: schoch.jini.security.reference.-, depends on all the packages.

### 7.1.1 Proxy Interface

Regularly, a service has to implement two interfaces. The service proxy has to implement the interface that specifies the protocol, how a client talks to the service. This is a local interface, because the service proxy is downloaded and then executed locally in the client thread.

### 7.1.2 RMI

The communication between the service proxy and in general, its backend service goes over the network. The most common solution is to use Java's built-in Remote Method Invocation (RMI).

RMI generates transparently the appropriate stubs and skeleton classes needed for marshalling and unmarshalling. Since JDK1.2, skeletons are needed no longer. To create a remote accessable `Object`, the most common way is to extend a `UnicastRemoteObject` or a `Activation` class and to instantiate it.

A client that wants to use the remote object, needs a remote reference of the object which it obtains from the RMI registry (not used in this architecture), or directly as a regular object: `RemoteReference`.

The interface which the client uses as a protocol to the talk to the remote object, and which the client stub and the remote service must support, has to extend `Remote`. `Remote`s only task is to mark the defined protocol interface as a remote protocol. That means a separate program, besides the java compiler (`javac`), the Remote Method Invocation Compiler (`rmic`), generates the appropriate stub classes for those interfaces which extend `Remote`.

### 7.1.3 Backend Interface

Back to the interfaces that a service has to implement. The service proxy has to implement the local protocol interface, whereas the service backend implements a remote interface. If the local and remote are equal except for the extension of `Remote`, it is possible to only use one interface. This also implies that it is not possible to hide `RemoteException`s, which are thrown by the stub, or the remote object.

In this architecture two different interfaces are needed, because the communication between the service proxy and the service backend is encrypted. Whereas the communication between the client and the service proxy is not encrypted, because it is a regular local method invocation. This also shows that the encryption is transparent to the client. A convention is to name the remote interface, like the local interface, with `Remote` as prefix, e.g. a local interface named `TV` implies that the remote interface has to be named `RemoteTV`.

## 7.2 Data containers

Many of the classes are just data containers. Their structure is to have public getter and private setter methods. The constructor is public, too and it uses the private setter methods to setup the fields. The naming convention is to add the postfix `Impl` to the interface name, e.g. interface Example implies class ExampleImpl. The following classes are only data containers:

- `RingDataImpl`
- `LocalChallengeCallbackImpl`
- `LocalNamePasswordCallbackImpl`
- `RemoteChallengeCallbackImpl`
- `RemoteNameCallbackImpl`

- `RemoteNamePasswordCallbackImpl`
- `RemotePasswordCallbackImpl`
- `UserDBCallbackImpl`
- `AuthDataImpl`
- `ChallengeDataImpl`
- `LoginDataImpl`
- `LoginResultImpl`
- `RemoteCallbackHandlerParamImpl`
- `SignedLoginDataImpl`
- `SignedRegisterDataImpl`
- `SubjectAuthResult`
- `SubjectDataImpl`
- `TokenDataImpl`
- `TransactionDataImpl`
- `UserDataImpl`
- `SlotResImpl`
- `JaasSlotResultImpl`

### 7.2.1 Extended Data Containers

Some data containers also define `toString()`, `hashCode()` and `equals()` methods:

- `PrincipalImpl`
- `SlotDataImpl`
- `SecureSlotDataImpl`

## 7.3 Cookie and Token

The four marker classes

- `LoginCookieImpl`
- `LoginTokenImpl`
- `ServiceCookieImpl`
- `ServiceTokenImpl`

have the common super class `CookieTokenImpl`. This class uses internally a 16 byte array (128 bit) to uniquely represent a cookie or token.

It is important to use a good random generator, because the token represents the right to act on behalf of a user. So guessing the right token should be very hard. This implementation uses the `SHA1PRNG` algorithm from the Java Security architecture. The 16 random bytes are set in the no argument constructor.

## 7.4 Exchange Data

`ChallengeDataImpl` has the same implementation as `CookieTokenImpl`. One difference is that the `toByteArray()` method is public, so that the byte array could be read.

`ResponseDataImpl` needs no random generator, because the byte array is given to the constructor.

`UserDataImpl` uses a `String` to store the user-name. `DefaultUserImpl` only is a marker class and extends `UserDataImpl`.

`PasswordImpl` uses a char array to store the password. The `clearPassword()` method overrides the char array with spaces. It also provides a `finalize()` method which only calls `clearPassword()`.

The `DefaultLoginPolicy` returns the `String` "Default" if `getName()` is called. It only implies `DefaultLoginPolicy`s.

## 7.5 Secure Communication

### 7.5.1 Encryption and Decryption

The constructor of `SealedMacObject` gets a `SecretKey` and an `Object`. It serializes the `Object` into a byte array using a `ByteArrayOutputStream` and an `ObjectOutputStream`. On that byte array and with the secret key the MAC and the encrypted version are calculated and stored into two byte arrays.

`getObject(SecretKey)` does the reverse operation. It decrypts the stored and encrypted byte array and compares the MAC of that decrypted byte array to the stored MAC. If they are not equal, an `Exception` will be thrown, otherwise the original object will be deserialized with a `ByteArrayInputStream` and an `ObjectInputStream`.

The `SealedCookieMacObject` is just an extension to `SealedMacObject` which additionally stores a `Cookie` that is given to the constructor.

### 7.5.2 Slot Provider

A common super class for secure and non secure slot providers is `AbstractSlotProviderImpl`. It stores the duration which is needed to calculate the expiration, a landlord to call back and uses a `LandlordLeaseFactory` to generate new `Leases`. Both arguments are given to the constructor.

It uses a `HashMap` to store the context information. `getToken()` and `getCookie()` are defined abstract, because extensions of this class define their own cookies and tokens. `getData()` and `setData` map the cookies to the context information.

The `cancel()` and `renew()` methods are also defined, but currently do nothing. These methods should be called by the landlord that was used to build the leases. An-

other approach is to make AbstractSlotProviderImpl an RemoteObject, so that it can be its own landlord.

#### SlotProviderImpl

SlotProviderImpl provides an implementation for generateCookie() and generateToken(). These methods return a new LoginCookie or LoginToken. The most important method is leaseSlot(). It generates a new Token, Cookie and a Lease using the three generateXXX() methods, so that a SlotData can be instantiated and be put into the HashMap using the Cookie as key.

#### SecureSlotProviderImpl

A more sophisticated extension of AbstractSlotProviderImpl is SecureSlotProviderImpl. How the public methods work, is described above more in detail. Three important private methods are completeKeyGeneration(), getEncryptCipher() and keyGeneration().

The last method is called, when the object is created. A KeyPairGenerator is initialized with the key agreement algorithm, specified in SecureClient and a DHParameterSpec, i.e. the modulo and the base, also in SecureClient. After the key pair is generated, the private and public key is stored separately.

The next step is to call completeKeyGeneration() with the public key of the secure client as argument. A KeyAgreement object is created and initialized with that public key and the own private key. In the end, the generated SecretKey could be returned.

getEncryptCipher() is quite simple, it takes a SecretKey as argument, creates a new Cipher object in encryption mode and returns it.

### 7.5.3 Secure Client

The counterpart of SecureSlotProviderImpl is SecureClient. The implementation of the public methods described in the architecture is straight forward.

It has fields to store the Cookie, Token, SecretKey and the transaction number needed to communicate with a SecureSlotProvider. Another field is type that is used to determine if the SecureClient is in active or passive mode.

One of two important private methods is checkTransactionNumber() that checks if the given and the own transaction number are equal and increases the transaction number in active mode.

The second private method is getEncryptCipher(). In that case, it needs no arguments, because it can access directly the SecretKey, so it can return a new instance of Cipher in encryption mode for the sealObject() method.

## 7.6 Common classes

### 7.6.1 ServiceFinderImpl

ServiceFinderImpl is created giving a ServiceTemplate to the constructor which stores it in a field. The constructor also creates a LookupDiscovery and adds itself as DioscoveryListener. It uses a Set to keep track of the discovered lookup services. Another field stores a proxy that fits to the ServiceTemplate.

If `getProxy()` is called or a new lookup service is found, `findService()` is called. This method iterates through the set of lookup services, tries to get a proxy and registers on that proxy if it implement the `Register` interface. The `getProxy()` method also tries to call `ping()` if the proxy implements the `Pingable` interface.

### 7.6.2 AbstractRingApplet

A common super class for all host programs that communicate with a Java Ring, is `AbstractRingApplet`. It has one field that stores the `RingData`, needed to communicate with the `Applet` on the ring and also provides the `decodeSW()` method that returns the status word received from the Ring, and returns a `String` describing the status word.

