FRÉDÉRIC SAMSON

ALTERNATIVE JAVA SECURITY POLICY MODEL

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en informatique
pour l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

SEPTEMBRE 2004

Résumé

Récemment, les systèmes distribués sont devenus une catégorie fondamentale de systèmes informatiques. Par conséquent, leur sécurité est devenue essentielle. La recherche décrite dans ce document vise à apporter un éclaircissement sur leurs vulnérabilités quant à la sécurité.

Pour ce faire, on a examiné les propriétés de sécurité qu'un système distribué considéré sécuritaire doit supporter. En cherchant un système avec lequel travailler, on a étudié des failles de sécurité des systèmes distribués existants. On a étudié la sécurité de Java et des outils utilisés pour sécuriser ces systèmes.

Suite à ces recherches, un nouveau modèle de sécurité Java imposant de nouvelles propriétés de sécurité a été développé.

Ce document commence par les résultats de notre recherche sur les systèmes distribués, les outils de sécurité, et la sécurité de Java. Ensuite, on décrit les détails du nouveau système pour finalement faire la démonstration des améliorations qu'apporte ce système avec un exemple.

Abstract

Recently, distributed systems have become a fundamental type of computer system. Because of this, their security is essential. The research described in this document aimed to find their weaknesses and to find the means to improve them with regards to their security.

To do that, we examined the security properties that a system considered secure must support. While looking for a system with which we could work, we studied security problems in existing distributed systems. We studied the security of Java and some tools used to secure these systems.

Following our research, we developed a new Java security model, which imposed new security properties.

This document begins with the results of our research in distributed systems, security tools, and Java security. Next, we go into detail about our new system to finally demonstrate the security enhancements of our system using an example.

Je dédie ce mémoire à ma mère, sans qui ce travail n'aurait même pas pu commencer...

Acknowledgments

I would like to thank my research supervisor Pr. Nadia Tawbi for all her guidance and advice throughout the course of my research. I would also like to thank the reviewers of this thesis Pr. Mourad Debbabi (Concordia University) and Pr. Mohamed Mejri (Laval University).

I would also like to extend my gratitude to all the members of the LSFM group at Laval University who worked with me on this project. I learned a lot from working with Dani Nassour. His insights and ideas have helped me to better understand all the subjects discussed in this thesis. The participation of Simon Cloutier and Raphaël Khoury also greatly to helped make this project a success. Working with these three people as well as with Pr. Nadia Tawbi has been a great and rewarding experience.

Finally, this thesis would not even exist if it weren't for the constant love, support, and encouragement from my parents. I cannot begin to thank them enough. Without them, none of this would have been possible.

The research reported in this document has been supported by the Research and Innovation Centre of Alcatel Canada located in Kanata, Ontario.

Contents

List of Figures

Chapter 1

Introduction

This document is the result of a research on the security of computer systems. We concentrate ourselves on the security of distributed systems because they are a very important part of computer research today. The use of distributed systems has increased a lot and will continue to do so therefore making them more secure is very important. We also concentrate on those distributed systems that use the Java programming language [13, 15, 24].

1.1 Motivations

Distributed systems have become very important in the world that is becoming more and more interconnected. As their use has increased, the need for security for those systems has become crucial. It is in this context that distributed systems are created and this document explains our work in improving their security.

The work that we did in this research can be divided into four parts:

- Learn what a secure distributed system is compared to a nonsecure distributed system.
- Study the various distributed systems to find out how they work and what security properties they support.
- Look for different tools that can help us make a distributed system secure.

- Find a way to use those tools to make a distributed system secure.

In the first part, we looked at different security properties that must be satisfied by a distributed system in order to be considered secure. We decided which security properties that in our opinion should be enforced by a secure distributed system.

We then looked at a few distributed systems to find out how they work. We were looking for a distributed system that we could use in our research to make it more secure. We had two main criteria when studying them:

- The distributed system must support applications that were created using the Java programming language.
- The distributed system must support the client-server type of network.

These two criteria are very important. They are used to judge whether we can or cannot use certain distributed systems. We were required to use Java because the language was created with security in mind. This means that applications programmed in Java inherit the security features already present in the language. Our distributed system therefore already contains some security features because it is written in Java.

We also needed a client-server network because we were looking for a type of network commonly in use today where servers offer services to a series of clients. As described in this document, some types of networks are not client-server but they are still distributed systems. In some research, it may be interesting to work with them but for our purposes, we needed a client-server network.

Looking at tools that can help us to make distributed systems secure, we found that there are many interesting ones. We learned what they are and how we use them so that we can create a system that will make the chosen distributed system more secure. We also looked in specification languages for ways to make our system secure.

The next step was to find out what security properties Java enforces and how it enforces them. We know that Java is a secure language but we need to know how secure it is and how it works to enforce its security properties.

Once we understand how Java works to enforce security, we can look into how we can modify its security mechanisms to add support for our security properties. This is the main part of the research project. We created a system that significantly improves

the security of distributed systems that work using the Java language. We improve the enforcement of the security properties already present in Java while adding support for new security properties.

In summary, this work is meant to be a contribution to the understanding of problems related to security in the context of distributed systems. This will be the subject of the second and third chapters where we look at the results of our research on the current state of security in distributed systems and in Java. We then use this information to contribute a solution to some of the problems that are described in those chapters. Finally, to show the capabilities of our solution, we apply it on a concerte example.

1.2 Overview of the Document

The rest of this document is structured as follows.

- Chapter 2 contains the discription of related work. This is the result of our studies on the different security properties that a distributed system must support as well as a description of the distributed systems that we studied, and a list of tools used to enforce our security properties. It also explains our choices on which distributed system we decided to use and on which tools we implemented in our system.
- Chapter 3 explains how the security of the Java language has evolved since it was first introduced in 1995. This chapter concentrates on how security works in the language today from the syntax of the Java policy files to the Java access control mechanisms.
- Chapter 4 is a description of the system that we developed to make our distributed system more secure. The system that we created enforces all the previous security properties already present in Java as well as the new security properties that we decided were necessary to put in a secure distributed system. We also describe the application that we built to demonstrate the capabilities of the system.

We also added an annex that is a description of the syntax of the policy files that are used in our system.

Chapter 2

Related Work

The first part of our research was to explore various distributed systems before making a recommendation on which one to use in our system. Java is becoming a language with major importance in a highly distributed and interconnected world. Furthermore, it was designed with security in mind. Consequently, our system is designed in order to support this language. Once we decide which type of distributed system to use in our system, we wish to add some new security properties to it. The security properties are explained first in this chapter. The first part in improving the security of a distributed system is to study various security properties to know what they mean and to decide if it is important or not to enforce them in a secure system. The authentication, authorization, confidentiality, and data integrity security properties were chosen to be implemented in the system.

The Java language already has some support for these security properties. As explained in Chapter 3, the support for these properties was unsatisfactory. Java only supports the authorization security property and only enforces it in a positive way. This means that administrators of resources can only tell the system which actions are permitted, never which actions are not permitted. How the enforcement of security properties in Java was improved is explained in Chapter 4, but first we look at a definition for each security property.

The second part of this section describes the various distributed systems that were studied. In this research, we started by studying different distributed systems to finally choose one to improve its security. We looked at Jini, CORBA, Microsoft .NET, and JESSICA2. We looked at what their weaknesses are. We looked at what differentiates the distributed systems and what they all have in common.

One thing to note immediately is that they are not all distributed systems of the same type. Jini, CORBA, and Microsoft .NET are distributed systems that offer a more traditional type of network based on clients and servers. Servers offer services on the network and clients communicate with those servers to make requests, which servers deliver. JESSICA2 is a distributed system that emphasizes more on parallel computing and load balancing. Load balancing is the process of spreading the execution of programs or processes across a network to make their execution parallel. When possible, the execution of a program is divided into different parts and they are sent to different points on the network that execute their part and send the result back to a central server that supervises the execution of the program. This makes the execution of the applications more efficient.

To secure the chosen distribued system, different tools were also studied and that is the subject of the third part of this section. Tools like digital signatures, encryption, and SSL were studied so that we could use them in our developed system. We also studied specification languages that describe methods to make distributed systems more secure.

2.1 Important Security Properties

This section defines the four main security properties that we wished to implement to secure a distributed system. We chose these four properties because by ensuring them, a distributed system can be considered very secure. We also felt that it was possible to implement those security properties in the time we had. In the next chapters, we go into more detail on how we implemented these security properties.

2.1.1 Authentication

Authentication is the process of proving a user's identity. Typically, a server and a client are communicating across a network and before any kind of sensitive information can be exchanged between the two, they both need to know exactly with who they are communicating. To do that, they perform authentication on each other. The client proves its identity to the server and the server proves its identity to the client. After this, they can both decide if they actually want to communicate with the other or not. In our system, we used the SSL protocol to perform secure authentication on the network. SSL was created by Netscape in 1994 and is now widely used to perform secure authentication. More information on SSL and how we have implemented the

authentication security property is given in Chapter 4.

2.1.2 Authorization

Authorization is the process of giving a client or a service permission to perform a specific action like executing a piece of code or accessing certain data. On a network, a client may be trying to access some data on a server. The client and the server begin by performing authentication on each other as explained in the previous section. Following that, the client requests to perform a certain action. The server then checks that the client is in fact permitted to perform this action. If it finds that the permission has been given then the server lets the client execute the desired operation. If the server finds that the permission has not been given then the server tells the client that the request has been denied and the sensitive operation is not executed.

The information used by the server to decide if the client has the permission or not is normally the security policy of the server. The security policy is normally written in a text file by an administrator that either owns the server or at least has some kind of priviledges on the server to let him or her control access of the information or programs contained by the server. The system that we developped uses the Extensible Markup Language (XML) to write the security policies. XML is a language used to write structured documents. More information on XML and on how we used XML to implement the authorization security property is in Chapter 4.

2.1.3 Confidentiality

Confidentiality is the security property related to protecting data from being read by unauthorized users. Data must be protected from being compromised. In our system, the encryption of data using public and private keys is used to ensure the confidentiality security property. Users who do not know the related public and private keys cannot read the encrypted data.

2.1.4 Data Integrity

Data integrity is the security property that guarantees that data that is read is valid and that it has not been modified by unauthorized users. If an unauthorized user

has somehow managed to modify the data, then the data has been compromized and cannot be considered valid. To prevent data from being modified, a system must prevent unauthorized users from accessing the data. The authorization security property does this but this is not enough. We need a way that proves to users that are reading the data that the data is valid. This is where digital signatures can be used. Digital signatures are part of the tools explained in the following sections. They sign the data and when a user is reading the data he or she can verify that the data is valid using the digital signature. More information on this is given later.

2.2 Access Control Mechanisms

We first examine different models of access control that have been developed. This research helps us to understand how access control can be performed in computer systems that we could create. We look at the access control matrix model, which uses a matrix to describe access rights. Next, we study the Bell-LaPadula Model, which gives to the objects of the system different security classifications and attempts to prevent objects from accessing other objects that have higher classication than them. Finally, we look at the Chinese Wall Model, which is an access control mechanism aimed at preventing conflict of interests.

2.2.1 Access Control Matrix

The access control matrix is the simplest framework for describing a protection system [4] and it can express any expressible security policy. This section is an introduction to this access control mechanism.

Before looking at the access control matrix, we must define a few terms. We refer to the **state** of the system as the collection of all the values in memory at the present time. A subset of this is the **protection state**, which refers only to the part of the state that is relevent to the protection of the system. Using this, we can define the access control matrix as a model used to describe a protection state. It is the most precise model that can do this. It describes the rights of subjects (a user or a process in the system) with respect to all the entities of the system. For example, we can describe a process using the access control matrix so that we can form a **specification**. We then compare the current state of the system with the specification and this lets us see if we are in an allowable state or not. Many different ways of creating specifications

have been developed.

As the system evolves, changes to the state of the system as well as to the protection state are made. The changes occur during **state transitions** and only certain state transitions are permitted. Those state transitions that are permitted are part of a set of allowed operations. The allowed operations vary depending on the protection state that the system is in at the moment. Performing a state transition on a protection state using an operation that is part of the allowable operations of that protection state is considered secure. This is transforming a protection state into another protection state. Finally, we only consider the state transitions that affect the protection state of the system and not the other transitions. Other state transitions do not need to be considered because they do not change the protection state of the system and therefore cannot violate any security property.

Definition of the Access Control Matrix

The access control matrix is the simplest framework to describe a protection system. This section defines the model and several entities that are part of the model. We first define the set O, which describes the entities of the system that are relevant to the protection state of the system. All the active processes and users are part of the set S. The access control matrix model captures those entities in a matrix A. The rights are part of the matrix and are referred to by $a[s, o]$, where a is a matrix, $s \subseteq S$, and $o \subseteq O$. All the rights of the matrix are a subset of a set R that contains all the possible rights that could be given in the system.

	notes.txt	lab.txt	Process 1	Process 2
Process 1	Read; write	read	Read, write, execute, own	Write
Process 2	append	Read, write	read	Read, write, execute, own

Figure 2.1: An Example of the Access Control Matrix

Figure 2.1 shows a simple example of an access control matrix. The matrix $a[s, o]$ contains two processes, process 1 and process 2 as well as two files, notes.txt and lab.txt. The subject s has the rights $a[s, o]$ for the object o. For example, the subject process 1 has the rights read and write for the object notes.txt. We say that $a[process1, notes.txt] = read, write$.

Conclusion on the Access Control Matrix

This section briefly introduced the access control matrix, an access control mechanism which is a primary abstraction mechanism in computer security. Any expressible security policy can be expressed using this model. However, this model is not used in practice because of the enormous space requirements needed to completely represent all the access rights of a system. Most systems have so many subjects and objects that they cannot be represented in such a system. We can still use the model in computer security research to represent in a simple way some security problems. Similar access control methods are described in Section 2.6.1, where we introduce access control lists (ACL).

2.2.2 Bell-LaPadula Model

The Bell-LaPadula Model is an access control mechanism that gives to system objects different classifications and only lets the system objects access other objects that have the same classification or a lower classification than them [4]. This section is a brief introduction to the model.

All objects, including subjects, have **security clearances**. For example, security clearances could be *unclassified*, *confidential*, *secret*, and *top secret*. These clearances are self-explanatory and obviously the secrecy of data marked "unclassified" is less important than the secrecy of data marked "confidential".