### 7.6.3 BasicUnicastService

`BasicUnicastService` is taken from Keith Edwards Core Jini. This abstract super class does most of the standard initialization with the Jini infrastructure. A service which extends this class has to implement the abstract method `getProxy()` which is used by `BasicUnicastService` to register the proxy with the lookup service.

An instantiation of an extended class works like this: the first step is to instantiate the super class, the second step is to call the `initialize()` method which does the remaining initialization and starts the service.

`BasiscUnicastService` extends the `UnicastRemoteObject` class and stores its context, e.g. `ServiceID` persistently in a file.

### 7.6.4 RegisterImpl

An extension to `BasicUnicastService` is `RemoteRegisterImpl` that provides the additional ability to register on a service using a proxy that extends `RegisterImpl`. Registration means to agree on a common secret key that is used by the client and the service to encrypt and decrypt the communication. There are no restrictions on who does the registration.

`RegisterImpl`, the proxy, has a `boolean` field to indicate if the client has registered. It also provides a field to store an own instance of `SecureClient` and the result of the `SecureSlotProvider` at service side: a `SlotResult`.

To communicate with the backend service, it has a `RemoteRegister` field. A `SignedObject` field is used, to store the `SignedObject` that the services creates and stores it to the field, before it registers the proxy on lookup services.

`register()` and `unregister()` have already been explained in the chapter about the architecture. The private methods `checkSignature()` and `installProvider()` are worth to explain.

`installProvider()` is called by `register()`. It creates a new instance of the `SunJCE` provider. If a provider with the name of the instantiated does not exist, the provider is added to the JVM.

The second method `checkSignature()` is called by `register()`, too. A new `KeyStore` using the `SecureClient.KEYSTOREALG` is created and the file `SecureClient.KEYSTORENAME` in the users home directory is used to build the `KeyStore`. The public verification key is retrieved from the `KeyStore` by the `DomainName` and calling `getPublicKey()` on the returned `Certificate`. A verification engine which is initialized with the `SecureClient.SIGALG` and the

PublicKey are given to the stored SignedObject method verify() as arguments. If the check is positive the data inside the SignedObject are returned, otherwise an Exception is thrown.

### 7.6.5 RemoteRegisterImpl

The counterpart of RegisterImpl is RemoteRegisterImpl with three fields and one constant. SERVICE_DURATION is the constant that stores the maximum duration of the leases that are used by the ServiceSlotProvider that is stored in one of the three fields. The DomainName and a PrivateKey, the signing key are the remaining two fields.

The two cancel() methods and the two renew() methods only call the corresponding methods of the SecureSlotProvider.

Two methods are explained more in detail: getSignedObject() and loadSigningKey().

loadSigningKey() is called by the constructor with the private signing key access password as argument. As described above, a KeyStore is created from a file. KeyStore.getKey() returns the private signing key, whereby the DomainName and the password are used to access the private signing key.

getSignedObject() is called, when the proxy is created which stores that SignedObject. Therefore a new SignedRegisterDataImpl is created and put together with the private signing key and a signing engine into the constructor of a SignedObject which is returned.

## 7.7 Callback

Most of the classes, i.e. the Callbacks itself are only data containers and therefore not worth to explain more in detail, whereas the LocalCallbackHandler and the RemoteCallbackHandler are worth to mention.

### 7.7.1 RemoteCallbackHandler

The RemoteCallbackHandler has fields to store the DomainName, a SecureClient for the communication and a ServiceFinder to find a RingAuthenticationService.

The SecureClient is created in the constructor in passive mode with the SecretKey that is used by the service proxy and the service backend.

The most important method is handle(). It gets a SealedCookieMacObject that contains an array of RemoteCallbacks. Depending on the type, which is checked by instanceof, different authentication schemes can be processed. This implementation supports two RemoteCallbacks: RemoteNamePassordCallback and RemoteChallengeCallback.

The implementation of both schemes is straight forward: from the RemoteNamePasswordCallbacks are extracted the RemoteNameCallback and the RemotePasswordCallback. Both messages are prompted to the user. The Username and the Password are stored in an AuthData object. After sealing the return value, the password is cleared. The implementation ignores the isEchoOn() value. readPassword() is a simple method that reads a char array from the console.

The processing of the RemoteChallengeCallback is even simpler. The RingAuthentication finder is used to get a proxy. On that proxy challengeResponse() is called with

the `Challenge` of the `RemoteChallengeCallback` and the stored `DomainName`. The return value is a `ChallengeData` object which is returned sealed.

The implementation currently does not fulfill the specification. Because it does not return an sealed array of objects. It only returns the first result. This must be altered in a future release.

### 7.7.2 LocalCallbackHandler

The `LocalCallbackHandler` is an implementation of JAAS' `CallbackHandler` interface. It stores a `UserDB` proxy which can be requested by a `LoginModule`, a `SecureClient` to secure the communication and a `RemoteCallbackHandler` which is described above.

The `LocalCallbackHandler` is invoked by a `LoginModule`. Depending on the `LocalCallback`, a `LocalNamePassword`, a `UserDBCallback` or a `LocalChallengeCallback` can be requested.

`UserDBCallback` only has to return the stored `UserDB`. The two remaining `LocalCallback`s are processed by calling the `handle()` method of the `RemoteCallbackHandler` with the corresponding `RemoteCallback`s. That means the `LocalCallbackHandler` acts as a wrapper for the `RemoteCallbackHandler`.

## 7.8 Authentication classes

The four classes given here are the foundation of all services which need authentication and authorization.

A service proxy extends `AbstractProxy` which uses an `AuthenticatorImpl`. Whereas the service backend extends `AbstractBackend` which uses an `AuthenticatorBackendImpl` to provide the authentication functionality.

Both classes, i.e. `AuthenticatorImpl` and `AuthenticatorBackendImpl`, can be used directly, so that is not mandatory to extend from the given classes that is the more flexible concept if extension is not possible. Whereas the most easiest way is to extend the both classes. These both classes only act as a wrapper. That means `AbstractProxy` calls the corresponding methods of `AuthenticatorImpl` and `AbstractBackend` calls the corresponding methods of `AuthenticatorBackendImpl`.

### 7.8.1 AbstractProxy

Therefore, an `AbstractProxy` provides two methods: `login()` and `logout()` that calls the appropriate methods on `Authenticator` that is stored in the field `authenticator`.

### 7.8.2 AbstractBackend

`AbstractBackend` is a little bit more complex than `AbstractProxy`. It implements the `Landlord` interface and calls the appropriate methods of the `AuthenticatorBackend` that is stored as a field. `RemoteAuthenticator` is implemented by this class, too.

Therefore the `login()` and `logout()` methods call the methods of the `AuthenticatorBackend`.

`getAuthenticator()` and `getSignedObject()` are two protected methods. The first returns the own field, whereas the second one returns the result of the corresponding method of the `AuthenticatorBackend`.

As explained above, AuthenticatorImpl and AuthenticatorBackendImpl are very crucial for the service itself.

In a way, these classes are similar to RegisterImpl and RemoteRegisterImpl. The main difference is that registration needs no authentication, whereas authentication is needed by log on sessions.

loggedIn is a boolean field that stores the state of the session. Two fields are used to store the slot results. slotRes stores the result that is needed to communicate with the SubjectAuthenticator service, whereas the second one is needed to relogin with a LoginToken which the loginSlotResult contains.

The SecureClient has got the object that encapsulates the secure communication. A SignedObject with the public DH key among other things is stored in the field signedObject. The last field remoteCallbackHandler holds a reference for a RemoteCallbackHandler.

The constructors sets the given values for signedObject and remoteAuthenticator. A new SecureClient is created here.

The private methods checkSignature() and installProvider() are mostly equal to the methods described in RegisterImpl. The methods getCallbackToken() and getDefaultUser() still have to be implemented. Currently, they return null.

Every method that calls methods of the backend, checks if already have been logged in. If not, login() is called explicitly, whereas login() calls logout() first if the user has already been logged in.

After that check, installProvider() and checkSignature() are called to ensure that the communication can be trusted. Then, the public DH key stored inside the SignedLoginData is used to generate the secret DH key by calling SecureClient.keyGeneration(). That newly created secret key and the DomainName, also stored in the SignedLoginData, are the arguments to the RemoteCallbackHandlerImpl.

After creation of a LoginData, the login() method of the backend can be called, whereas the LoginData will be sealed and the result will be unsealed. The result contains the both SlotResults which are stored in the both fields. In the end, the Token and the Cookie for the SecureClient can be set for further communication. The very last step is to set loggedIn to true, since the authentication has been succeeded.

### 7.8.3 AuthenticatorBackendImpl

The backend version of AuthenticatorImpl is AuthenticatorBackendImpl that implements the AuthenticatorBackend interface.

This implementation has four fields to store a DomainName, a session slot provider, a signing key and a field for a SubjectAuthenticator finder. A constant called SESSION_DURATION states the duration of a LoginToken.

The constructor takes a Landlord, a DomainName and a password as char array. The password is used in loadSigningKey() to restore the private signing key from its stored encrypted version. The constructor creates a new ServiceFinder to find a SubjectAuthenticator. It also creates a new SecureSlotProvider, using the Landlord and the SESSION_DURATION as arguments. Finally, the verb—domainName—is set and loadSigningKey() is called with the password.

loadSigningKey() and getSignedObject() are very close to the same methods in RemoteRegisterImpl. One difference is that SignedLoginDataImpl is used instead of its superclass SignedRegisterDataImpl.

`login()` uses the parameter `publicKey` to lease a slot. After retrieving the `Cookie` from the `SlotResult`, the encrypted `LoginData` could be decrypted. Then, the `LoginData` and the stored `RemoteCallbackHandler` is used to call the `login()` method of the `SubjectAuthenticator` service.

If the authentication succeeds, the service returns a `SubjectAuthResult` which is stored in the `SecureSlotProvider` with the Cookie as key. Finally, the encrypted and newly created `LoginResult` containing the both `SlotResult`s is returned.

## 7.9 BlueDot Service

### 7.9.1 Proxy

The `BlueDotProxy` implements the `BlueDot` interface. This implementation has one field that is a reference to the backend, a `RemoteBlueDot` and is set by the constructor. `ping()` and `runApplet()` only call the corresponding methods of the backend.

### 7.9.2 Backend

`BlueDotService` which implements `RemoteBlueDot` is more sophisticated. It has to provide a scheduling for all incoming requests that want to run an `Applet` on the Ring.

The incoming requests are stored in a `List` called `queue`. The number of ports is stored in `ports`, whereby the number of free ports is stored separately in `free`. A mapping between ports and requests to run on are stored in the `Map plan`.

There also must be a field to store a reference to a `JibMultiFactory`. It is used to communicate to the Blue Dot Receptor and therefore to the Java Ring.

The constructor sets `free` and `ports` to one. There also should be a way to determine the number of ports dynamically. A `Vector` is created as `queue` and a new `HashMap` is used as `plan`. The `JibMultiFactory` is created with a no arguments constructor. Then `addJibListener()` and `startPolling()` have to be called. Since `BlueDotService` implements `JibMultiListener`, it registers itself.

One of the two arguments of the `BlueDotService` is the storage location which is given as argument to the superclass `BasicUnicastService`. The other parameter is the `String path` which determines, where to store the downloaded files. But the `main()` method only takes one argument, i.e. the path. The filename of the storage location is the constant `FILENAME`.

`getProxy()` returns a new instance of a `BlueDotProxy` as expected.

The request scheduling is done by the four methods `iButtonInserted()`, `iButtonRemoved()`, `runApplet()` and `dispatchSlots()`.

The `runApplet()` method creates a new `BlueDotEntry` which stores the caller `Thread`, an exception if thrown while execution and the `RingApplet` to run.

After enqueuing, `dispatchSlots()` is called. The execution of the current`Thread` can now be suspended by calling `suspend()` on the `BlueDotEntry`'s `Thread`. Since the execution only resumes if the `Applet` has been executed, the exception inside the `BlueDotEntry` can be checked. If it is `null`, `ringApplet.getData()` is returned, otherwise the given `Exception` will be thrown.

`dispatchSlots()` searches for the first free slot if the amount of free slots is more than zero. The `BlueDotEntry` is removed from the `queue` and put into the `Map plan` using the number of the port as key. A message, which is stored in the `RingData` inside the `RingApplet`, is printed to the console.

`iButtonInserted()` is called automatically if a Java Ring has been inserted. Using the `JibMultiEvent` parameter, the slot can be determined, and therefore the `BlueDotEntry` of the plan can be retrieved. If there is no `BlueDotEntry` for the specified port, an error message is printed. Otherwise the `setSlot()` method of the `JibMultiFactory` is called with the slot as argument.

A `Selector` has to be created to run the specified `Applet` on the Java Ring. The data for that is stored in the `RingApplet` and the `RingData` which are accessable by the `plan`. After the invocation of the `Selector`'s `select()` method, the `RingApplet`'s `run()` method could be called.

The counterpart of `iButtonInserted()` is `iButtonRemoved()`. Depending on the slot, the corresponding `BlueDotEntry` is retrieved from the `plan`. The caller `Thread` is reactivated by calling `resume()` on it. The amount of free slots will increment, the `BlueDotEntry` will be removed from the `plan` and `dispatchSlots()` is called, since the current slot can be dispatched.

The proprietary software to select and run an `Applet` on the Java Ring needs an external "jib file. Therefore, the jib file needs to be downloaded. The method `writeJibFile()`, which is called in `runApplet()`, uses the URL inside the `RingData` to download the jib file with a `InputStream`. It is written using a `FileOutputStream` into the stored path with its applet name.

## 7.10 RingAuthentication Service

Building on the `BlueDot` service, the `RingAuthentication` service provides the ability to run a challenge-response `Applet` on a Java Ring. Therefore a `RingApplet` is needed which can be given as an argument to the `BlueDot` service.

### 7.10.1 RingAuthenticationApplet

`RingAuthenticationApplet` extends `AbstractRingApplet` and can execute a challenge-response `Applet` on the Java Ring.

Three fields are needed: `challenge` stores the given `Challenge`, `challengeData` references the `ChallengeData` that contains the `Response` and a `byte` array `send` is used to communicate with the Java Ring.

Only `byte` arrays can be exchanged with the `Applet` on the Java Ring, so that a protocol needs to be defined. All the relevant information are given to the constructor, i.e. the `RingData`, the `Challenge` and the `DomainName`.

The protocol for the challenge-response invocation is: length of `Password` + `password` + length of `DomainName` + `DomainName` + `Challenge`. Currently, in this sample implementation no password is required, so the first `byte` is zero and is followed by the length of the domain.

`DomainName` and `Challenge` provide the method `toByteArray()` which returns the representation of these objects as `byte` arrays. Doing it that way, the fields `send` and `challenge` can be set in the constructor.

The `run()` method calls the `encryptDispatch()` method and prints out the end message. `encryptDispatch()` is the central point of that class. It uses the

JibMultiFactory that is given as parameter, to contact the Java Ring Applet using the sendAPDU message, whereby it creates a new CommandAPDU and receives a ResponseAPDU.

Every ResponseAPDU has got a status word that can be accessed by calling sw() on it. The response contains a byte array that is structured that way: length of User + User + Response.

Both objects are restored from the byte array representation and a new ChallengeData can be created with both arguments, so that getData() can return it.

### 7.10.2 Proxy

The RingAuthenticationProxy implements the RingAuthentication interface and contains a reference to a RemoteRingAuthentication object which is contacted if ping() or challengeResponse() is called.

### 7.10.3 Backend

RingAuthenticationService extends BasicUnicastService and implements the RemoteRingAuthentication interface.

It has for constants: APPLET_NAME is used to access the Applet on the Java Ring. FILENAME is needed to store the context information of that service. MESSAGE is used by the BlueDotService to print some information for the user and at last PASSWORD currently as a ""-String, since this implementation does not use a password to access the Java Ring Applet.

The class provides two fields besides the constants. blueDotFinder is a ServiceFinder used to contact the BlueDotService, i.e. to invoke the runApplet() method of it. Finally, it also stores a RingData object that contains the access information needed by the BlueDot service.

The main() method expects one argument: the URL, where the jib file, needed by the BlueDot service can be downloaded. Using the constant FILENAME and the URL, which is enhanced by the APPLET_NAME and the APPLET_POSTIX, the constructor is called.

The real start of the service is done by calling the initialize() method. The constructor creates a new RingDataImpl with the given information and sets the field. It also searches a BlueDot service by creating a new ServiceFinder. The reference is stored to the appropriate field.

getProxy() returns a new instance of RingAuthenticationProxy as expected that only takes the reference of the service as argument.