The subjects of the system contain the same security clearances. For example, a subject named Alice could have the security clearance "top secret" while a subject named Bob could have the security clearance "confidential". When Bob attempts to read an object, the security clearance of that object is verified. If it is the same security clearance or if it is a lower security clearance, then the read action is permitted. Otherwise, it is denied. Because Alice has the highest security clearance, she would be permitted to read all objects. Bob would only be permitted to read objects that have a security clearance of either "confidential" or "unclassified".

As in the access control matrix, the notion of protection states is present in the Bell-LaPadula Model. A system moves from protection state to protection state as different actions are taken by the subjects. A protection state is defined as any system state that is secure and a secure system state is one where only permitted types of accesses can be made by subjects on objects in accordance with the security policy. This means that

no one with a specific security clearance can read objects that have a higher security clearance. Also, subjects cannot write data and give it a security clearance that is lower than their own. This is to prevent a subject from taking data that is "top secret" and putting it in a file that is "unclassified" thereby giving preactically all subjects the possibility to read this data.

Categories in the Bell-LaPadula Model

We can expand this model by creating **categories** to each security classification. Each category represents a type of information. For example categories such as FAC, DEP, and PRO could exist and different objects are placed in the different categories with objects having the possibility of being in more than one category at a time.

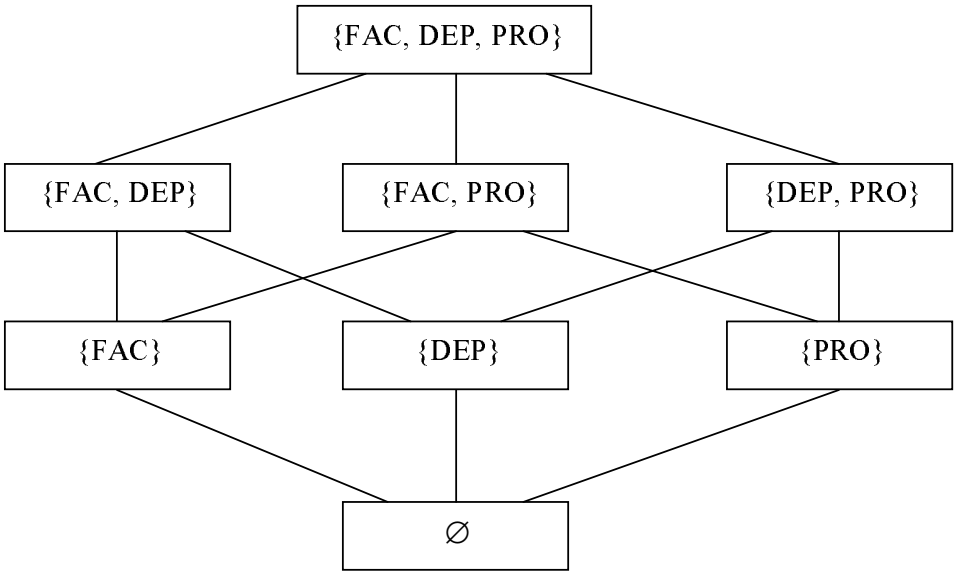

Figure 2.2: Examples of Categories in the Bell-LaPadula Model

From the categories, we work with the "need to know" principle, which states that no subject should be able to read an object unless it is vital to the performing of its operations. Subjects are given a series of categories that they can read from. These are a subset of all the categories. For example, if the categories are $C = \{FAC, DEP, PRO\}$ then a subject Alice can have access to a subset of this, for example $\{FAC, PRO\}$ and Bob can have access to another subset of this, for example $\{DEP, PRO\}$. The categories form a graph where sets are subsets of other sets as shown in Figure 2.2. Subjects have access to the power set of the set of categories. In the example above, Bob has access to the subset $\{DEP, PRO\}$ and therefore he would have access to the following categories (subsets) $\{DEP\}$, $\{PRO\}$, and $\{DEP, PRO\}$.

The categories form a **security level** and subjects and objects are at security levels. For example, Bob is at the security level $(CONFIDENTIAL, \{DEP, PRO\})$ and an object could be at the security level $(TOP\ SECRET, \{FAC\})$.

The introduction of categories changes how the authorization verification process works. Since categories work on the "need to know" principle, we can assume that a subject with access to the categories $\{DEP, PRO\}$ has no need to access objects in the category $\{FAC\}$. Therefore, even if the subject tries to access an object for which he or she has the security clearance, access will still be denied. However, if the object is either in the category $\emptyset$, $\{DEP\}$, $\{PRO\}$, or $\{DEP, PRO\}$ then access can be permitted as long as the security clearance also permits it.

Conclusion on the Bell-LaPadula Model

This section introduced the Bell-LaPadula Model, an access control mechanism based on security clearances. We first looked at the basics of the model with security clearances. Objects and subjects represented using this model are given security clearances such as "unclassified" and "confidential". The model denies subjects from reading objects that have a security clearance that is higher than their own. It also denies subjects from writing to objects and giving them a security clearance that is lower than their own.

We then looked at one way to expand this model, which uses the addition of categories. Now subjects and objects are part of categories and for a subject to be able to read an object it must not only have a security clearance that is the same or lower than his but it must also be part of the same category. This also introduced the "need to know" principle, which says that a subject is only permitted access if it absolutely necessary for the correct execution of its actions.

2.2.3 Chinese Wall Model

The Chinese Wall Model is an access control mechanism that refers to both confidentiality and integrity [4]. The model is well adapted to the business world where we must describe policies that could involve conflict of interest. We define three levels of significance:

1. We first consider each individual items of information. Each of them concerns a single corporation. The files, which contain this information are called objects.

2. We group all objects which concern the same corporation together. This is called the company dataset.

3. Finally, we group together all company datasets, which have corporations that are in competition. This is called the conflict of interest class.

The goal of the Chinese Wall Model is to prevent a conflict of interest. A conflict of interest could occur for example when a trader in the stock market represents two clients. If the best interests of those two clients are in conflict then the only way a trader could help one of the clients is at the expense of the other. This model aims to prevent this type of situation.

To perform the access control, the model puts different company datasets (CD) that must be separated in different conflict of interests classes (COI). Figure 2.3 shows an example of this. There are two different COI classes, which each contain four CD classes.

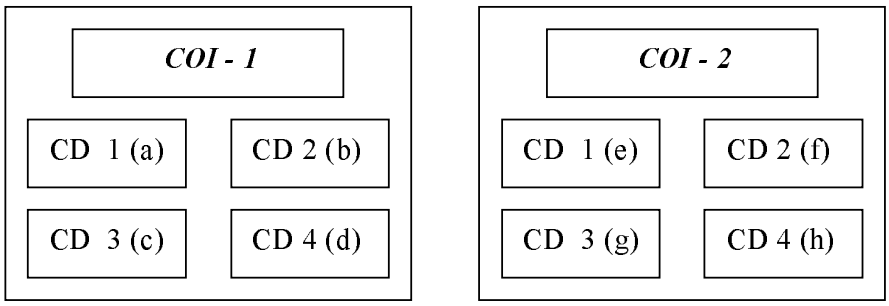

Figure 2.3: Examples of Conflict of Interest Classes (COI)

A subject cannot access two CD classes that are in the same COI class. For example, user Alice cannot access both CD 1 (a) and CD 2 (b) because they are in the same COI class. This system will not allow this type of access. She can however access CD 1 (a) and CD 1 (e) because they are in different COI classes.

We must also consider that Alice should not be able to work with CD 2 (b) after having worked with CD 1 (a) because we can safely assume that she could use the information gained by first working with CD 1 (a) to work against CD 2 (b). The model consideres this temporal element where one user cannot work with a company after having worked with another company of the same COI.

At first, users are permitted to access any CD classes. Then, when a subject reads one object in a COI class then the only other objects of that COI class that can be read

are of the same CD class. This also fixes a minimum number of subjects that must exist. If a COI has four CD classes, then there must be at least four different subjects to be able to read all the objects of the COI class. The model also considers that there is data that does not have to be confidential. That data can be released to the public and no restrictions are put on it.

Conclusion on the Chinese Wall Model

The Chinese Wall Model is an access control model that takes into account both integrity and confidentiality. It controls the access of information by not letting subjects access data that could put them at a conflict of interest. This model is especially useful in the business world where conflicts of interest can easily occur. Objects related to a company are put in different company dataset (CD) objects and the CD objects are put into conflict of interest (COI) objects. The model performs access control by saying that once a subject has read one CD object, he or she cannot read objects from other CD objects in the same COI object. This prevents users from accessing data from other companies that could put them at a conflict of interest.

2.2.4 Conclusion on Access Control Mechanisms

This section introduced three different access control mechanisms. Firstly, we looked at the access control matrix, an abstraction mechanism that in theory can express any expressible security policy. The idea is to create a table of access control information. The table describes all the rights of users or processes in a system. For example, the table could contain a row describing the rights of user "Bob" and a column describing the rights on the file "text.txt". When Bob attempts a senstive operation on the file text.txt, the system finds the rights associated with Bob on that file. Bob's rights on this file are found at the intersection of Bob's row and the file's column. If the attempted operation is found here, then the operation is permitted to be executed. Otherwise, it is denied.

The access control matrix is similar to access control lists that are described in Section 2.6.1. In practice, the model is difficult to use because the number of subjects and objects can grow to make it impossible to store all the information and to look up the information efficiently.

Secondly, we explored the Bell-LaPadula Model. This model is especially suited for

use in the military by giving a series of security clearances to subjects and objects to perform access control. For example, subjects and objects could be given the following clearances, *classified* and *top secret*, with top secret being a higher clearance than classified. A subject given the clearance top secret is permitted to read objects with a top secret clearance and objects with a classified clearance. A subject with the clearance classified is only permitted to read objects with a classified clearance. If the subject attempts to read an object with a higher clearance, it is denied.

When a read access is attempted, the security clearances are compared to perform access control. The write operation is also controlled in the same way by only letting subjects write to objects that have a security clearance that is higher than their own. We also looked at how to expand the Bell-LaPadula Model using categories.

Lastly, we looked at the Chinese Wall Model. This model aims to prevent conflict of interests in the business world by restricting subjects from accessing data that could put the subjects at a conflict of interest. To do that, it divides objects in conflict of interest (COI) and company dataset (CD) objects. Subjects, once they have accessed a CD object of a COI object cannot access other CD objects of the same COI object. They can however access other CD objects in other COI objects.

To create CD and COI objects, the administrator of a system can decide where conflict of interests can occur. One example is when a stock market broker is working for two clients and that the only way to help one client is at the expense of another. This is a conflit of interest. The Chinese Wall Model can be used in this case to prevent the conflict of interest by putting the two clients in the same COI object. By doing that, the broker cannot access data of both clients.

2.3 Distributed Systems

The next step is the study of distributed systems. A distributed system refers to a series of computer systems located at multiple locations working together in a cooperative fashion to either offer different services to clients or to work together to accomplish a specific task. We looked for a distributed system that we could make more secure. One important criterion in our decision on which type of system to use was that the chosen distributed system must support the Java programming language. We know that Java is a secure programming language. It was built with security in mind and has been tested and improved over the years. Java also provides efficient support for mobile code, something that is very interesting for distributed systems. Lastly, in a

distributed system, we should not expect all the different participants to use the same types of computers. Java is portable across different computer platforms and operating systems so it was a good choice.

We also looked for a traditional type of distributed system. This is the client-server model where clients make requests on servers, which respond to those requests. Most of the studied distribued systems follow this model but not all of them. In total, four distributed systems were studied. They are Jini, CORBA, Microsoft. NET, and JESSICA2.

2.3.1 Jini

Jini is developed by Sun Microsystems, the same company that created Java, and was first introduced in January 1999 [3, 18, 23]. Jini seemed to be an interesting distributed system because it is written in Java and clients and servers that use Jini can be written in Java. This is interesting because it means that Jini inherits all the security features of the language. Studying Jini, we found out that while it does have the security features of Java, it is still not secure enough. For example, it is lacking some security properties such as authentication. Data that travels across a Jini network is not necessarily encrypted, which means that the confidentiality security property is not always ensured.

This section looks into Jini. We go into detail on what it is, what it can do, and how it does it. We studied every aspect of the distributed system and the results of our studies are given here. The lookup service, proxies, leasing, and transactions are explained in this section.

An Introduction to Jini

Jini is a network technology. This means that it offers to developers the possibility of writing applications that are to be used by users that are located at a remote location relative to the place where the application is running. In Jini however, we can refer to these applications as being centralized. This is because there exists a sort of central server that contains the links to all the available services of the network. We can say that Jini offers a federation of applications that work together to offer services to the Jini community.

Another aspect of Jini is that it was created to try to limit user intervention in configuring and using applications. One of the goals of Jini is to have things work right away instead of having to take a lot of time to configure the systems before using them. As soon as a Jini application is connected on the network, it is available for the others on the Jini community for use. Users of Jini networks need no prior knowledge of the implementation of services to use them. The services are loaded dynamically with no configuration from the user.

For example, an administrator can add a printer to the Jini network. On traditional networks, this may require each user to install drivers to be able to use the printer. With Jini, the user sees a new printer on the network and automatically, the system downloads the necessary code called a proxy and lets the user use the printer with no need for configuration.

Jini makes no distinction between hardware and software applications. The user who connects to a hardware or to a software sees no difference.

As we will see later, Jini offers a way for the network to avoid trying to connect clients to dead servers. Dead servers are servers that are supposed to be connected on the network and offer services to clients but for some reason are unable to respond. By using leasing, the central server knows that the servers of the network were alive very recently and can continuously verify that they still are. When the printer added to the network above is disconnected, by accident or not, the Jini central server will automatically find out very soon and stop sending clients to this printer. This makes Jini a very robust distributed system.

The Lookup Service

Jini is a distributed system that offers services to clients in a centralized way. The lookup service is the central service in Jini [18]. It is used by every participant on the network. Servers use it to advertize their services while clients use it to find out which services exist that can answer their requests. The lookup service is the heart of the Jini network. All Jini services register themselves on a lookup service and all Jini clients use the lookup service to find services.

The lookup service contains a series of service registrations that each correspond to a service. Normally, the first step in creating a network that uses Jini is to start a lookup service. Following that, the services are started and automatically they search and find a lookup service and register themselves on it. Next, clients are started. They

also look for the lookup service. When they find it, they download the service registry and the service proxy (code used to communicate with the service). Using that proxy, they can create requests and send them to the service and wait for the answer. The order of the steps can be different.