The most important method of this class: challengeResponse() is very short. Using the RingData field and the two parameters Challenge and DomainName a new RingAuthenticationApplet is created. Then, the ServiceFinder is used to retrieve a valid reference of a BlueDot proxy. The result of the invocation of the runApplet() method of the BlueDot service is returned, whereby the argument was the RingAuthenticationApplet.

## 7.11 UserDB service

The UserDB service consists of two parts: the service itself and the corresponding proxy.

### 7.11.1 Proxy

The UserDBProxy extends the RegisterImpl and implements the UserDB interface.

Its constructor has to the set the field remoteUserDB that references the backend service and it is used by the methods authenticate(), ping() and refresh() to call the appropriate methods on the service backend. They also check if already been registered, otherwise register() is called first.

### 7.11.2 Backend

The service backend, i.e. UserDBService extends RemoteRegisterImpl and implements the RemoteUserDB interface.

Currently, the UserDB only provides two users for two authentication schemes. It does not contact some kind of underlaying user database.

For password authentication, the users schoch and krone with their passwords thomas and oliver are hard coded. For use with an Java Ring, two DESede keys are created and stored into a well-defined directory if they does not already exist there.

Therefore, a few constants are needed. PATH defines where the keys should be stored, i.e. from where to restore. SEPARATOR defines, how the user and the domain name are separated in a filename and SYMTYP defines the prefix of such a filename: schoch-john.secretKey.

SYMALG specifies which algorithm should be used for the challenge-response procedure. The one used in this implementation is DESede/ECB/NoPadding. SYMKEYTYP defines the keys: DESede.

FILENAME and SERVICE_DURATION are two more fields that store, where to store the services context information and the maximum duration of a lease.

passwordMap and secretKey are two Maps that map the user-name to the password, i.e. SecretKey.

The main() method expects two arguments: the DomainName and the password to access the key store. The first thing is to add a new SunJCE security provider to the JVM. Then, a new UserDBService can be created using the FILENAME, the DomainName and the password as char array. The last thing is to call the initialize() method of that object to start the service.

Inside the constructor, two HashMaps are created and generateMaps() is called for the two fields. This methods put the password thomas and oliver in the passwordMap for the users schoch and krone.

It also tries to retrieve the keys for the challenge-response procedure from file. Therefore getFilename() builds the appropriate filename consisting of path, User, DomainName and SYMTYP. If the file exists, getSecretKey() is called which retrieves the SecretKey by using an ObjectInputStream, otherwise writeKey() is called which generates a new SecretKey, writes it to a file using a ObjectOutputStream and returns the key as result of the invocation. Now, the SecretKey and the User could be put into the secretKeyMap.

Since the user data is static, refresh() does currently nothing. getProxy() returns a new instance of UserDBProxy as expected.

The arguments for the constructor are a reference of the service itself and the result of getSignedObject(). The most important method is authenticate(). It unseals the SealedCookieMacObject and calls the appropriate doAuthentication()

methods with the unsealed object as parameter if the the unsealed object is an instance of `AuthData` or `ChallengeData` method.

The verification of the password is done by comparing the stored password with the one in `AuthData`, whereby the user-name is the key to retrieve the password. If the password is incorrect, the password or the user does not exist in the database an `Exception` will be thrown, otherwise the stored `subjectData` will be returned.

The verification of the challenge-response is similar. The `SubjectData` is retrieved from the map using the user-name. Then, the `SecretKey` is taken from the `SubjectData` and the `Response` is calculated from the challenge using the `SecretKey`. If the calculated `Response` equals to the `Response` in the `ChallengeData` parameter the `subjectData` will be returned, otherwise an `Exception` will be thrown.

The calculation of the `Response` is done by initializing an `Cipher` object in `ENCRYPT_MODE` with the `SecretKey`. The `Response` is retrieved by calling `doFinal()` on the `Challenge`.

## 7.12 LoginPolicyDB service

The structure of this service is very similar to the `UserDB`'s one.

### 7.12.1 Proxy

The `LoginPolicyDBProxy` extends `RegisterImpl` and implements the `LoginPolicyDB` interface. The field `remoteLoginPolicyDB` stores a reference to the backend which is used in the three methods `ping()`, `refresh()` and `getLoginPolicy()` to call the corresponding methods. Every methods checks if the proxy has already been registered, otherwise `register()` will be called first.

### 7.12.2 Backend

`LoginPolicyDBService` is the backend for the proxy, it extends `RemoteRegisterImpl` and it implements the `RemoteLoginPolicyDB` interface.

Its only constant is `FILENAME` which determines where to store the context information.

Since this is a sample implementation, only one `LoginPolicy` is returned, not depending on the three dimensions: `DefaultUser`, service and attempt. That means `getLoginPolicy()` does the sealing and unsealing of the parameter, i.e. the result and uses `doLookup()` to retrieve the `LoginPolicy`.

`doLookup()` returns a new instance of `DefaultLoginPolicy`.

`getProxy()` returns a new instance of `LoginPolicyDBProxy` as expected.

`refresh()` does currently nothing, since the policy is hard coded.

## 7.13 Jaas service

The `Jaas` service uses the JAAS module to implement the `SubjectAuthenticator` service. This implementation also includes two implementations of a `LoginModule`.

### 7.13.1 ConsolePasswordLoginModule

`ConsolePasswordLoginModule` is one these implementations. The four fields that stores `Subject`, `CallbackHandler`, `sharedState` and `options` are set by `initialize()`'s arguments.

The `boolean` field `commitSucceeded` and `succeeded` are used to state that these phases are completed successfully. The third `boolean` variable `debug` is retrieved from `options` and it is used to display additional debug information.

Finally, a field that stores `SubjectData` is required, since this reference is set by `login()` and used in `commit()`.

The constructor does nothing, since `initialize()` does the mandatory initialization. That means it sets `debug`, `options`, `sharedState`, `subject` and `callbackHandler`.

`login()` creates a `NameCallback` and a `PasswordCallback` to create a `LocalNamePasswordCallback`. It also creates a `UserDBCallback` and invokes the `CallbackHandler`'s `handle()` methods with these `Callback`s.

After that, a `User` and a `Password` object can be instantiated with the information from the `Callback`s. Since the `UserDBCallback` returns a `UserDB` proxy, the service can be consulted using a `AuthDataImpl`, consisting of the `User` and the `Password`. The result of that invocation, a `SubjectData` is stored. If an `Exception` occurs in `login()`, all the information will be wiped out and an `|LoginException|` will be thrown.

`commit()` uses the `SubjectData` to add the `Principal`s, the public credentials and the private Credentials to the `Subject` field. `logout()` does the reverse operation: it removes the data from the `Subject` field and clears all information, like `abort()` does it.

### 7.13.2 RingLoginModule

`RingLoginModule` is very similar to `ConsolePasswordLoginModule`, so that only the differences need to be stated.

`login()` uses the `LocalChallengeCallback` with a new `ChallengeImpl()` to retrieve the `User` and the `Response`. Then the `UserDB` can be contacted with a new `ChallengeDataImpl`, consisting of the `Challenge`, the `Response` and the `User`. The rest is the same as described above.

### 7.13.3 Proxy

The `JaasProxy` also extends `RegisterImpl` and implements `SubjectAuthenticator`. The fields `remoteSubjectAuthenticator` references the backend service.

The methods `login()`, `logout()`, `ping()` and `refresh()` are responsible to do the sealing, unsealing and call the methods of the backend. If the proxy has not been registered, `register()` is called first.

### 7.13.4 Backend

The `JaasService` is a little bit more complex than the other services. The service extends `RemoteRegisterImpl` and implements the `RemoteSubjectAuthenticator` interface.

FILENAME and LOGIN_DURATION are two constants that specifies where to store the context information and the duration of a lease that is given out for a JAASSlotResultImpl. For that reason the field landlordLeaseFactory is used to create new Leases.

A SlotProvider as a field is also needed, because it contains the JAASSlotResults. Since this service uses two other services, the ServiceFinder fields loginPolicyDBFinder and userDBFinder are needed.

The main() method expects two arguments: the DomainName and the password to access the key store. But before, a new JaasService can be instantiated, a new SunJCE security provider is added to JVM.

Finally, initialize() is called on the JaasService which starts the service. The constructor sets the field for landlordLeaseFactory, the slotProvider and the two ServiceFinders.

getProxy() returns a new instance of JaasProxy as expected.

Since JaasService gives out two different kinds of leases, this implementation overrides the four landlord methods. If the cookie is a instance of LoginCookie, the corresponding method of the slotProvider is called, otherwise the super method is called.

refresh() and logout() are currently not implemented.

The most important method is login(). It gets the two proxies from the ServiceFinder fields and unseals the LoginData. Using the LoginData, the LoginPolicyDB service is contacted and returns a LoginPolicy.

To create a Jaas LoginContext, the name of the LoginPolicy and a CallbackHandler are needed, whereby the last one is a new instance of LocalCallbackHandler that takes the SecretKey, the RemoteCallbackHandler and the UserDB proxy as arguments for the constructor.

After the instantiation of the LoginContext, the login() method can be called and if no Exception has occurred, getSubject() is called. The Subject and the LoginPolicy are bundled into a JaasSlotResultImpl object which is stored into the SlotProvider by calling leaseSlot(). The result of the leaseSlot() invocation and the Subject are bundled as SubjectAuthResultImpl and returned sealed.

## 7.14 Challenge-Response Applet and Host for the Java Ring

### 7.14.1 Applet

The ChResp_Applet extends Applet which is defined in the javacard.framework package. Since the communication only works on byte array level, the Applet itself and all the methods must be specified by byte values.

Therefore, the following constants are defined as proposed by the Dallas Semiconductors IDE: CHRESP_CLA, CHRESP_INS_SET_PIN, CHRESP_INS_SET_KEY, CHRESP_INS_ENCRYPT, CHRESP_INS_SWITCH_PIN, SELECT_CLA, SELECT_INS. The constant MAX_SEND_LENGTH is defined to determine the length of a APDU package.

The Domain, the User are stored as byte arrays. The key is stored as DES3_EncKey. To provide an applet PIN, the fields use_pin, appletPIN, pinSize and lastPINCheckTime are used. To store the data inside the APDU a byte array apduData is defined.

The constructor of the Applet registers itself with the Java Card Environment using register(). Then use_pin is set to true, pinSize is set to zero and a new OwnerPIN is generated. Calling updateAndUnblock() on that PIN, sets the pin. install() just creates a new instance of that Applet.

The method dispatcher is process() which calls one of the four methods, specified by the four constants. The most important method is encryptDispatch(). If use_pin is true checkPin() is called which first checks the time if the Java Ring is blocked and unblocks it if the duration is more than three minutes. The lastPINCheckTime is set to the current time and the pin is checked by calling check on the appletPIN.

The format of the incoming data is length of PIN + PIN + length of DomainName + DomainName + Challenge. The Domain and Challenge are retrieved from that information. First the domain is compared. If it is not equal a ISO.SW_WRONG_DATA Exception is thrown. Then the encryptECB() method is called on the key and stored in ret which is sent back in the end by calling sendByteArray(), whereas the method only sends the byte array in pieces of MAX_SEND_LENGTH.

The method set_keyDispatch() expects the byte array format: length of PIN + PIN + length of Domain + Domain + length of User + User + Key. domain and user are retrieved directly from the byte array, whereas first the key is generated as DES3_EncKey and is set using the setKey() method with the byte array.

set_pinDispatch() expects the byte array format: length of old PIN + old PIN + new PIN. It just calls checkPIN(), sets pinSize and calls updateAndUnblock() with the new PIN on the appletPIN. switch_pinDispatch() expects the byte array format: PIN + boolean. So first the method calls checkPin() and then it sets use_pin by calling booleanFromByteArray() which interprets the byte value as boolean value.

### 7.14.2 Host

The corresponding host program is called ChResp_Host and implements JibMultiListener and Runnable. Since the encryption request is done by RingAuthenticationApplet, this host program is only used to load the keys. The main() method expects four arguments: the jib file path, the key file path, the DomainName and the user-name.

The program is started as Thread. The run() methods only calls Thread.sleep, so that the program cannot leave the JVM.

If iButtonInserted() is invoked, the Applet is selected and set_keyDispatch is called. The method builds a byte array as described above and sends it using the sendAPDU() method. Since no response is requested, ”no response received” is printed.

# Chapter 8

# Case Study

This chapter shows how a simple service can be implemented using the authentication and authorization architecture. The intention is to show how the communication between the service layer and the security layer works. That means the given sample service does not have to be very sophisticated. Its only task is to show two messages. The first message should be shown after the successful authentication. The second one should only be shown if the authenticated user has the permission for it.

## 8.1 Interfaces

The communication protocol between client and service is defined by a Java interface. Since the client does not need to be changed after integrating the security architecture, the interface should also stay the same. That means the interface has no notion about the security architecture:

```
public interface Sample extends Serializable{
   String getMessage();
   String getAuthMessage();
}
```

It only defines the two methods `getMessage()` and `getAuthMessage()` which return a String.

The Sample interface should be implemented by a `SampleProxy`, whereas the `RemoteSample` interface, which defines the communication between the service proxy and the service backend, should be implemented by the backend:

```
public interface RemoteSample extends RemoteAuthenticator{
   SealedMacObject getMessage(SealedCookieMacObject data);
   SealedMacObject getAuthMessage(SealedCookieMacObject data);
}
```

It has to extend `RemoteAuthenticator`, since the proxy has to invoke the `login()` and `logout()` methods. `getMessage()` and `getAuthMessage()` are responsible to secure the insecure communication channel. So they use the `SealedMacObject` as return values and `SealedCookieMacObject` as arguments.

## 8.2 SampleAction

The authorization check is done by calling the `Subject.doAs()` methods with a `Subject` and a `PrivilegedAction` as arguments. Therefore, the actions that need authorization, have to be encapsulated in the `run()` method of an implementation of the `PrivilegedAction` interface:

```
public class SampleAction implements PrivilegedAction {
   Object run(){
      return System.getProperty("user.home");
   }
}
```

The `run()` method is very simple. It only returns a `String` object containing the users home directory.

## 8.3 SampleProxy

Most of the work of the proxy side work is done by the super class `AbstractProxy`, so that the `SampleProxy` only has to extend that class and has to implement the `Sample` interface:

```
public class SampleProxy extends AbstractProxy implements Sample{
   RemoteSample remoteSample;
   SampleProxy(RemoteSample, SignedObject){};
   String getMessage(){};
   String getAuthMessage(){};
}
```

The constructor calls the super constructor with the `RemoteSample` and the `SignedObject` and sets its own `RemoteSample` field.

The `getMessage()` is responsible to seal the request and to unseal the response from the backend which is called by using the `RemoteSample` field:

```
public String getMessage() throws Exception {
    SealedCookieMacObject param =
       authenticator.getSecureClient().sealObject(null);
    SealedMacObject result =
       remoteSample.getMessage(param);
    String ret = (String)authenticator.getSecureClient().
       unsealObject(result);
    return ret;
}
```

Since `getMessage()` needs no parameters, only `null` has to be sealed, using the `sealObject()` method of a secure client which can be retrieved by the super class field `authenticator`. With that sealed object, the appropriate method of the service backend can be called. In the end, the result of the remote method invocation has to be unsealed using the `unseal()` method of the secure client.

`getAuthMessage()` does exactly the same, except the invocation of another method on the backend.

The overall pattern is:

- sealing the parameters
- calling the backend method
- unsealing the result

If the sealing and unsealing is done in the secure socket layer, the backend method can be called directly, because the sealing and unsealing is done implicitly in the secure socket layer.

## 8.4 SampleService

The counterpart of the service proxy is the service backend. The `SampleService` has to extend the `AbstractBackend` which does the sealing among other things and it has to implement the `RemoteSample` interface. It also needs a `main()` method to start the service and an implementation of `getProxy()` to return the appropriate service proxy:

```
class SampleService extends AbstractBackend implements
RemoteSample{
   final static String FILENAME = "SampleService.dat";
   SampleService(String, DomainName, char[]);
   main(String[]);
   Object getProxy();
   SealedMacObject getMessage(SealedCookieMacObject data){};
   SealedMacObject getAuthMessage(SealedCookieMacObject data){};
}
```

The constructor calls the super constructor with the location of the service context data, the `DomainName` and the password to access the key store entry which is retrieved using the `DomainName`.