The lookup service also contains information on the services. Earlier, we gave an example of an administrator adding a printer to the Jini network. The printer can have different properties such as the size of the pages that it can use or if it is a colour printer or not. These properties can be found on the lookup service for the client to have information about the service before it uses it. We can even use Jini to let the lookup service decide which printer the user needs for this particular document. If the document does not contain colour, the lookup service sends the proxy of the printer that cannot print in colour. If the document does have colour then it sends the proxy of the printer that can print in colour.

More than one lookup service should be present on the network at a time for security reasons. If the lookup service should fail for some reason, the network becomes unusable. Redundancy helps prevent this problem. If there are more than one lookup service, then the services must register with all of them.

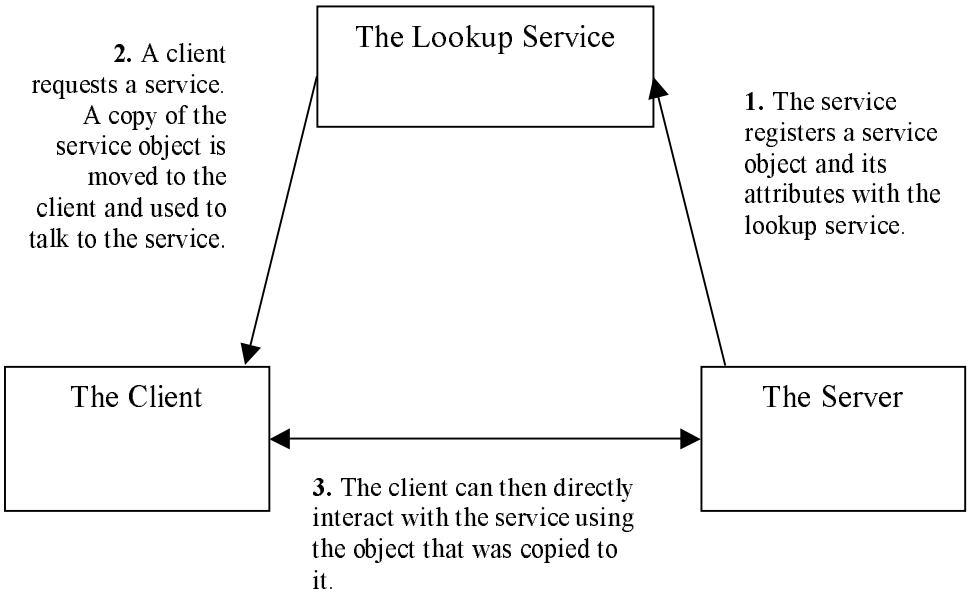

Figure 2.4: The Jini Lookup Service

Finding the Lookup Service

There must be a way for clients and services to find lookup services. Each client and service on a Jini network is required to connect to every lookup service available. To find a lookup service, a service or a client uses a process called lookup discovery. There are two types of lookup discovery, multicast discovery and unicast discovery. The multicast discovery finds lookup services without any knowledge of where they could be located on the network while the unicast discovery knows where a lookup service is located and simply connects to it.

The Multicast Discovery

The lookup service monitors the network for a certain type of packet. This is the multicast request. When a client or a server is looking for a lookup service using the multicast discovery process, it sends a certain packet across the network that the lookup service will pick up and respond to. The service or client receives this response and now knows where the lookup service is located and can use it accordingly.

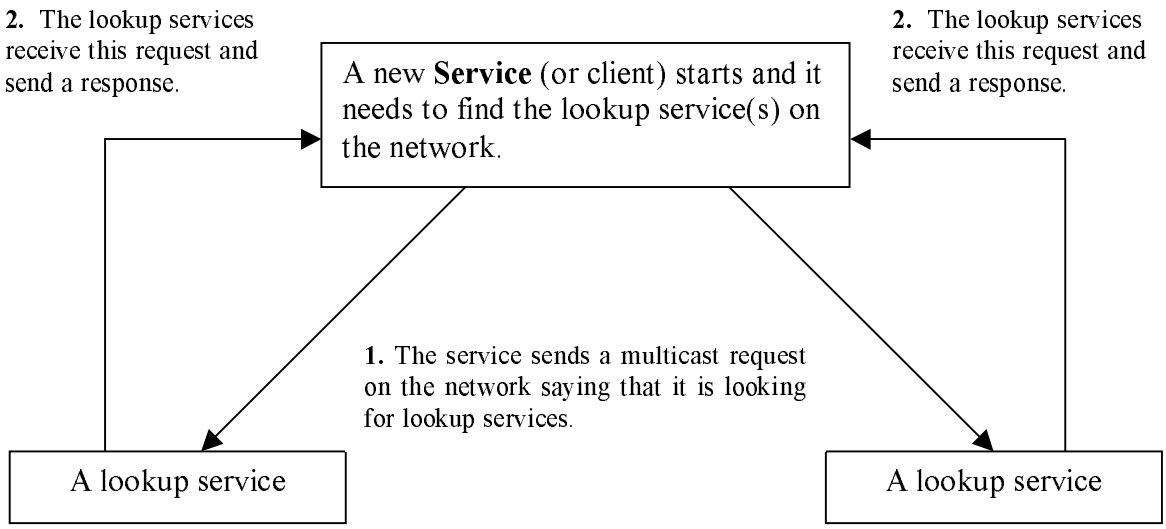

Figure 2.5: How The Lookup Service Multicast Discovery Works

The Unicast Discovery

For the unicast discovery, the client or the service already knows the location of the lookup service so it simply needs to connect to it. It sends a message to the lookup service, which acknowledges its presence. Following this, the client or service can use the lookup service accordingly. This is less effective than a multicast discovery because the clients and servers must know ahead of time the address of the lookup service. Nevertheless, it can still be useful in certain cases and reduces the use of the network to let other types of communication take place. It is also much faster for a client and a server to connect to a lookup service using unicast than by using multicast as long as they are sure that there is a lookup service at this address.

Leasing

When registering on a lookup service, a server also negociates a lease with the lookup service. The lease is an amount of time that a service can guarantee its presence on the network and its ability to respond to client requests. This is how Jini gets its robustness. It can limit the possibility of clients trying to communicate with dead servers. To do that, the service and the lookup service negociate an amount of time during the registration process. This is the length of the lease. For example, when the printer finds the lookup service through unicast or multicast, it negociates with the server an amount of time for which it thinks it will be present. This can be any length of time, for example 10 minutes. After 10 minutes, the lookup service should receive a lease renewal request from the printer telling it that it is still alive and would like to guarantee its presence for another 10 minutes or for any other amount of time. If no word is heard from the printer at the end of the lease, the lookup service assumes that the printer is not available anymore and deletes its registration from the database and stops sending clients to that printer.

The printer could decide on its own that it does not want to respond to client requests anymore. In this case, it can inform the lookup service of that or simply wait for the end of the lease and not make a lease renewal request. This will stop clients from trying to use the printer. The printer could also experience problems after having registered on the lookup service. For example, it could become broken or there could be a power failure in the room where the printer is located. When this happens, it is unable to respond to client requests. If clients attempt to contact it, they will not get a response and this can considerably slow down the network. The chances of this happening are lower because of leasing and therefore the efficiency of the network is

much better. For example, if the lease time is 10 minutes then we know that the printer has manifested itself less than 10 minutes ago. In other words, we know that it was alive less 10 minutes ago. This means that the chances of it not responding to a request are lower but it can still happen.

Transactions

Transactions are another important part of Jini. They are a set of operations that should be executed as a group or not at all. Many operations can be combined into one and if one of these operations fail for some reason, the complete operation is canceled. Jini has support for transactions. We can mark actions as part of a transaction and this tells Jini that it must start a transaction manager that supervises the execution of the operations to make sure that if any one of them fail, the status of the applications stays the way it was before the beginning of the transaction.

The classic example of a transaction is payment. Payment means that money is taken from one person and then given to another. Using bank accounts this means that money is withdrawn from a first account and then deposited in the bank account of another person. These are two operations, withdraw and deposit, that must be treated as one operation. Money could be lost if a power failure occurs after the withdraw operation but before the deposit operation. By marking both operations as being part of a transaction, we tell Jini to use a transaction manager that monitors both operations. Essentially, it tells the two participants on the network which actions to do and then tells them to do them basically at the same time. Following this, it verifies that all the operations in the transaction have been made. If not, it tells everyone to revert back to the previous situation.

Conclusion on Jini

This section introduced the Jini network technology. We saw that it uses the Java programming language and therefore inherits all the security features of the Java programming language. However this is not enough. Jini still lacks in security. For example, the data that is sent across the Jini networks is not protected by default. Third party users can listen on the network to see what is going on. The lookup service is very vulnerable to attack. If it falls, the network cannot be used anymore. There is still some work to be done to improve the security of this distributed system.

2.3.2 Common Object Request Broker Architecture (CORBA)

CORBA is a distributed object architecture introduced in 1990 by the Object Management Group (OMG) [22, 19]. The architecture allows objects to operate across networks regardless of the language in which they were written or the platform on which they were deployed. CORBA allows users to develop applications in the language in which they are the most comfortable. It is interesting because it can use the Java programming language. It is also very mature and has some security.

CORBA consists of many different and very important components. We describe some of the most important ones in this section.

The Object Request Broker (ORB)

The Object Request Broker (ORB) is a software component used to facilitate communication between objects. It is the central element to CORBA. The ORB is the programming that acts like a broker between the client request and the completion of the request. It is therefore responsible for the negotiation of the use of the server by the client. Using the ORB on a network means that the client may request a service without the need to know where the server is on the network and what the interface of the server looks like.

To facilitate the communication between objects, the ORB provides many services:

- It permits the location of a remote object, given an object reference so that given an object reference by a client, it will find the corresponding object implementation on the server.
- When a server is located, the ORB makes sure that the server is ready to receive the request before sending a request to that server.
- It marshalls parameters and return values to and from the remote method invocations.

The CORBA Object Request Broker allows clients to invoke operations on distributed objects without concern for the object's location, programming language, operating system, communication protocols, or hardware used.

The Interface Definition Language (IDL)

CORBA is designed to be independent of the programming languages. Servers and clients that communicate together can use different languages. The Interface Definition Language (IDL) specifies the interface between languages. It is an abstraction of different programming languages, hardware, and operating systems architectures. Interfaces described in IDL can be mapped to many languages. Developers may use whatever language they like to program their applications, but the interfaces are developed in IDL. Because of this independence of language, a client programmed in Java can easily communicate with a server programmed in COBOL, which communicates with another server programmed in C++. This is made possible because the interfaces are all developed in the same way using the IDL.

Language mapping is a specification that maps the IDL language to another language. For example, in IDL when we are programming an interface, we use the term `interface` that is not used in C++. So in the IDL-C++ language mapping, the term `interface` is replaced with the term `class`. Different languages obviously have different mappings. Put together, they make CORBA into a language independent architecture.

The current IDL language mappings available are for Java, C, C++, Smalltalk, COBOL, Ada, Lisp, Python, and IDLScript.

The Trader Service and The Naming Service

In CORBA, there are two ways to find a service. One is the more direct way with the naming service. This is when we know what the object's name is and we want to contact it. The naming service uses the naming context, which contains a set of bindings with unique names. These are name-to-object associations. A naming context is like any other object so it can be bound to a name in a naming context too. Binding contexts with other contexts creates a naming graph. This is a directed graph with nodes that are contexts themselves. A naming graph allows more complex names to reference objects.

The other way to find a service is with the trader service. This is when we want to look for a type of service. The trader service advertises services and discovers them. It therefore enables objects to locate other CORBA objects, but it does not use their names to do that. To find an object using the trader service, the objects look for object types like their parameters or the result types.

The classic comparison of this is the phone book. The naming service is like the white pages. We use them when we know the name of the person or company that we are looking for. If we do not know its name but know its type, we use the yellow pages. We look for all the computer software companies in a city for example. The yellow pages give us a list of all the companies under that type. In the trader service, we look for a type of object, without looking for a specific one necessarily.

The Internet Inter-ORB Protocol (IIOP)

The Internet Inter-ORB Protocol (IIOP) is the standard protocol for communication between ORBs on TCP/IP based networks like the Internet. IIOP is a specialization of the General Inter-ORB protocol (GIOP), a high-level standard protocol for communication between ORBs. It cannot be used directly because it is an abstract protocol so it is specialized by IIOP to work on TCP/IP based networks. The specialization is then used directly.

In order to be considered CORBA 2.0 compliant, ORBs must be able to support IIOP. They can, of course, support other protocols as well. Before CORBA 2.0, the OMG let vendors decide what protocols they wanted to use for their ORBs. The interface portability was present because of the IDL, but the ORBs could not talk to each other. ORBs from different vendors needed a standard way to communicate with each other with no compatibility problem so a new specification was needed. It would let ORBs make requests to each other. This is what CORBA 2.0 offers in IIOP.

CORBA 2.0 defines this network protocol to allow clients using a CORBA product from any vendor to communicate with objects using a CORBA product from any other vendor. The standard protocol being TCP/IP, it means that any CORBA product can contact any object from any CORBA product across the Internet. This standardization was needed because of the increasing number of people using CORBA. Also, networks are quickly becoming bigger and companies merge and they all use the Internet now. Different components of networks often come from different sources. Using CORBA, developers are certain that the objects should always be able to communicate together, even when they come from different vendors.

The Portable Object Adapter (POA)

The Portable Object Adapter (POA) allows programmers to build object implementations that are portable between different ORB products. They are used to dispatch their requests to their servants, which are the implementations of the interfaces. The IDL definitions' implementations are coded and are referred to as servants.

The POA was introduced in CORBA to replace the Basic Object Adapter (BOA) that was considered incomplete. The BOA was less portable and provided less features than the POA does. The POA replaces the BOA as the primary way of making implementation objects available to the ORB for servicing requests. All object adapters are concerned with the mechanisms to create CORBA objects and associate the CORBA objects with programming language objects that can do the work. The POA provides an expanded set of mechanisms for doing this.

The POA is not a simple replacement for the BOA; it is designed to be universal. It was designed to be portable and flexible and it is an object only visible to the server object. The clients hold object references on which they can invoke methods, but the server object (the one that implements the methods) only talks to the POA. The POA plays a role with the ORB to decide to which function the client requests must be passed.

The following explains how the POA works.

The server creates the POA when it is launched. The server must then tell its servants about the POA. When this is done, it asks the POA for an object reference. This reference is advertised to the rest of the network with the naming service or an Interoperable Object Reference (IOR) string. The IOR is another way to reference objects, using a string instead of a name. The IOR contains remote object information that allows a client application to invoke methods on a remote object.

When the client is launched, it starts by asking the Naming Service about the object reference. Then it is able to invoke the distant object's methods with the object reference. This means that the object reference holds the information to locate the server. The client, through the stub, passes the invocation to the ORB. Once the ORB has found the correct server with the object reference, it hands the request to the server.

At this point, using the information contained in the object reference, the server can locate the POA that created the object reference. It sends the request to the POA. The POA can then send the request to the appropriate servant.

Conclusion on CORBA

CORBA is the second distributed system that we studied. Comparing it to Jini, we notice that it is older and offers the possibility of using a number of different programming languages. These are two interesting characteristics, but we still decided against using CORBA because Jini still offers more interesting possibilities for us. For example, Jini has built-in support for leasing. Something that is very interesting but not found in CORBA. Also, support for mobile code is simpler in Java where we only use one programming language. Using only one programming language is not really a disadvantage but it is still possible to use more than one programming language in Jini by using proxies to connect the Jini network to the client or server coded in other languages.

Some work is being done to connect networks that use only Java to CORBA networks. Java uses remote method invocation (RMI) to let clients invoke methods on remote servers while CORBA uses IIOP for communication between clients and servers. A protocol called RMI-over-IIOP exists that lets those two types of networks communicate. It would be possible to connect CORBA networks to Jini networks using this system if it became necessary. However, we still made the decision not to use CORBA in our research so we do not go into more detail about this distributed system here.

2.3.3 Microsoft .NET

Microsoft .NET is an Internet software development technology released by Microsoft in 2000 [7, 20]. The .NET initiative is ambitious; it revolves around the .NET framework and includes programming languages, exception platforms, and extensive class libraries to provide built-in functionalities. This section introduces Microsoft .NET, which was studied in our search for a distributed system to use and make more secure.

The Microsoft .NET framework introduces a new model for the programming of applications, especially distributed applications. It uses the eXtensible Markup Language (XML) for the creation of structured documents. With XML, it is simple to create documents, verify their syntax, and then read the documents. The framework introduces a new communication protocol called Simple Object Access Protocol (SOAP) to share information over networks. SOAP uses the HTTP[1] protocol for the transmission of information written in XML over a network so that it can be read by machines running any kind of operating systems. Using HTTP ensures that everyone can use SOAP since

[1]HyperText Transfer Protocol. This is the protocol used to transfer hypertext documents over a network. It is widely used on the Internet world wide web service.

it is the most common protocol of the Internet. Microsoft .NET works together with XML and SOAP to create a new level of integration of software over the Internet.

The Microsoft .NET framework is a platform for the development of applications that includes a virtual machine that abstracts a lot of the Windows API[2] for the developers. It also includes class libraries with more functionalities and a development environment that includes many programming languages including Visual Basic .NET, Visual C++ .NET, Visual C# .NET, and Visual J# .NET. The first two are already well known, we briefly introduce the last two later. Those languages are separate but can work together with .NET. Like in CORBA, communicating between applications that use any of those programming languages is simple but .NET goes even further. For example, a class written in one of those languages can be subclassed using another language. This is something that was not done before and makes the languages even more integrated together.

To create applications running with Microsoft .NET, a new development tool was created called Visual Studio .NET. It is an upgrade from Visual Studio 6 released in 1998. Visual Studio .NET supports all the programming languages listed above and makes the creation of applications running with the Windows operating system interface simple.

The idea for .NET is the creation of globally distributed systems. Using XML, applications running on different machines all over the world can come together as a single application. The Microsoft vision is similar to Sun Microsystem's vision with Java. It is to have all the different types of systems, whether they are servers or wireless small machines to share the same platform with versions of .NET available for all of them. Each different type of systems can transparantly work and communicate together.

Finally, the Microsoft .NET applications are simpler than the traditional Windows applications because they only need to install their own core. For example, the application produced using .NET does not need to install the runtime or some modules, which are installed separately. They are part of the .NET framework and are only installed once per system. This means that it is quicker to install .NET applications and it takes less space because components of .NET do not need to be installed more than once. Also, it is no longer necessary to run an installation program. It is possible for some .NET applications to simply copy the files from its source to the hard disk where the application is installed.

[2]Application Program Interface. This interface describes methods that developers must use to communicate with the operating system or other applications running on that operating system.

Many Programming Languages

Microsoft .NET lets developers use programming languages that are already well known such as Basic and C++. It also introduces new languages that are more directly adapted to the .NET technology. Developers can use the language that they are more familiar with to create their applications and .NET will let the different languages work together in the application.

C# and J# are new programming languages. As their names indicate, C# is based on C++ while J# is based on Java but they are not the same. C#, introduced in 2001, uses a lot of the C++ syntax to make the creation of .NET applications more straightforward without losing the familiar C++ syntax.

J# offers to developers only familiar with the Java syntax a way to create applications that use .NET. It keeps the familiar syntax of of the language while offering to developers the possibility of creating advanced XML web services.

Microsoft .NET offers no support for the standard version of Java developed and maintained by Sun Microsystems.

This section does not go into more detail about the languages. We concentrate only on what .NET is and not on how to create applications that use it.

Communication in Microsoft .NET: XML and SOAP

To let applications communicate using Microsoft .NET, XML and SOAP are used. XML is the language used to create documents that are to be transferred while SOAP is the protocol used to transfer information. XML is especially important for the use of web services. A web service refers to an application that is meant to offer its services on the Internet. Clients can contact the web service using the Internet to make requests for actions on the services. Information regarding the request is written using XML and is transferred online using SOAP under the HTTP protocol.

SOAP is independent of the platform used by the services, clients, or anyone in between in charge of routing the requests and responses between the client and the service. It does not matter which programming language is used in any of the participants in the communication. SOAP also works with the HTTP protocol for the transmission of data to avoid asking system administrators to modify their firewalls. The HTTP

protocol is already used by everyone for Internet communication on the world wide web so it is not necessary to make any changes to the networks to let the participants use Microsoft .NET.

SOAP is an XML messaging specification that uses XML to send and receive data. It codifies the practice of using XML with HTTP. Before SOAP, the transmission of XML documents using HTTP was already possible but SOAP offers a simple and efficient way to do it with the Microsoft .NET technology. SOAP does not use an API or a runtime because it is independent of platform and programming language. It does not need to be used by a specific or traditional web server. Developers can adapt it to many different types of servers.

The Common Language Runtime (CLR)

The Common Language Runtime (CLR) is the heart of the Microsoft .NET technology. It is the key to the functionalities of the technology. For example, the CLR offers a common system for data types. Using the common types with a standard interface convention makes the cross-language inheritence possible. Earlier, we said that Microsoft .NET lets classes written in one programming language inherit classes written in another language. This is possible because of the CLR common system for data types. This section looks into the CLR in more detail.

Microsoft .NET offers a system of garbage collection where the memory is cleaned of things that are no longer in use. CLR is what makes the garbage collection work by counting objects to make sure that they have all been deleted once a method has finished its execution.

The capabilities of the CLR resemble in many ways those of the Java virtual machine. But while the JVM is designed for platform independence, Microsoft .NET is designed for language independence. For now, Microsoft .NET is supposed to be independent of platform but in practice, it is not always the case. It is still sometimes difficult to use Microsoft .NET on systems that do not use Windows as an operating system.

The CLR is the environment in which the programs are executed. It runs the .NET applications compiled to a common language called the Microsoft Intermediate Language (IL), which is a CPU independent instruction set. Programs written for the .NET framework are compiled to IL. In some ways, IL resembles the Java bytecode, which makes Java a platform independent language. IL makes it possible for .NET

applications to have cross-language integration and it contains instructions for calling, loading, storing, and initializing methods on objects. The language is executed (not interpreted) so it means that it is compiled before being run. This gives IL good performance. Security features are present in IL such as type safety to verify the types of parameters before the code is executed.

Using the common type system (CTS) found in the CLR, Microsoft .NET can support the types and operations found in many programming languages. The system can support multiple languages because of this. By describing the types of various langauges in a common way, the CTS is used to enforce type safety to ensure that all the classes are compatible with each other. The CTS defines how the declaration and usage of types during runtime work. This enables types in one language to work with types of another language. This includes exception handling. Using the CTS, the runtime can also ensure that code does not attempt to access memory that has not been allocated to it.

A simple and fast development and an automatic handling of code "plumbing" are some of the goals of the CLR. This means that memory management and process communication are done efficiently. Scalability was also an issue because systems running Microsoft .NET must work with small and large systems.

The simpler and faster development goals are achieved through a consistent framework that allows developers to reuse more code. The system offers a large set of functionalities so that more code is reused. Standard methods of accessing these functionalities exist in .NET. The development of programs is much simpler because of the consistency in the interfaces used to intereact with those functionalities.

Taking away from developers the handling of memory management and process communication relieves them of a big responsibility. It makes the development of .NET applications simpler and safer because there is less room for mistakes to be made. C++ programmers for example are used to worry about memory management but they do not have to with Microsoft .NET.

Conclusion on Microsoft .NET

Microsoft .NET is an Internet software development technology that is used to create distributed systems primarily aimed at web services. It includes many different programming languages, exception platforms, and class libraries to provide simple functionalities that makes the creation and use of those web services simple. These charac-

teristics all agree with our criteria for a choice of a distributed system. Unfortunately, one of our criteria is not met. This distributed system technology does not support the Java programming language. There is some support for the older versions of Java and for a programming language similar to Java called J# but this is not enough.

Microsoft .NET was not chosen because there is little support for Java, the programming language that we are using in our system. Also, Microsoft technology is known for being more difficult to personalize. Java lets us make many changes to the way it works so that it can more adequately fit our needs such as support for more security properties. Microsoft .NET does not give us the same liberties.

2.3.4 JESSICA Project

The JESSICA project is a middleware architecture that runs on top of the UNIX operating system to support the parallel execution of multi-threaded Java applications in a cluster of computers [30]. Its goal is to improve systems that use Java Distributed Virtual Machines (DJVM). Goals of this project include maximizing parallelism and minimize load imbalance on networks.

A more general goal is to achieve high-speed execution of real-life Java programs on large-scale clusters without having to modify existing Java applications. JESSICA stands for "Java-Enabled Single-System Image Computing Architecture". It is part of a research project at the Department of Computer Science and Information Systems at the University of Hong Kong.

The project is reviewed here because it was studied as a possible distributed system that we could use to improve its security. It is a very promising subject but does not respond to one of our criteria. It does not offer a traditional client-server type of network. However, as described in this section, it could be used in those networks to improve the execution of the client requests on servers that could distribute their tasks elsewhere on the network. The researchers have found efficient ways to distribute threads among Java virtual machines.

It is important to note that this section goes into some details on the second JESSICA architecture called JESSICA2. The first project, called JESSICA, started in 1998 and was successful in distributing Java threads but was not as efficient as JESSICA2. In 2001, the researchers tried again and created the more efficient system and called it JESSICA2.

JESSICA2 is one of the rare DJVMs to support thread migration. The system uses it transparently among virtual machines to run multi-threaded Java applications on clusters.

Doing thread migration means that the Java virtual machines can receive and send threads to and from other Java virtual machines. They work together to decide what is efficient to do. It may be more efficient to keep the thread and execute it or it may be more efficient to move it to another machine. The machines must decide whether to keep the thread or move it and if they decide to move it, they must decide where to move it. All this is done very rapidly and efficiently because thread migration increases the efficiency of Java applications. It must not slow the execution down.

JESSICA2 uses just-in-time (JIT) compilation [6]. The JIT type of compilation increases greatly the performance of this architecture by compiling the code or part of the code directly into machine code to run it directly on the machine instead of on the virtual machine. This can be done right before the execution with the results cached for future use. The independence of platform and operating system is not lost because the compilation is done just before the execution. In the past, the DJVM would not use JIT. JIT complicates greatly the JESSICA2 architecture but it makes it more efficient.

A group of servers and other resources that work as a single system to enable load-balancing and parallel processing is called a cluster. Load-balancing in this case is very important because this is part of what JESSICA2 aimed to accomplish. It takes a task that one machine must do and divides it into two or more smaller tasks. Those tasks are normally threads, which are a series of operations within a program. The system transfers those threads to other machines that are part of the cluster. Together, the machines work in parallel to accomplish the larger task. The system must examine those tasks to see if they can be divided into threads and if so, if they can be divided efficiently. Then it must divide the task and then merge the results of each thread to get the larger result. The result is that an application has run in parallel across different machines and therefore has been executed more quickly.

The migration of threads is transparent. This means that the user will not see the thread migration at work. The idea is to give the illusion that the user is using only one virtual machine. This is the single system image (SSI) illusion and offers a true parallel execution environment.

The monitoring of the threads on the network must be perfect. Losing threads in between the virtual machines on the network is not acceptable. The choice of where to send the threads on the network has to be done quickly and efficiently. Threads cannot

be sent to already overloaded machines while other machines are doing nothing. The details of how this is done are not given here because the goal of this section is simply to understand what JESSICA2 can do.

JESSICA2 introduces JITEE, the just-in-time execution engine, which supports the execution of Java threads in a distributed mode but also in JIT compilation mode. JESSCIA2 also introduces the global object space (GOS) [11]. It is a layer that supports the access of shared objects by multiple distributed Java threads. It is used a lot in JESSICA2, but it is transparent.

The GOS is a way the researchers found to save data and classes in a way for them to be found easily regardless of which JVM currently is executing the thread that uses them. This is a very efficient way to save data and classes. It provides access to location-independent objects. One of the goals of JESSICA2 is to create a unified shared object space for all the distributed object threads.

GOS is an important part of JESSICA2 because it lets the threads move around the network without the need to constantly move the data around with them. When developing this new DJVM, the researchers needed to follow the already existing Java memory model for compatibility. Based on that, multi-thread Java applications assume that there is a single heap visible to all the threads. The heap stores all the master copies of the objects. Each thread has a local working memory to keep the copies of the heap that it must access. When the thread is in execution, it operates on this local memory. Java threads use monitors to synchronize the concurrent thread execution in a critical section. When a thread enters one of the monitors, it must move its working memory to the heap to make sure that it can access the latest object data. When leaving the monitor, a copy of the objects in local memory must be made to the heap. The JESSICA2 architecture follows this.