To start the service, the `main()` method expects two parameters, the `DomainName` and the password for the key store. The third parameter which is needed to instantiate the `SampleService`: the location of the context data is stored in the constant `FILENAME`. After creation of `SampleService`, the `initialize()` method is called:

```
...
SampleService sampleService = new SampleService (FILENAME,
   new DomainNameImpl(args[0]), args[1].toCharArray());
sampleService.initialize();
...
```

`getProxy()` returns an instance of the `SampleProxy`. Therefore it calls the constructor with a reference of itself and the class name of itself:

```
return new SampleProxy(this, getSignedObject(
   this.getClass().getName()));
```

`getMessage()` should return a `String` containing the two words: ”Hello World”. Therefore, `getMessage()` first has to unseal, has to check implicitly the parameter and has to seal the `String`:

```
SecureSlotProvider sessionSlotProvider =
   authenticatorBackend.getSessionSlotProvider();
sessionSlotProvider.unsealObject(data);
return
   sessionSlotProvider.sealObject(data.getCookie(), "Hello World!");
```

The sealing and unsealing is done by a secure slot provider which needs the `Cookie` stored in the parameter to retrieve the context information.

The overall pattern is:

- unseal the parameter
- perform the actions
- seal the result

If the sealing and unsealing is done by the secure socket layer, only the actions have to be performed.

The difference between `getMessage()` and `getAuthMessage()` is that `getAuthMessage()` uses `Subject.doAs()` to perform an action that needs authorization:

```
SecureSlotProvider sessionSlotProvider =
authenticatorBackend.getSessionSlotProvider();

sessionSlotProvider.unsealObject(data); Cookie cookie =

data.getCookie();
SubjectAuthResult subjectAuthResult =
    (SubjectAuthResult)sessionSlotProvider.getData(cookie);
String ret = (String)Subject.doAs(subjectAuthResult.
    getSubject(), new SampleAction());
return sessionSlotProvider.sealObject(cookie, ret);
```

Therefore, it retrieves the `Subject` from the context information stored by the secure slot provider. It also creates a new `SampleAction` that `run()` method returns the current users home directory. The rest is unsealing the parameter and sealing the result.

## 8.5 SampleClient

Finally, a `SampleClient` is needed that uses the `SampleService`. The `SampleClient` is a regular Jini client that uses a `ServiceTemplate` to specify, what Jini service is needed. The whole security architecture is transparent to the client, so that the client only has to call the both methods on the proxy:

```
JOptionPane.showMessageDialog(null, "It's message is:\n" +
((Sample) o).getMessage(), "SampleClient",
JOptionPane.PLAIN_MESSAGE);
JOptionPane.showMessageDialog(null,
"It's 'auth' message is:\n" + ((Sample) o).getAuthMessage(),
"SampleClient", JOptionPane.PLAIN_MESSAGE);
```

## 8.6 Setup the policy files

The setup of the policy files is very crucial in the whole security architecture, because wrong entries could allow attacks. The policy file at client side has to ensure that only service proxies that are signed by trusted parties are executed, whereas the policy file at service backend side is used to check if the current user is authorizated to execute the code. The authentication service also needs some information of how to perform the authentication process.

### 8.6.1 Client

The client only needs one policy file that only grants the appropriate permissions to signed and trusted code. Currently, the `SocketPermission` must be granted to unsigned code, otherwise the `ClassLoader` cannot be instantiated. The key store, which contains the public verifying keys, could also be specified:

```
keystore "file:/${user.home}/.keystore", "JKS";

grant signedBy "john"{
    permission java.security.AllPermission "", "";
};

grant{
    permission java.net.SocketPermission "*:1024-",
    "connect,accept";
};
```

### 8.6.2 Service

The sample service needs two different policy files. One JDK1.2 style police file that grants all local archives the `AllPermission`, except the archives that contain the classes which need authorization. The permissions for these classes are granted in a separate policy file. The first policy file looks like:

```
grant codebase "file:/H:/project/jaas1_0/lib/jaas.jar"{
    permission java.security.AllPermission;
};

...

grant codebase "file:/H:/project/jini1_0_1/lib/sun-util.jar"{
    permission java.security.AllPermission;
};

grant {
    permission java.net.SocketPermission "*:1024-",
    "connect,accept";
};
```

The second policy file grants permission to authorization relevant classes:

```
grant codebase
"file:/H:/project/jaaa/sampleservice/schoch-sample-action.jar",
    Principal schoch.jini.security.reference.auth.PrincipalImpl
    "schoch" {

   permission java.util.PropertyPermission "user.home", "read";
};
```

That entry grants the `PropertyPermission` to all classes inside the jar archive, which are executed by a `Subject` that contains the Principal schoch, to read the `user.home` system property.

### 8.6.3 Authentication Service

The Authentication service itself uses a regular policy file. This `SampleService` uses an authentication service, which uses JAAS, so that also a configuration file for the login policies is needed. The configuration file used in that example only provides one entry `Default` which specifies a console login as sufficient and a ring login as required if the first fails:

```
Default{
  schoch.jini.security.reference.service.jaas.
     ConsolePasswordLoginModule sufficient debug=true;
  schoch.jini.security.reference.service.jaas.
     RingLoginModule required debug=true;
};
```

## 8.7 Key Management and Signing

A few jar archives, including the downloadable `ServiceProxy` have to be signed. First, the private signing key and the public verifying key must be generated. The `DomainName` is used as alias to access the key store. In the given example a user called John Smith will be created. His alias and therefore the `DomainName` is john:

```
> keytool -genkey -alias john -keypass 123456
```

The password to access the private signing key is 123456. The private signing key is used to sign the dynamic DH key and the static code among other things.

All jar archives that are used by the `SampleClient` including the downloadable code, have to be signed:

- `jce1_2_1.jar`
- `sunjce_provider.jar`
- `jaas.jar`
- `jini-core.jar`
- `jini-ext.jar`
- `sun-util.jar`

- schoch-sample-client.jar
- schoch-common-int.jar
- schoch-common-impl.jar
- schoch-security-int.jar
- schoch-security-impl.jar
- schoch-signed-dl.jar

Another approach is to sign only the downloadable code and to grant the local classes the required permissions.

## 8.8 Setup the environment

The easiest way to run the example is to install the whole project in `D:\project\jaaa` under Window NT 4.0 or Windows 2000. If another operating system or another path is used the file `jaaa/template/environment.template` needs to be updated. Especially the paths and variable names have to be updated. If there are changes to `enviroment.template`, `generate-files.bat` must be called to update the batch files which are used to run all the Java programs. If another operating system is used, all files in the template subdirectory have to be updated. If another path is used, all policy files in all subdirectories must be updated.

## 8.9 Loading the key onto the Java Ring

If the `UserDBService` is started, it tries to read the directory `.jiniauth/challresp` in the users home directory. If the directory does not exist, the user has to create the directory first, otherwise the `UserDBService` cannot work correctly. The `UserDBService` tries to read two Triple DES keys from the files `[DomainName]-schoch.secretKey` and `[DomainName]-krone.secretKey`. If the keys do not exist, they will be generated automatically.

To load a Java Ring with the user schoch, `loadkey1.bat` in the `ringauthentication` subdirectory has to be launched. If the program is up, the Java Ring must be pushed onto the blue dot receptor and has to be released. After the program reports that there is no response from the Java Ring, because it is only an upload, and therefore no response is expected, the Ring can be released. To load a Java Ring with the user krone, `loadkey2.bat` has to be launched in the same manner.

## 8.10 Starting the example

The example can be started if the steps described above have been done:

- `environment.template` has been updated and `generate-files.bat` has been called if necessary
- `.jiniauth/challresp` has been created in the users home directory and the UserDB.bat has been started once to create the keys

- the keys have been loaded onto the Java Rings

The first thing is to start the Jini infrastructure in the `jiniinfra` subdirectory:

- `deltmp.bat`, delete the persistent rmid and lus log directories
- `lus-http.bat`, the http server for the lus
- `rmid.bat`, the rmi activation daemon
- `lus.bat`, the lus

Next, a http server for the proxies and the services have to be started:

- `common-dl/common-http.bat`
- `bluedot/bluedot.bat`
- `ringauthentication/ringauthentication.bat`
- `userdb/userdb.bat`
- `loginpolicydb.bat`
- `jaas/jass.bat`
- `sampleservice/[sample`signed]service.bat—
- `sampleclient/sampleclient.bat`

It is possible to start `sampleservice.bat` or `signedservice.bat`. The difference is that `sampleservice.bat` starts a service which downloads a not signed `SampleProxy`, so that the execution fails.

First, the user is prompted for a user-name and password, after all services and the client is up. The password for the user schoch is thomas. The password for the user krone is oliver. If the user-name-password-dialog fails, the Java Ring authentication starts and prompts the user to push the Java Ring onto the blue dot receptor. Only the user schoch has the permission to read the authorizated message. Both users can retrieve the non authorizated message after authentication. If nothing entered at the user-name-password dialog, a null pointer exception will be thrown in that version. If the Ring authentication should be used, something must be entered at the user-name-password-dialog.