Conclusion on JESSICA

JESSICA2 is an interesting project for efficient load-balancing on networks to distribute the execution of Java applications to improve their performance. A lot of work has been done on this subject and it works very well. It works using the Java programming language so it meets one of our criteria. However, our research subject is to find a distributed system that supports the traditional client-server model so that we can work on improving its security. JESSICA2 does not meet this criteria so we do not use it in our research.

In the future, it could be possible to include this technology in our systems. Sometimes clients may request the execution of complex tasks on the server. When this happens, the servers could be connected on networks that use the JESSICA2 architecture to distribute the request and obtain the result more quickly. We do not do this in our systems, but it would be possible to further study JESSICA2 and to eventually include this technology in our system.

2.3.5 Chosen Distributed Network Technology

After reviewing Jini, CORBA, Microsoft .NET, and JESSICA2, it was decided that we would use the Jini network technology from Sun Microsystems for our system. Jini is written in Java so it inherits the security features of the language. It also has various interesting features not found in CORBA such as leasing and the lookup service. CORBA is more complex to use because of its support for other languages, something we did not need in our system.

Microsoft .NET was also considered but its lack of support for the Java language ruled it out. It was required in our research project that the chosen distributed system must use the Java language. Also, Microsoft products are more difficult to modify for our personal use. To improve security, we would need more access to all the features of .NET.

JESSICA2 is a very interesting project but it does not do exactly what is needed in our system. It is not of the traditional type of networks where servers offer services on a network to clients. JESSICA2 distributes the execution of processes on the network, which is different. In the future, it may be interesting to merge that type of network to our system, where the services offered by servers are distributed across the network for their execution to be faster but that was beyond the scope of our research.

2.4 Tools and Techniques to Enforce Security Properties

This section explores the different tools that can be used to improve the chosen distributed system. The previous sections studied various network technologies to find one to use and improve its security. In those sections, it was decided that we would use the Jini network technology. Now, we look into the different tools that can be

used to improve its security such as encryption, digital signatures, JAAS, and SSL authentication.

2.4.1 Encryption

Encryption is the translation of data into a form that should be unreadable by unauthorized users. It has become very popular to ensure the confidentiality security property. Encryption is interesting in distributed systems because data travelling on networks are not protected by default. Anybody with the right equipement can listen to what is being transmitted on a network and that becomes a problem when the data being transfered is supposed to be confidential. This is the case for example in electronic commerce when credit card information is being transmitted or in authentication systems when a password is being sent to the server. These types of data should always be protected from unauthorized users.

To prevent this type of data from being compromized, we use encryption. Sometimes systems will encrypt all data that is being transmitted, not just confidential data so that any information that is being transferred can be confidential.

To encrypt data, algorithms are used with special encryption keys. A key is used on data, which is passed through an algorithm. The result of this algorithm is data that is completely different from the original data. By reading the result of the encryption algorithm, it is impossible to understand the original data. To obtain the original data, a user passes the encrypted data through another algorithm. This is called decryption. The result of the decryption is the original encrypted data. Without the knowledge of the encryption algorithm or the encryption key, one cannot decrypt the information and find out what it is.

There are two types of encryption methods that are explained here. They are symmetric encryption and asymmetric encryption. Symmetric encryption refers to encryption that only uses one key to encrypt and to decrypt information. Asymmetric encryption refers to encryption that uses two different keys, one to encrypt data and one to decrypt data.

Symmetric encryption is used for example by a user who wants to protect his or her data. This person may encrypt a file with a key and then when he or she needs the data, the key is used to decrypt it. It is also used on networks. For example, the SSL authentication algorithm results in a symmetric key that the users communicating use to protect their data. Only the users who have access to the data know the key and

can decrypt the information. It is very secure and very fast. The problem with this method is that anyone who needs access to the data must know the same key. This can be a problem if many different people must have access to data. Users may change the data without permission because the same key is used for encryption and decryption. Asymmetric encryption is a solution to this problem.

Asymmetric encryption is encryption using a key pair. A public key and a private key are used together to encrypt or decrypt data. This is a slower encryption method than symmetric encryption but sometimes it is the only choice available for some secure systems.

As their names say, the public key is public so it is available to anyone who may need it while a private key is only known by the owner of the key pair. Both can be used to encrypt or decrypt data but they have their specific uses. Information encrypted using a public key can only be decrypted by the corresponding private key while information encrypted with a private key can only be decrypted using the corresponding public key. This is very intersting for transmitting confidential data because as long as the private key remains private, the integrity of the information can be ensured when it is encrypted with the private key. Also, users can be sure that only the receiver of a message will be able to read it if it is encrypted with that user's public key.

For example if Alice and Bob are communicating and Alice wants to send a confidential message to Bob, she can encrypt the data with Bob's public key and send the data across the network to Bob. Anyone listening on the network will not be able to understand this data because they do not know Bob's private key. When the data arrives, Bob can read it by decrypting the message using his private key.

If Bob wants to send a message that is not confidential to Alice but wants Alice to be convinced that it actually comes from him, he can encrypt the data using his private key and send it to Alice. Alice receives the data and can use Bob's public key to decrypt it. If the decryption works using Bob's public key, she knows that it must have been encrypted using Bob's private key so the information must come from Bob.

The problem with asymmetric encryption is that it takes considerable more time to encrypt and decrypt. However, this technology is absolutely necessary for digital signatures seen in Section 2.4.3. Also, the public keys must be distributed somehow between the users who need them. We normally use a trusted certificate authority (CA) but this requires a lot of work to implement. The CA generates public and private keys for users and issues digital certificates that contain this information and sends them to the owners of those keys. The certificates contain the name of the user along with his or

her public and private keys and expiration dates. They are digitally signed by the CA and anyone who receives them can verifiy their authenticity. They also create public key certificates that are sent to users who must have a copy of other users' public keys.

Our system described in Chapter 4 uses a lot of encryption. The participants on the network are identified by public keys and the SSL authentication results in symmetric keys that are used to encrypt all the data that is transmitted during this communication.

2.4.2 Message Digests

A message digest is a hash of a message. It is a sequence of bits that is the result of passing the message through an algorithm. It is simple to understand and it is used to ensure the data integrity security property. Digital signatures, the topic of the next section, use message digests.

Data that is sent across the network is vulnerable to being accidently or purposely modified before it arrives at its destination. To prevent this, a way must be found for the receiver to easily verify that the data that he or she receives is the same as the data that was sent by the sender. To do that, the sender puts the data through an algorithm that hashes the data. This result is a small number of bits that correspond to the data. Everytime the same message passes through the same algorithm, the same sequence of bits will be the result but if the message is somehow different then the end result of the digest will be very different. The result of the hash algorithm is called the digest.

The sender and the receiver of the message use the same algorithm. The sender takes his or her data and passes it through the algorithm to get the digest. Then, he or she attaches the digest to the message and sends it on the network. The receiver of the message and the digest takes the message and puts it through the same algorithm. The receiver then compares this result to the digest that was sent by the sender and if they are the same, the receiver can be sure that the message was not modified before it arrived.

It is important to note that it is impossible to use the bits that create the digest to get the original message. This is because the number of bits is significantly smaller than the number of bits that make up the message. This does not matter because the goal of the message digest is only to verify the integrity of the data and not to recreate the original message.

2.4.3 Digital Signatures

A digital signature is a signature performed on digital documents. It is used with message digests and asymmetric encryption to authenticate the origin of a message. The goal of digital signatures is to convince the receiver of a document that the person claiming to have created the message actually did create the message.

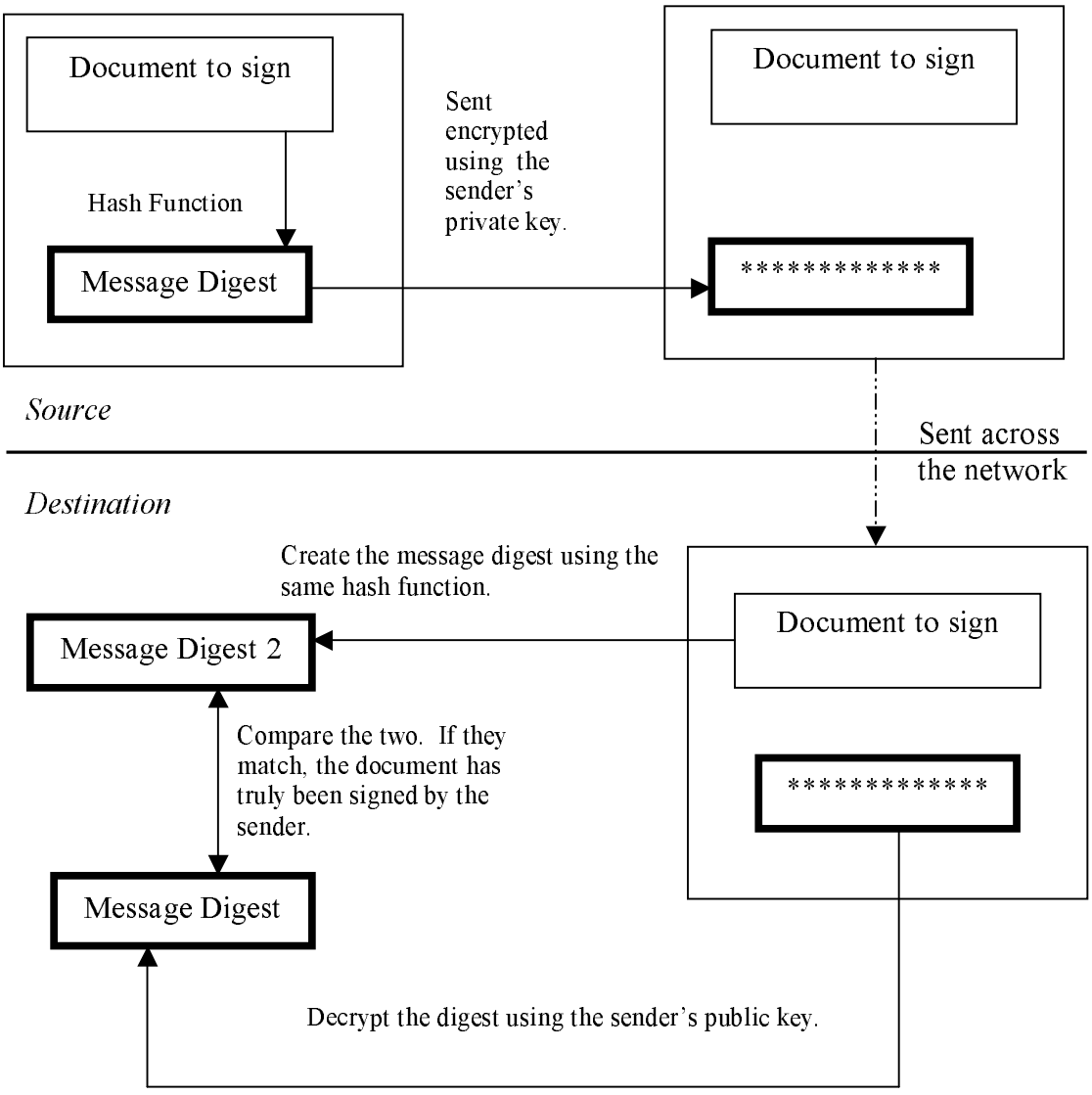

Figure 2.6: Summary of the Digital Signature Process

As shown in Figure 2.6, the sender takes his or her message and creates a message digest using the algorithm described in the previous section. The sender then takes his or her private key and encrypts the message digest. The message, along with the

encrypted message digest, are sent on the network to the receiver.

The receiver knows the public key of the sender. He or she may get it in any secure way. This can be from a trusted certificate authority for example, as explained in Section 2.4.1. Using the sender's public key, the receiver decrypts the message digest. Then he or she takes the received message and puts it through the same message digest algorithm used by the sender. At this point, the receiver can compare the decrypted message digest with the one just created with the algorithm. If they are the same, the receiver is not only convinced that the message has not been changed (accidently or purposely) during transmission but can be sure that the message was actually created by the person claiming to be the creator of the message.

The system can be improved when the data must be confidential. The sender may encrypt the entire message including the message digest using the receiver's public key. By doing that, the sender is sure that the only person who is able to decrypt the data is the receiver. The receiver decrypts the entire message using his or her private key and then the rest of the steps are the same to get the original data and verify its origin.

2.4.4 Java Authentication and Authorization Service (JAAS)

The Java Authentication and Authorization Service (JAAS) is an extension to the Java 2 security model used to perform authentication and authorization for Java applications [27]. The Java 2 security model offers developers the possibility to perfrom access control for protected resources by verifying permissions based on the origin of the code and the signer of the code when it is downloaded from a remote location. This information is verified on the code before it is executed and is compared to the security policies written by the owner or the administrator of the resources. The access controller makes those verifications and decides if code should be executed or not. JAAS adds to this by performing access control on the identity of the user executing the code.

This section explains how JAAS works. We concentrate on the authentication portion of JAAS because the authorization componant is the subject of Chapter 3.

Authentication

In JAAS, **subjects** are authenticated. We call a user or a service that has been authenticated using JAAS a subject. It is the information about the authenticated entity

so it contains one or many **principals** and may contain some **credentials**. Principals are user identities of which users may have more than one. During access control, they are used to identify a person or service.

The credentials are the information used to authenticate users. They can be public or private and they contain security related attributes about the principals. Examples of public credentials include the user's public encryption key, which is known by anyone who needs the information while private credentials include for example user passwords.

To let Java applications be independent from the authentication process, JAAS performs authentication in a pluggable fashion. This means that developers may change the way authentication is performed in a system without changing the application itself. To authenticate users and services, JAAS uses a **login context**, one or many **login modules**, and a **callback handler**.

The login context provides the basics for performing authentication and one login context can be used by many applications. It is called when an application needs to perform an authentication process. The login context consults its configuration to find out which authentication services must be used for authentication for this particular application. These authentication services are called login modules. An application may require that authentication occur with many login modules or just one.

The login module contains the implementation of an authentication protocol. It can be as simple as the traditional user name / password combination or more complex like Kerberos or SSL authentication. The login module will sometimes use the callback handler to communicate with the user to receive or send information. The callback handler is used for communicating information between the login module and the user or service requesting authentication. The user may send information such as his or her username and password while the login module would send information on whether authentication has been sucessful or not. The callback handler is always separate from the login modules so that many callback handlers can use the same login modules. Normally, the callback handlers are defined in the application.

The user or service requesting an authentication sees the calling of the login context followed by the calling of one or many login modules as a single operation. The application calls the login context, which calls each login module associated with this application. Normally, each login module must correctly authenticate the user or service for authentication to be successful. If the login modules need to receive or send information from or to the user or service, they use the callback handler to communicate with them. The login modules return true if authentication is successful on them

or false otherwise. Sometimes, the login modules may try to authenticate a user or service many times before returning false. The login context receives the answers from the login modules and returns to the application true or false depending on the login modules' answers.

If the authentication is succesful, a subject is created that contains information about the principals that were authenticated as well as credentials about the subject. Those credentials can be anything that could be useful for the application, such as information on how to communicate with the subject.

We give a summary of the steps towards the authentication of a user or a service using JAAS.

1. A login context is started by an application.
2. An application tells the login context that it requires the authenitcation of a user or a service by using the login method.
3. The login context consults its configuration to load all the login modules configured for that application. The application can require one or many login modules.
4. The login context's login method invokes all of login modules associated with that application in the configuration. Each login module attempts to authenticate the user or service. If one or more of the login modules succeed, then the login modules associate the relevant principals and credentials with a subject.
5. The login context returns the authentication status to the application. This is where the application finds out if authentication was succesful or not.
6. If the authentication was succesful, the application retrieves the subject from the login context and associates credentials to the subject. The credentials are used to identify the authenticated subject and may contain additional information about it.

Authorization

The authorization part of JAAS is similar to the Java 2 security model except it adds the possibility to perform authorization based on the identity of the user as well as the origin and signers of the code. Chapter 3 goes into detail on how authorizations are performed in Java. This includes the description of the authorization component of

JAAS. For our system, JAAS and the original Java 2 security model are always used together so they are not explained separately here. Furtheremore, since the release of the Java 2 SDK 1.4 platform, the JAAS package has been completely integrated into Java so the Java security model includes JAAS.

Conclusion on JAAS

JAAS is a useful Java component that authenticates users so that applications can perform some access control based not only on the origin and signers of code but also on the user or service using the code. In our system, we required that access control be possible using those three parameters so we used JAAS in our system to perform authentication and authorization. As explained in Chapter 4, we created a login module that authenticates users using the Secure Sockets Layer (SSL) (see Section 2.5.2) protocol.

2.5 Authentication Protocols

This section explores two authentication protocols. We studied authentication protocols because authentication is one of the security properties that we were to implement in our system. It is one of the security properties that must be enforced in a distributed system for it to be considered secure. We must therefore find a way to enforce this security property. We decided to study Kerberos and SSL because we know that they are mature and reliable. In the end we must make a choice between the two.

2.5.1 Kerberos

Kerberos is an authentication protocol developed by researchers at the Massachusetts Institute of Technology as part of the Athena project in the late 1980's [17]. This section is a description of this authentication protocol, which allows a client to prove its identity to a server without sending confidential information on the network. It is summarized in Figure 2.7.

Kerberos is a big improvement over the traditional name and password authentication. In the past, these passwords were sent across the networks and sometimes they

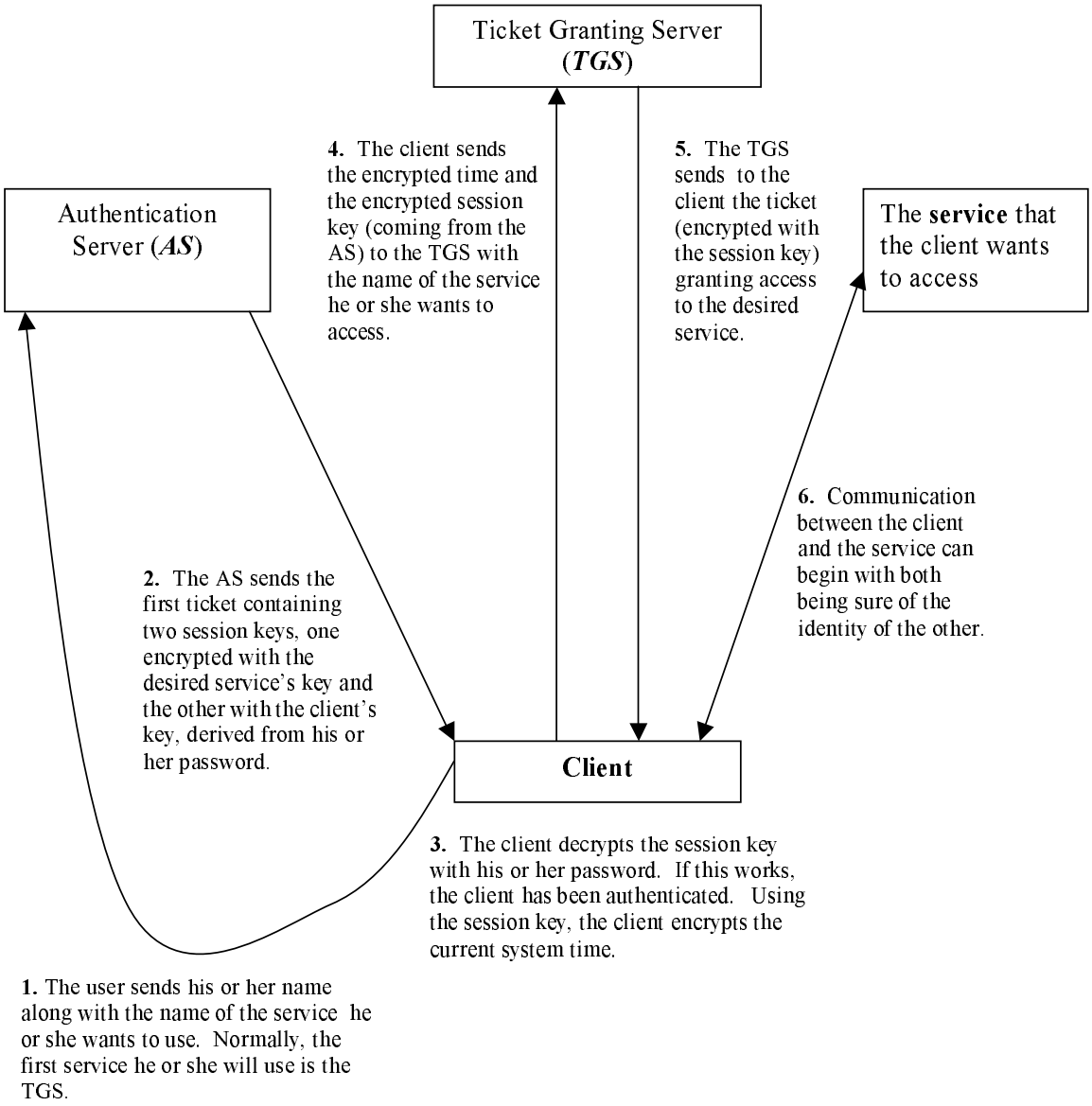

Figure 2.7: Summary of the Kerberos Authentication Protocol

were sent without encryption. Kerberos fixes this problem by using secret-key cryptography and by not sending the passwords on the network at all. Kerberos also reduces the number of times typical users must enter a password and reduces the number of passwords that a user may need. By doing this, the chances of a password being discovered are less important. Users will be less tempted to use easy to guess passwords if they have less passwords to remember. Hackers will have less opportunities to find passwords if the user uses their passwords less often.

Kerberos uses two types of servers, authentication servers (AS) and ticket-granting servers (TGS). It is possible for these two servers to be on the same machine but they are separate. The AS has one different secret key per user and one different key per service of the network. The secret key that is present on the server is derived from the user's password while the service's secret keys are generated randomly. The TGS gives to users and services a ticket that gives access to services on the network.

The first step in Kerberos authentication is the user who sends his or her name to the AS along with the name of the service he or she wants to use. The first service that the user will normally want to use is the TGS. The AS builds a ticket that contains a session key encrypted with the TGS key and a session key encrypted with the user's key. This ticket is sent across the network to the user. This information, known as the ticket granting ticket, stays confidential because it is encrypted. Users normally store this ticket so that they no longer need to contact the AS until this ticket expires. Whenever they want to access a different service, they use this ticket with the steps described below.

When the user receives this ticket, he or she can decrypt part of it using their password because this ticket is encrypted with a key that is derived from the password. This is the only time that the password is used and it is not sent across the network. It is only used locally. If this decryption works then the ticket granting ticket has been sent to the same user who requested it. This user now has a copy of the session key.

Using the session key, the user encrypts the current system time. The encrypted time and the second half of the ticket granting ticket (the part encrypted using the TGS key) are sent to the TGS along with the name of the actual service that the user wants to use. The TGS receives this and decrypts the half encrypted with his key to obtain the user's session key. Using that key, the TGS decrypts the second half of the message to find the current time. If the TGS finds the current time then the user has been authenticated with the TGS without ever sending his or her password on the network. The current time decrypted by the TGS may not be the actual current time at the time of decryption because some time will pass between the encryption and the decryption

of the current time. A difference in time is acceptable and depends on the particular implementation of Kerberos.

At this point, the TGS returns a ticket to the user. This ticket grants access to the desired service. Whenever the client wants to authenticate itself with that service, it sends this ticket. Services running on this protocol will only accept valid tickets that come from a valid TGS. This ticket contains the name of the client and its address to prevent users who may intercept the ticket from using it. If someone listening on the network copies the ticket and tries to use it from another address, the ticket is rejected. The tickets also contain a timestamp. This means that the service knows when the TGS created this ticket and will only accept it for a specific number of hours that depend on the implementation of Kerberos. After that time has passed, the user needs to get a new ticket.

Conclusion on Kerberos

We decided against using Kerberos in our system. The reason for this is that, as explained in Section 4.3.1, our system must use a local name space. Essentially, this means that the participants on the network cannot be uniquely identified by their names, they must be identified by a public key. Other authentication protocols such as SSL are more adapted to this type of name space.

2.5.2 Secure Sockets Layer (SSL) Authentication Protocol

The Secure Sockets Layer authentication protocol was created by Netscape in 1994. It is used a lot on the Internet to authenticate users, especially for electronic commerce. This section explains this authentication protocol. We look at how it works before looking at its algorithm. We refer to the authentication protocol as SSL, but the most recent version of the protocol is called the Transport Layer Security (TLS). This is SSL version 3.0, the one we describe in this section.

The first step in the protocol is when the client first contacts the server. The client wants to authenticate itself on the server. The authentication of the server on the client is optional in the definition of the protocol but it is necessary in our system. The client and the server begin by negociating the use of a particular cipher and the use of a certain version of the SSL protocol to use. If they cannot agree on a version of SSL to use then authentication does not occur and the communication between the two ends

here.

Next, the client and the server exchange public key certificates. They check the authenticity of those certificates using the public key of the certificate authority, which they have received previously using a secure method. They encrypt random numbers and send them to the other to test their ability to decrypt them using their public key. On the Internet, this information is normally present in the Web browsers.

After authentication has occured using the algorithm described below, the client knows the identity of the server and the server knows the identity of the client. To communicate, they can use a symmetric key that is generated separately on the client and on the server using information that was securely exchanged by the two parties during the authentication process. It is symmetric so the client and the server both use the same key to encrypt and decrypt information. That key was never sent across the network so no one can know what it is except for the two participants in the communication so it is secure.

The SSL Algorithm

1. The client sends to the server a "Hello" message. This message contains the version of SSL that the client would like to use along with a session identifier in case the client is trying to continue a previous authentication session. The client also joins the list of cryptographic algorithms that it can support. Finally, the client generates some random texts and sends them to the server.

2. The server sends to the client its "Hello" message. This message contains the same type of information that the client sent to the server in step one. The server also sends its public key certificate. If the authentication is supposed to be bidirectional, which is the case in our system, it also asks the client for its public key certificate.

3. The client validates the public key certificate and with the received public key certificate, the client begins to authenticate the server.

4. The client creates a "Premaster Secret". This is a message generated by the client that is used in the creation of a symmetric key later. This message is encrypted using the server's public key received in the certificate. The client then sends the encrypted premaster secret to the server.

5. The client sends its public key certificate to the server along with data that has been signed by the client's private key. This is used for the client authentication

on the server. This is an optional step in SSL, but a compulsory one in our system because we require bidirectional authentication, as explained in Chapter 4.

6. The server validates the public key certificate and with the received public key certificate, the server begins to authenticate the client. It can begin by verifying the signed data sent by the client with the public key found in the received certificate.

7. The server decrypts the received premaster secret using its private key. Using the premaster secret, the server can generate the master secret. The client can generate the same master secret using the same premaster key that it created in part four and the random texts that were generated in steps one and two.

8. Using the master secret, the client and the server can both generate the same key. This is the symmetric key.

9. The client and server each send a message "Finished" to each other to indicate to that they are done. The "Finished" message is sent encrypted with the symmetric key that was just generated with the random data of steps one and two and the master secret generated in step seven. This symmetric key cannot be generated on both the client and the server if one of them is trying to authenticate itself with someone else's public key.

10. At this point, both the client and the server are authenticated. Secure communication can begin. The symmetric key generated here is used to protect the information sent between the client and the server.

The Security of the SSL Protocol

As mentioned above, the SSL protocol is not new. It was first introduced in 1994. Since then, it has been carefully analyzed. Researchers have looked for, found, and fixed flaws in the protocol. An in depth analysis of the protocol [28, 16] was especially studied by us in our studies to look for an authentication protocol to use. It found a few minor flaws as well as several new active attacks against SSL. However, these flaws are easily corrected. The conclusion of that analysis is that SSL is a valuable contribution towards practical communications security.

The study also found that, in general, SSL provides excellent security against eavesdropping and other passive attacks. A passive attack is when an attacker is not communicating with any of the participants. He or she is simply listening on the network to try to gain confidential information or information that will later help him or her

perform active attacks. An active attack on the other hand is when an attacker does some communication with some or all the participants on the network, perhaps by impersonating a client or a server. An active attack can also consist in blocking some of the communication between the participants of the network.