# Chapter 9

# Conclusion and Future Work

First, a short overview about Jini related projects is given. Then, conclusions and future work are subject to the following two sections.

## 9.1 Jini and Friends

In the field of standardization these protocols, all mayor players in the computer and home entertainment business are involved. They are trying to build an infrastructure for service discovery and well-defined protocols of how to communicate.

Microsoft and 28 industry partners initiated the Universal Plug and Play (UPnP) initiative in 1999. The UPnP layer provides discovery, description and usage of home applications, but there is currently no implementation available.

Sun Microsytems provides an architecture called Jini instead which is already available. The core concepts of Jini, which have already been explained in detail above, are discovery, lookup, leasing, remote events and transactions for services. Sun and Eko Systems announced in April 2000 one of the first broad-scaled implementations of Jini technology in healthcare. In February 2000, Motorola and Sun announced that they will collaborate on the development of a next-generation architecture to support distributed Command, Control, Communication, Computer and Intelligence (C41) operations for the US Army.

An approach of Sun is to make Jini's underlaying computer language Java the lingua franca for every digital device. Java came on the market in 1995 and was originally intended for embedded systems, but it was heavily used as the Internet computer language, because one of the features is platform independence: "Write once, run anywhere". At the moment it seems that Java returns to its root: the embedded systems. Embedded Java, Personal Java and the K Virtual Machine are products that fit into environments with less resources. So every small device is able to understand Java. Microsoft goes the same way pushing its operating system Windows CE in every small device. Especially palm computers are running Windows CE.

Sun joined some other initiatives in that segment, and put Java support in it:

- HAVi: the Home Audio and Video Initiative (HAVi) is a Consumer Electronics (CE) industry standard that will ensure interoperability between digital audio and video devices from different vendors and brands that are connected via a network in the consumer's home,

- GSM: the mobile phone standard,
- DVB: the digital television standard,
- OSGi: the Open Service Gateway Initiative (OSGi) defines a set of APIs and provides a sample implementation of a service gateway architecture. A service gateway is a focal point, which connects the local network containing the services, to the Internet.

In January 2000 Sun and Cisco formed an alliance to develop an open architecture for an Internet home gateway. Cisco provides a next-generation prototype of a Home Gateway Server, running Sun's Java Embedded Server on it. Sun's Embedded Server itself can provide Jini services among other things. Two other important collaberations in this field, is the development of a .com Home environment in Europe with Telia and the Connected Family environment with GTE and Cisco.

At CES 2000, the Consumer Electronics Show 2000, Sun presented a refrigerator with web panel as a part of the .com Home which already uses the Java and Jini technology.

## 9.2 Conclusion

This thesis showed that it is possible to provide a secure authentication and authorization architecture for Jini services, only using present Java technologies, although there are a few things to do to complete the architecture and the prototype.

Currently, Jini does not provide any security features, since Suns goal was to be the first on the market, so there was not enough time to include a whole security architecture. Sun reached its goal to be the first and still the only one, who provides a very general service brokerage. There are also efforts to put security into Jini. That means to provide a secure RMI with integrity and confidentiality: RMI Security Extension. This extension also builds on JAAS to provide authentication and authorization, but in a very general way. So, the architecture shown in this thesis provides a very easy plug and play approach for service developers to include authentication and authorization right now.

The transparency for the client is worth a discussion. The authors think that it is a good design decision to provide transparency for the client for various reasons. First, the client should not be able to weak the security strength, so that the decision is up to the service. Second, existing clients does not need to be changed and third, it is much simpler if only the service side is involved. Another approach is that the client is aware of the security issues and is able to specify the level of security or both sides can negotiate about the level of security. Since the clients can receive events and the events are secured in some way, the client has to deal with security issues.

## 9.3 Future Work

Currently, the sealing and the unsealing of data is done at application level, i.e. the methods therefore have to be called explicitly. If the sealing is moved to socket level, then the sealing is done implicitly, so that the architecture is almost transparent to service side. If the sealing is done at application level, there is a potentially security hole in the deserialisation process, since the serialisation stream could be altered during

the transmission. Although this potential security hole exists, there is no concrete bad scenario known by the authors.

Another important point is to secure the leases. The implementation given by Sun is of course not secure, so an own implementation using the secure communication classes must be provided. Also, the administration of leases must be completed, so that a lease can be renewed or cancelled. Currently, leases are created, but not really maintained, since this a standard task and therefore not that important to this architecture.

Also, events should be secured using a secure communication channel, since replay or deletion of events on the wire is a heavy attack.

The Jini infrastructure itself was not subject to this architecture, since the authentication process is build on top of the Jini interfaces. It is also imaginable to put security in the core Jini interfaces, but then, the problem of standardization arises.

Since the reference implementation is only made to show, how and if the architecture works in small examples, some services, especially the LoginPolicyDB service and the UserDB service have to be expanded, e.g. they could contact databases using the JDBC interface.

However, since this is the first trial of building a secure architecture, there are still remain many points which can be done better at second glance. Interested parties are welcome to continue the work on this architecture and they may contact the author for that purpose.

## 9.4 Acknowledgements

There are many people I have to thank for supporting me to work on my diploma thesis. First, there is Prof. Mattern from ETH Zürich, who is responsible for my diploma thesis. He also started the procedure that made it possible for me to stay at the International Computer Science Institute in Berkeley, California, by contacting Oliver Krone from SwissCom, who still works at the Institute for one year in all. Oliver Krone looked after this diploma thesis and worked with me in that field. He also established the contact to Sun Microsystems in Cubertino. At the institute, Joachim Beer, the leader of the NSA Group and Lila Finhill, the administrative manager, helped me in the administrative field. I had many discussions with Hannes Federrath from TU Dresden, a security expert at the Institute, about security in general and in my architecture. Also, Bob Scheifler from Sun Microsystems gave me an important feedback on this architecture. Sandra Faul, a colleague in giving private lessons for pupils, at least corrected the most wrong parts of the English grammar. And last but not least, I want to thank all the people at the Institute for that great time I had from February until May 2000.

# Bibliography


[1] W. Keith Edwards. Core Jini. Prentice Hall, 1999.

[2] George Coulouris, Jean Dollimore, Tim Kindberg. Distributed Systems. Concepts and Design. Second Edition. Chapter 16 Security, page 477-516. Addison-Wesley, 1998.

[3] Jay Miller, Kelby Zorgdrager. The Jiro Technology On-Line Tutorial. Series I: Part III. What are Management Components? Sun Microsystems. 2000.

[4] Felix Baessler, Jos Bonnet, Pascal Brisset, Ciara Byrne, Claudio Lobo, Roger Kehr, Peter Keller, Oliver Krone, Susanna Mäkinen, Joachim Posegga, Roland Schmitz, Telmo Silva, Edwin Wiedmer, Peter Windirsch, Frederico Vieira, Pete Young. Eurescom. Jini & Friends @ Work: towards secured service access, Version 0.9. May 2000.

[5] Tish Williams. Java Rings Are A Girl's Best Friend. Upside Today. 1998-03-25.

[6] Bill Day. The Java Platform Scales To Fit Small Spaces. Java World. 1998-03-26.

[7] Beth Dickey, N'Gai Croal, Thomas Hayden. Run Rings Around Your Friends in Cyperscope. Newsweek. 1999-05-24

[8] Casey Cameron, Bill Day. Knuckletop Computing: The Java Ring. Sun Microsystems. 1999-11-15.

[9] David Fiedler. Lord Of The Rings. WebDeveloper. 1998-03-25.

[10] Warren Webb. Electronic stamps lick interest. EDN Access. 1998-08-03.

[11] Rinaldo Di Giorgio. iButtons: The first ready-to-buy 2.0 Java Card API devices. Java World. April 1998.

[12] Steven Meloan. Inside The Java Ring Event. Sun Microsystems. 1999-11-15

[13] Steve Lockwood, Joe Duhamel. The Jewel in the Java Ring. 1998-04-03.

[14] Chuck McManis. Ring fever: A first-hand look at Java-powered jewelry. Java World. April 1998.

[15] Stephen M. Curry. An introduction to the Java Ring. Java World. April 1998.

[16] Dallas Semiconductors. javadoc documentation of Java Ring packages.

[17] Jakob Nielsen. The Java Ring: A Wearable Computer. useit.com. 1998-01-05.

[18] Dallas Semiconducttors. iButton with Java Command Processor. 1999-06-11.