The protocol was not found to be perfect. A number of small flaws were found as well as some active attacks but the study emphasizes that it is not hard to patch up the small flaws that permit those attacks. Most of the weaknesses of the protocol found in this analysis can be corrected without changing the entire protocol.

Conclusion on SSL

SSL is the authentication protocol we decided to use for our system described in Chapter 4. More details about SSL and our implementation of it are available in that chapter. The protocol is tested, secure, and mature [28]. We implemented the protocol using the public keys as the unique identity of the users and we created a JAAS login module as described in Section 2.4.4 to make it work. The resulting symmetric key is used to create secure sockets, which are used in the transmission of data in a secure way. SSL answers our criteria as being able to use it with a local name space. We do not need to identify users with a name and a password so it was perfect for our system.

2.6 Specification Languages

As part of our research, we studied two specification languages to see if they could be used to improve the Jini networking technology's security. SDSI / SPKI was an interesting choice because of its use of a local name space, which is a criterion of our system. The Ponder Specification Language also offered many security properties that were interesting to us. This section introduces these two specifications with some detail. We did not implement those languages into our system. What we did was use them to improve Jini security.

2.6.1 SDSI / SPKI

This section is a description of the SDSI / SPKI technology. SDSI / SPKI is a merging of the Simple Distributed Security Infrastructure (SDSI) [21] and the Simple Public Key

Infrastructure (SPKI) [5]. The specification offers ways to efficiently introduce some security in computer networks. It was first proposed by researchers at Microsoft and at the Massachusetts Insititute of Technology (MIT). The specification combines a simple public key infrastructure design with a means of defining groups and issuing group membership certificates. Groups defined in SDSI / SPKI provide simple terminlogy for defining security policies using access control lists (ACL).

Another important point in SDSI / SPKI is that it gives up the traditional hierarchical design of the traditional certificate systems of name spaces. This means that participants on the network create their own names, public and private keys, and authentication certificates. This enables SDSI / SPKI to offer a mechanism of local names instead of global names. This makes participants on the network more independent from each other but at the same time, it becomes more complicated to verify the authenticity of certificates. Performing authentication is more difficult in this type of name space but SDSI / SPKI has some good ideas on how to make the networks secure anyway.

SDSI was introduced in late 1996. At the same time, the Internet Engineering Task Force (IETF) was proposing SPKI, which introduced a flexible means of specifying authorizations. In 1997, both technologies were merged to form SDSI / SPKI. The creation of this new technology was motivated by the complexity of current designs for certificates like X.509[3], which are often considered incomplete and rely too much on global name spaces.

SDSI is based on public key cryptography. All participants on the network own a public key and a private key, which work in the same way described in Section 2.4.1. This lets the participants on the network sign the messages that they exchange so that the origin of the messages can always be verified. The problem with that is, to verify the origin of a message, a user must decrypt the signature, which was encrypted using the sender's private key. The only way to do that is by using the sender's public key. To get a user's public key, we normally use a certificate authority but SDSI / SPKI does not use this. Another way must be found for users to send to other users their public keys.

This is where SDSI / SPKI introduces the notion of trust. To find another user's public key, a user can ask one of his or her friends on the network. This friend is trusted by the user and encrypts the needed public key with his or her private key to send it to the user. The user can decrypt the key with his or her public key and then use that decrypted key to verify the origin of the received message. If the friend does not know

[3] An authentication certificate standard especially used in SSL/TLS authentication protocols.

the needed public key, that friend can ask another friend and a chain can begin between friends of friends until the needed public key is found. Since the user asked a friend, it can be assumed that the user trusts him or her. Also, the user would know the public key of this friend. When the needed public key arrives, the user can use it and then save it for future use.

This system takes away the need for a central certificate authority. It also lets users use a local name space. This means that users are free to chose their own public key, private key, and name. Just like in the real world, there can be many people named Bob. This is not a problem because people can still be uniquely identified using other methods, such as their social insurance number, which is the equivalent of the public key on this type of network. It is always unique. The advantage of this is that a participant on the network can use the network with no prior experience with it. Maybe the user is uniquely identified as Bob on another network, but when he connects to this one, he is no longer unique. Nothing needs to be changed from one network to the other. Participants make friends with the other participants and with time they get to know the other participants and get their public keys and messages can be exchanged in a secure way.

The SDSI / SPKI specification is more than just encrypting messages and verifying their origin. The authorization security property is also ensured in this specification. To access a protected ressource, the user presents proof that he or she has been authorized to access it. The proof is a certificate chain that starts at the administrator or owner of the resource and goes all the way down to the actual user. This permits actual authorization by the administrator to the user and delegation of this permission from authorized users to the actual user. Delegation is an interesting security property. We explain what it is and how we implemented it in Chapter 4.

Using SDSI / SPKI to Secure Jini

After introducing the basics of SDSI / SPKI, we look into how it is possible to include some of the ideas from this technology into the distributed system that we chose to make more secure. The goal of our research is always to find ways to make a distributed system that uses the Java language more secure. We decided to use Jini, now we look at how SDSI / SPKI can help us make that distributed system more secure.

One thing we must do to make a system more secure is to add support for authorization. Support for authorization can be done using access control lists as explained above. We look into two types of access control lists, the first one is the traditional type

of ACL. It is very simple to understand and to use, but does not offer much more than a simple authorization process for users. The second one uses SDSI / SPKI authorization certificates to enforce authorization. It also permits the delegation of permissions. Both could be used to include some authorization in Jini.

The traditional ACL, shown in Figure 2.8, contains a list of users with their passwords and a list of resources (R_x) that they are permitted to use. The resources are controled by a central server, which receives requests from clients for use of the resources of the network. The server verifies the identity of the clients and using that, it checks to see if the client is permitted to use the desired resource or not.

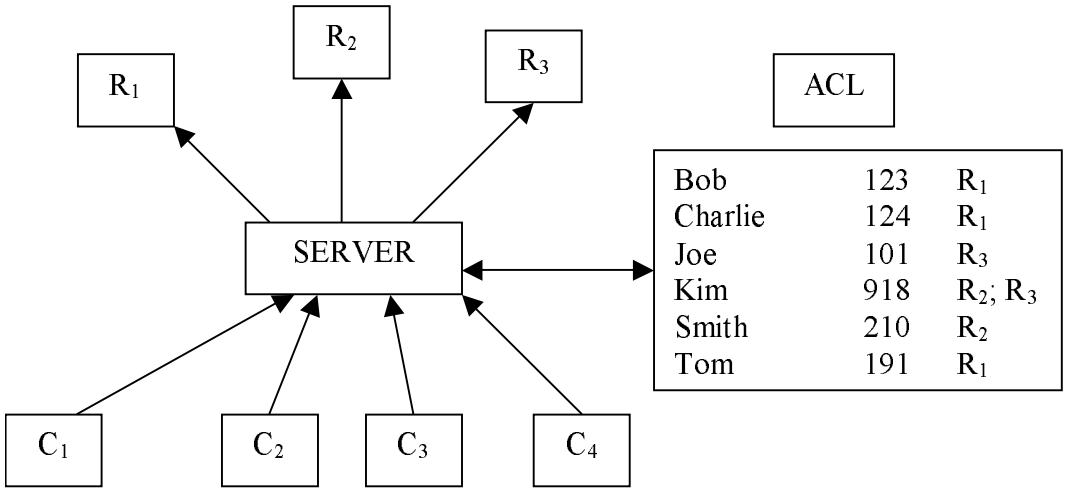

Figure 2.8: Access Control List: Version 1

A series of clients uniquely identified by names have access to a server, which controls the access to the resources. The different clients can be used by anybody. A user, using one of the clients, sends his or her name to the server along with the password and the name of the resource he or she wishes to use. This is normally transmitted over a secure network. The server receives this information and compares the name and password with the ones in the ACL. If there are no matches, then the communication is denied. If there is a match then authentication has been performed and the server knows for sure who is trying to use the resource. The server then checks to see if that authenticated user is permitted to use the desired resource. If so, then the server lets the user use the resource. If not, communication is denied.

The list of resources associated with the users is not always necessary. Sometimes just being correctly authenticated may be enough to access the resources.

This is a very simple ACL. They can become more complex, but that type of ACL

always relies on global names. There is no secure way to perform delegation. A user would have to give his or her name and password to another user to delegate permissions and this is very insecure. There is also no room for anonymity. If may be interesting sometimes to let users be anonymous on networks, but using an ACL that identifies users with a name, it is not possible.

We will look at another type of ACL that uses SDSI / SPKI authorization certificates. This second type of ACL is more powerful and is interesting also because it allows for the secure delegation of permissions.

Figure 2.9 demonstrates the second type of ACL. This one uses SDSI / SPKI authorization certificates to perform authorization [10]. In this case, users are not required to be identified by name; they must send an authorization certificate to the resource (R_x), which they wish to access. The resource may be administered by a server when it does not have enough resources to perform access control on its own.

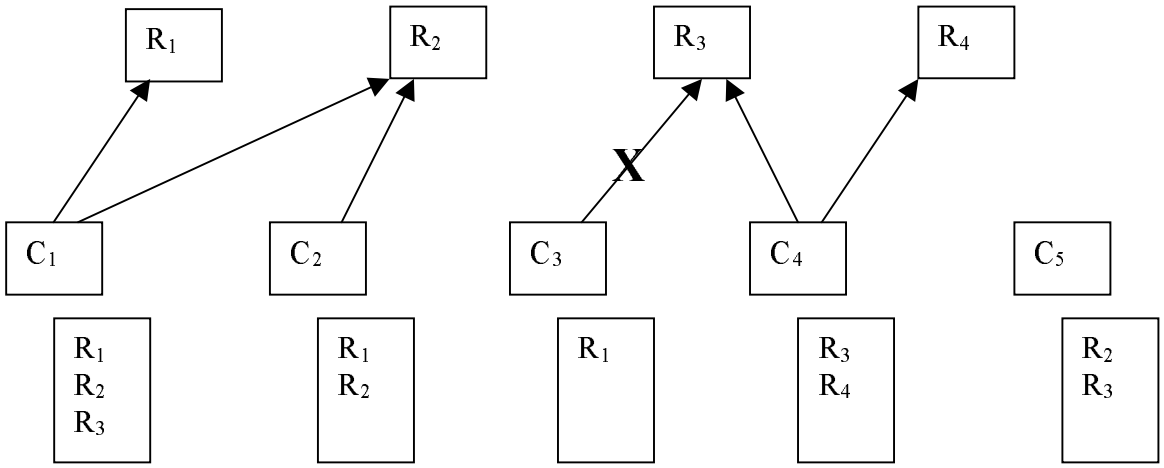

Figure 2.9: Access Control List: Version 2

For using this type of ACL, clients have a list of resources that they wish to access. In the first type of ACL, the list of resources was present on the server performing access control but in this ACL, the list of resources is present on the clients. The resources are identified by authorization certificates, which come from either the administrator of the resource or from someone who has delegated his or her permission to this user.

For example, a system administrator may have given to C_1 access to R_2. To do that, the administrator creates an SDSI / SPKI authorization certificate of R_2 for C_1, signs it, and sends it to C_1. When C_1 attempts to use R_2, he sends his certificate to the resource. The resource verifies the validity of the certificate. If it is considered valid, C_1 can access the desired resource. If not the access is denied.

The administrator may permit C_1 to delegate its permission to other users. This is marked with a boolean flag in the certificate set to true. If C_1 is permitted to delegate his permission, he can create a new certificate, called a delegation certificate, giving access to R_2 and sends it to another user, for example C_2. Now C_2 has access to R_1 just like C_1 but this access was not given by the administrator. C_1 may permit C_2 to delegate the delegated permission or not with another boolean flag in the delegation certificate. Delegating permissions securely was not possible with the first version of the ACL, but it can be done in this version.

Whenever the client wants to use this resource, he uses the delegation certificate. He sends it to the resource that he wants to use. Upon reception, the certificate's signature is verified. To verify the signature, the resource needs the public key that corresponds to the private key that was used to sign the certificate. He either has the public key on his system or must ask a friend on the network for it. The signature is verified and an authorization certificate or delegation certificate is asked of the client who signed the delegation certificate. The process then starts again. The resource continues to do this until the certificate chain is complete to the administrator of the resource. The administrator of the resource first gave the permission so if the chain can go back to him, then the delegation certificate was valid and access can be granted.

The use of local names is possible using this ACL. Each client can call itself whatever it wants. This was not the case in the first version of the ACL where clients had to use a unique global name that a server could recognize. In this version of the ACL, if the authorization certificate is valid or if the delegation certificate is valid with a valid certificate chain going back to the original authorization certificate then access is granted.

The first version of the ACL did not permit anonymity because servers needed the name of the client before being able to decide whether access can be granted or not. In this version of the ACL, the name of the client is not always necessary. Networks can permit the anonymous logins. The minimum requirement is to have a valid authorization certificate from a valid source. With that certificate, access is given. There was no way to do this in the first version of the ACL except by creating an anonymous (or guest) account on the server giving everyone who desires to be anonymous the same permissions. The second version of the ACL is more powerful.

The second version of the ACL is also more independent. There is no need for a central server that controls resources. This is more secure because there is not one machine on the network that can be attacked to bring the entire network down. Also, when permissions change, there is no need to make changes on the access control lists.

Certificates have expiration dates and permissions can be taken away by changing the signature of certificates or letting them expire. The first version of the ACL requires that constant changes be made to the actual ACL to reflect changes in the permissions.

Conclusion on SDSI / SPKI

The Simple Distributed Security Infrastructure (SDSI) and the Simple Public Key Infrastructure were studied to see if we could use them to secure our distributed system. It would have been possible to implement many aspects of SDSI / SPKI in our system but we studied another specification language that proved more interesting. That language is the subject of the next section. Still, time spent studying SDSI / SPKI was not wasted because we learned about local name spaces. Local name spaces let us create networks in which participants are more independent of each other and, as explained in Chapter 4, this is something we did implement in our system and it comes from SDSI / SPKI.

2.6.2 The Ponder Specification Language

The Ponder Specification Language, developed at Imperial College, provides a way of standardizing the creation of security policies [8]. We studied Ponder to see if it could be used to improve the security of Jini. It helped us understand some security properties that we could implement in our system. At this point we knew that we were going to make Jini more secure, we now needed to find out how to do it.

The Ponder specification implements a way to write all kinds of different security policies that use many different security properties. It provides the user with a simple way to write and manage those security policies.