[19] Dallas Semiconductors. APDUSender.

[20] Chuck McManis. My ENIGMAtic Java Ring. Java World. August 1998.

[21] Sun Microsystems. Java Card 2.0 Application Programming Interfaces. Revision 1.0 Final. 1997-10-13.

[22] Jon Byous. Java Technology: An Early History. Sun Microsystems. 2000-02-07.

[23] OpenCard Cosortium. OpenCard Framework 1.2 – Programmer's Guide. Fourth Edition. December 1999.

[24] Thomas Schäck, Rinaldo Di Giorgio. How to write OpenCard services for Java Card applets. Java World. October 1998.

[25] Mike Wendler, Stephan Breidenreich, Rinaldo Di Giorgio. How to write a CardTerminal class for simple and complex readers in an OpenCard environment. Java World. January 1999.

[26] Rinaldo Di Giorgio, Mike Montgomery. Write OpenCard services for downloading Java Card apps. Java World. February 1999.

[27] OpenCard Consortium. OpenCard Framework (OCF): Frequently Asked Questions. 1998-07-14.

[28] OpenCard Consortium. OpenCard Framework – General information Web Document. October 1998.

[29] Sun Microsystems. Java Naming & Directory Interface.

[30] Sun Microsystems. Creating a Custom Socket Type. 1999.

[31] Sun Microsystems. Creating a Custom RMI Socket Factory. 1999.

[32] Hewlett-Packard Company. Cool Town Vision. 2000.

[33] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra. People, Places, Things: Web Presence for the Real World. Hewlett-Packard Company

[34] Rinaldo Di Giorgio. An introduction to the URL programming interface. Java World. September 1999.

[35] Microsoft. Universal Plug and Play. 1999-11-02

[36] Sun Microsystems. Motorola And Sun Collaborate To Dot-Com Next-Generation C4I Systems. 2000-02-10

[37] Sun Microsystems. Sun Microsystems And Eko Systems Announce One Of The First Broad-Scale Implementations Of Jini Technology In Healthcare. 2000-04-10.

[38] Sun Microsystems. K Virtual Machine (KVM). 2000.

[39] Robert D. Hof, Roger Crockett. A Java In Every Pot? Business Week. 1998-07-16.

[40] John Rommel. The Java-enabled wireless world. Java World. March 2000.

[41] Sun Microsystems. Standards at Work. 2000.

[42] Sun Microsystems. Java Technology Adopted As The Standard For A Wide Range Of Consumer Devices Across Multiple Markets. 2000-01-06.

[43] Open Services Gateway Initiative (OSGi). Frequently Asked Questions.

[44] Home Audio Video Interoperability (HAVi). Frequently Asked Questions.

[45] Sun Microsystems. Home Gateway Enables Home Services. 2000.

[46] Sun Microsystems. Sun Microsystems And Cisco Systems Form Alliance To Develop Open Architecture For Internet Home Gateway. 2000-01-06.

[47] Sun Microsystems. Sun Microsystems And Telia To Develop A .Com Home Environment In Europe. 2000-01-06.

[48] Sun Microsystems. Sun Microsystems, GTE And Cisco Systems Create ”Connected Family” Internet-Enabled Products And Technologies. 2000-01-06.

[49] Sun Microsystems. The .Com Home. 2000.

[50] Sun Microsystems. Sun Microsystems’ Scott McNealy Introduces The .Com Home Of The Future At The Consumer Electronics Show. 2000-01-06.

[51] Sun Microsystems. Sony and Sun Join Forces To Link Digital Consumer Electronics Appliances Directly To Internet Content And Services. 1999-11-10.

[52] Roger Kehr, Joachim Posegga, Harald Vogt. PCA: Jini-based Personal Card Assistant.

[53] Michael Rohs, Roger Kehr, Harald Vogt. Slides: Jini-enabled Smartcard Terminal.

[54] Rinaldo Di Giorgio, Mike Montgomery. Write OpenCard services for downloading Java Card apps. February 1999.

[55] Roger Kehr, Michael Rohs, Harald Vogt. Mobile Code As An Enabling Technology for Smartcard-Middleware.

[56] Zhiquin Chen, Rinaldo Di Giorgio. Understanding Java Card 2.0. Java World. March 1998.

[57] ProSyst Software AG. mBedded Server. Whitepaper. 2000

[58] Eric Freeman, Susanne Hupfer. Make room for JavaSpaces, Part 1. Java World. November 1999.

[59] Eric Freeman. Make room for JavaSpaces, Part 2. Java World. January 2000.

[60] Susanne Hupfer. Make room for JavaSpaces, Part 3. Java World. March 2000.

[61] Roger Kehr, Andreas Zeidler, Harald Vogt. Towards a Generic Proxy Executive Service for Small Devices. FuseNetDE Workshop Position Paper. October 1999.

[62] Pascal Briset. Real Genies Live in Lightbulls: The Case for a Lightweight Jini Registration Protocol. February 2000.

[63] Gerd Aschemann, Roger Kehr, Andreas Zeidler. A Jini-based Gateway Architecture for Mobile Devices. JIT '99. Springer Verlag, 1999.

[64] Gerd Aschemann, Svetlana Domnitcheva, Peer Hasselmeyer, Roger Kehr, Andreas Zeidler. A Framework for the Integration of Legacy Devices into a Jini Management Federation. DSCOM '99. Springer-Verlag, 1999.

[65] Gerd Aschemann, Roger Kehr, Andreas Zeidler. Slides: Jini Shortcomings. 2. Jini Workshop, Zrich. 1999-11-22.

[66] Verlag Heise. Jini im Flachmann. heise online news. 2000-02-25.

[67] Bill Venners. How to attach a user interface to a Jini service. Java World. October 1999.

[68] Hannes Federrath. Slides: Mathematische Grundlagen asymmetrischer Systeme. Draft. 2000.

[69] CRYPTOCard. Homepage. 1999.

[70] Denis Boder. pam_cryptg PAM module.

[71] Bob Bosen. Positive User Authentication with Challenge/Response Super-Smart Cards. Secure Computing Corporation. 1996.

[72] Secure Computing Corporation. White Paper: Virtual Smart Card Solution. 2000.

[73] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, Randy H. Katz. An Architecture for a Secure Service Discovery Service. Mobicom '99 Seatle. ACM, 1999.

[74] Bob Bosen. Internet Encryption, Node Authentication, and User Authentication. Secure Computing Corporation. 1996.

[75] Sun Microsystems. Java Authentication and Authorization Service (JAAS) 1.0. Developer's Guide. 2000-01-14.

[76] Sun Microsystems. Java Authentication and Authorization Service (JAAS) 1.0. LoginModule Developer's Guide. 2000-01-12.

[77] Sun Microsystems. Java Authentication and Authorization Service (JAAS) 1.0. Frequently Asked Questions. 2000-01-12.

[78] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, Roland Schemers. User Authentication and Authorization in the Java Platform. Published in the Proceedings of the 15th Annual Computer Security Application Conference, Phoenix, AZ, December 1999.

[79] Vipin Samar, Charlie Lai. Making Login Services Independent of Authentication Technologies. Sun Microsystems.

[80] Sun Microsystems. Java Cryptography Extension 1.2.1 API Specification & Reference. 2000-02-03.

[81] Sun Microsystems. keytool - Key and Certificate Management Tool.

[82] Sun Microsystems. Java Cryptography Architecture API Specification & Reference. 1998-10-30.

[83] Li Gong. Java Security Architecture (JDK1.2). Version 1.0. Sun Microsystems. 1998-10-02.

[84] Sun Microsytems. Java Remote Methos Invocation Specification. Revision 1.50, JDK 1.2, October 1998.

[85] Sun Microsystems. Java Object Serialization Specification. Revision 1.43, JDK 1.2, 1998-11-30.

[86] Sun Microsystems. Default Policy Implementation and Policy File Syntax. 1998-10-30.

[87] Sun Microsystems. Java Remote Method Invocation Security Extension. Early Look Draft 2. 1999-09-24.

[88] Sun Microsystems. How to Implement a Provider for the Java Cryptography Architecture. 1998-09-21.

[89] Sun Microsystems. X.509 Certificates and Certificate Revocation List (CRLs). 1998-05-20.

[90] Sun Microsystems. Security Managers and JDK 1.2. 1998-10-19.

[91] Sun Microsystems. API for Privileged Blocks. 1998-09-22.

[92] Sun Microsystems. Permissions in the Java 2 SDK. 1998-10-30.