We do not directly use Ponder in our system explained in Chapter 4. As we did with SDSI / SPKI, the topic of the previous section, we use the specification language to find ways to improve Jini security. It would be possible to actually use Ponder to write our security policies but this would require us to write a policy parser that can read Ponder policies or a compiler that creates Java policies from Ponder policies. This required too much time so we took some security policies from Ponder and elaborated a policy syntax to write policies using those security properties. Then our system reads and uses those policy files to enforce access control. More information on this is given in Chapter 4, but this section goes into detail about what Ponder is without repeating

what is found in that chapter.

Ponder Security Properties

Section 4.2 explains each security property that we chose to implement in our system so we do not explain them again here. Instead, we explain four other security properties of Ponder that we did not implement in our system. They are interesting security properties but we did not have enough time to study them all and implement them in our system to secure Jini.

The first Ponder security property that we look at is the **filter**. Filters are used with positive authorization[4] policies to create actions from the parameters entered by a user (input parameters) or from parameters that are the result of an action (output parameters). A user may be permitted to perform an action only with certain parameters and filters are used to control the permissions on those actions.

If the filters work on the input parameter, the permission may be checked before the execution of the action. The system verifies that the user has the needed positive authorization and then checks the authorization's filters againts the parameters entered by the user. If the action is permitted, it is executed and the result is sent back to the user.

If the filters work on the output parameter, the action must be executed before the permission can be verified. The system first checks the positive authorization and if the user has this permission, the action is executed. Following that, the result of the action is compared against the filters in the positive authorization. If the filters permit the user to see those parameters, then the result is sent to the user. If not, the result is never sent to the user.

Filters can work with a combination of input parameters and output parameters in which case both types of verification occur.

The second Ponder security property that we look at is the **refrain**. Refrains are similar to the negative authorization security property (see Section 4.2.2). They work to limit the actions that a user may perfom on a protected resource. Even if a user has the permission to perform an action, it should still be denied if it is attempted. Refrains are used when we cannot trust the access controller to actually enforce the

[4]Positive authorizations are a particular form of the authorization security property. They are explained in Section 4.2.1.

negative authorization. This can be the case for example when the resource where the access control is performed is something or someone cannot be trusted and would be happy that the restricted action would instead be executed. Refrains control the actions using the name of the user attempting the action instead of the resource it is trying to restrict or protect to make sure that the actions are restricted.

The third Ponder security property is the **obligation**. Obligations are a series of actions that must be performed by the system when certain conditions are present in the system. For example, if an access violation is detected, the system could shut down or it could alert the administrator of the system automatically. Obligations let the administrator of the system specify those actions that are to be executed when those conditions are present. Some of those actions can be started automatically while some are specified to be executed by a specific user.

The last Ponder security property that we look at is the **role**. Roles are different than the previous three security properties in that instead of being aimed at implementing the restriction of actions by creating new types of security properties, they help the administrator write specific security policies.

Chapter 4 shows that our system supports groups where different users can be members of groups to make the writing of security policies simpler. Roles are similar but they represent a positition in an organization. For example, one of the company's security policies may be written for the type of user called *employee*. When a user logs on and the system checks this user's properties based on his or her name, the time of day, or any other criteria, it may see that this user at this time is of the type *employee*. This gives the user the security policies written for the employees of the company. Another user may be of the type *customer*, *president*, or *unknown* for example. When logging on, the user would get the security policies of the type of user the system decides to give him or her. Deciding which role a user will have occurs when he or she is authenticated and it may depend on more than their name because permissions may depend on other criteria. For example, an employee could have more permissions during his or her normal working hours but logging on to the system at any other time would give this type of user less permissions. During his or her working hours, the person is of the type *employee* but at any other time, this person is of the type *customer*, which normally has less permissions. Using roles simplies the creation of security properties for the administrator.

Conclusion on the Ponder Specification Language

This section looked at the Ponder Specification Language. It is a very interesting language that standardizes the creation of security properties. We use it in our system but only indirectly. We took some of the security properties that Ponder offers and wrote our own policy syntax for them. We then created a system that would enforce the chosen security properties. This section explained the security properties of Ponder that we did not implement in our system. Having more time, it would have been possible to implement them as well. The other Ponder security properties are described in Section 4.2. They are the ones that we implemented. The rest of that chapter is dedicated to explaining our developed system that uses those security properties.

2.7 Conclusion

This chapter described the first part of our research, which consisted in exploring various distributed systems before making a choice on which one we would use and make more secure. Our main criteria was that the chosen distributed system supports the Java programming language. It also had to be a traditional type of client-server network where clients send requets to servers, which treat those requests and send responses back to the clients.

First, we looked into various security properties that we must find in distributed systems to call them secure. We use these security properties to know what to look for in a distributed system and also what we can implement in a distributed system to make it more secure.

We also looked at different access control mechanisms. The access control matrix, the Bell-LaPadula Model and the Chinese Wall Model were studied to find out how access control can be implemented in distributed and non-distributed systems.

We studied Jini, a young distributed system from Sun Microsystems. Jini is written in Java and therefore inherits all of its security properties but does not offer much security beyond that. Jini has interesting features such as a central lookup service where clients look for possible services that can respond to their requests. Another feature included in Jini is leasing, which significantly lowers the chances of clients being directed towards servers that have been disconnected. Finally, transactions are included in Jini. They give support for the accurate and efficient support of a series of actions

that must be executed as one action.

Secondly, we looked at CORBA, which is an older distributed system with support for many programming languages. This is a strong point of CORBA because it means that not all developers need to be using the same programming language to create applications that can communicate together. The distributed system also supports the traditional client-server model and it can run with Java. However, we feel that it is not as strong as Jini in regards to Java code mobility. Finally, leasing is not directly implemented in CORBA. For these reasons, we decided not to use this distributed system.

Another distributed system that we studied was Microsoft .NET. This distributed system is interesting because it is created for efficient use of web services. It supports the traditional client-server model and like CORBA, .NET is independent of programming languages, which is a very strong point. Unfortunately, it does not support the standard version of Java from Sun Microsystems so it could not be used by us. Also, personalizing Microsoft products is historically harder than personalizing Java so we feel that Jini is a better chose if we want to be able to improve the security of a distributed system.

The last distributed system that we studied was JESSICA2. This is a different type of distributed system than the last three. It does not offer the traditional type of client-server model. Instead, it offers to distribute the execution of Java appplications on a network to make the application run more efficiently. We could not use JESSICA2 for our research because it is not a client-server type of network but this technology could still be used to improve the efficiency of the Java applications running on the servers.

After studying the four distributed systems, we decided to use the Jini network technology. Jini fits all our criteria. It is written in Java and so we use that language to create networks with Jini. Also, it supports the client-server model. It has some security features that it gets from Java but it is not secure enough, leaving room for us to make improvements. It also has several interesting characteristics such as the lookup service, leasing, efficient code mobility as well as support for transactions.

Following the study of the distributed systems, we studied methods to make them more secure. We studied encryption, message digests, digital signatures, and JAAS. These are tools that can be used to make Jini more secure. We also want to implement authentication in our distributed system so we looked at the Kerberos and Secure Sockets Layer (SSL) authentication protocols. SSL is the one that we chose to implement in our system to improve Jini because it is a mature and reliable authentication protocol that works well across a network and more importantly can be used with the local name

space that we must use in our system. More information on how we implemented SSL in our system as well as on the local name space is in Chapter 4 along with all the details on how we secure Jini.

Finally, this chapter looked into specification languages that we studied to find ways on how to improve the security of networks. SDSI / SPKI was studied and this is where we learned how to use a local name space. The Ponder Specification Language was studied to find more precise security properties that we could implement in our system.

The next chapter looks into how Java access control is performed in the standard version of Java. We look at how it works before we look at how we can change it in Chapter 4.

Chapter 3

Java Access Control Mechanisms

In this chapter, we look into the Java access control mechanisms. We begin with a short history of the security of the language from the original sandbox model to how security is enforced in the language today. Following that, we look into more details on how security is enforced in the most recent version of the Java security model. This was used extensively in the development of our system. We must first know how Java security works before we can improve it to add our new security properties.

3.1 Introduction

To understand how the Java access control mechanisms work, we begin by looking into how it has evolved. As shown in Figure 3.1, the original Java version, JDK[1] 1.0, has very strict security mechanisms [26]. The execution of the code on the virtual machine is divided into two types, local code and remote code. Local code is the code that originates from the machine where it is to be executed and remote code is the code that originates from outside the machine where it is to be executed. Obviously, remote code is the one most likely to be dangerous so it must be executed with caution. Security in JDK 1.0 works using the sandbox model, which encapsulates the remote code to execute it with limited access to the system's resources. The local code on the other hand is executed with full access to the system's resources. The sandbox refers to the virtual box that contains the code and executes it while at the same time preventing it from accessing resources outside of the sandbox.

[1] Java Development Kit.

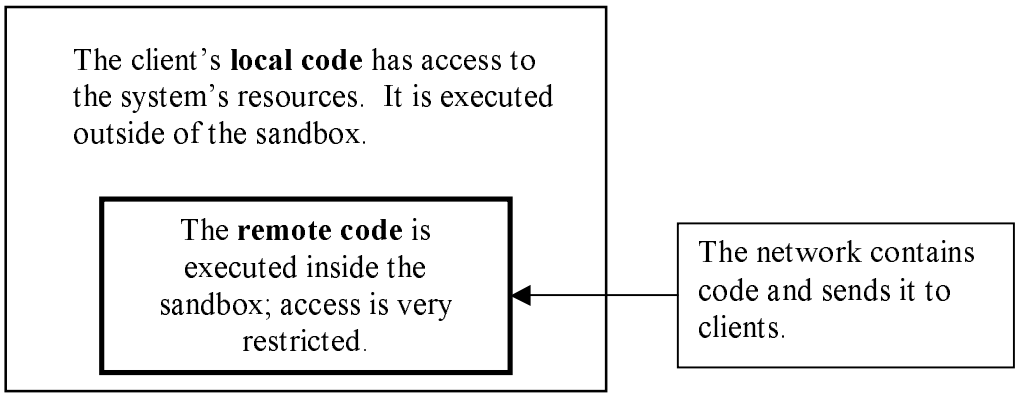

Figure 3.1: The JDK 1.0 Security Model

The sandbox model as described above was found to be too restrictive. It was sometimes necessary and still secure for remote code to have the same access rights as local code. A new security model was introduced later in JDK 1.1 to improve the Java security model. In this new model, the code is divided into three parts instead of two. The local code is still present with full system access but there are now two types of remote code, signed remote code and unsigned remote code. Signed remote code refers to code signed with a trusted signature. The code still originates from outside the machine where it is to be executed but this time it is signed. If the system recognizes the signature and trusts it then it lets the code execute itself with full access just like local code. Unsigned remote code or code that is signed by untrusted signatures is executed in the original sandbox.

To further improve security in the language, a new more powerful security model was introduced in JDK 1.2 [12]. This was a very big improvement. The older security models only had two types of code execution environments, "full access" and "restricted access". Full access gave to code complete access to the system and severely restricted the execution of the remote code but only in one way and it was not simple to modify the permissions of code executed in the sandbox. It became necessary to change this. The new model introduced the protection domain. The protection domain refers to a virtual box similar to the sandbox in which code is placed to be executed safely within its permissions. This lets the system create custom sandboxes. Figure 3.2 shows three protection domains. One sandbox can give to code some permissions while another sandbox gives to code different permissions. These sandboxes are the protection domains. They are created by examining the code's origin and signature.

To create the protection domains, the administrator of the system writes policy files, which contain permissions. In the policy file, the administrator specifies who is affected

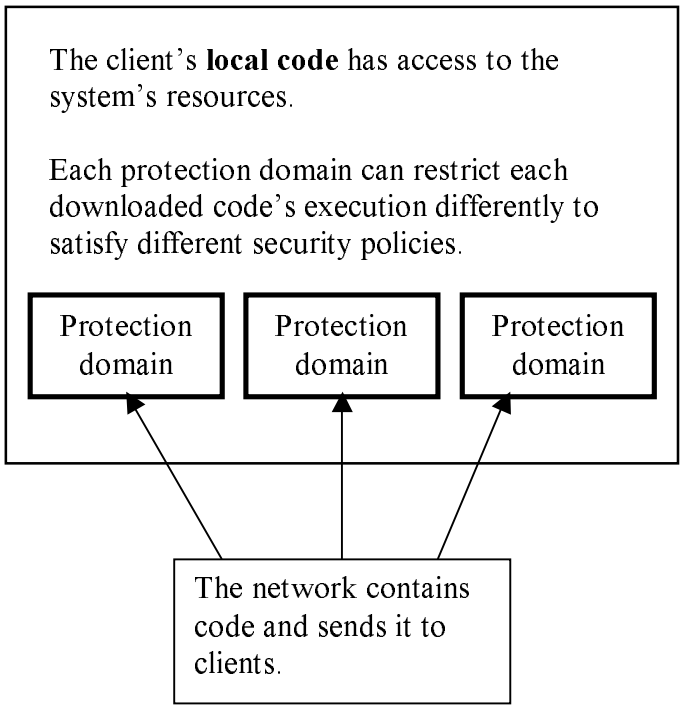

Figure 3.2: The JDK 1.2 Security Model

by the policies using either the origin of the code or the signature of the code or both. Then he or she writes a series of permissions for that code. When the code arrives, its signature and its origin are examined to see if they are affected by the security policies of the machine. If they are, a protection domain is created and the code is executed inside it. The creation of the protection domains using the policy file is explained in more detail in this chapter along with how the system uses the protection domains to perform access control. This is where most of the changes were made by us in our system. Those changes are explained in the next chapter.

JDK 1.2 is not the latest version but most of it is still in use today. There is one improvement worth mentioning that was put into SDK[2] 1.4 (the next major version of Java after JDK 1.2).

JDK 1.2 uses the origin of the code and the signature of the code to perform access control, SDK 1.4 adds the element of the identity of who is executing the code. To decide if access should be given or not, the system still checks the origin and the signature of code, but it also checks who is executing the code. This can be a human user or a machine connected to the system. Depending on who is executing the code, permissions can be different. This is the security tool called JAAS introduced in Section 2.4.4. JAAS

[2]Software Development Kit.

was created as an optional package in SDK 1.3 and was completely integrated into SDK 1.4.

This chapter goes into detail about the security mechanisms and algorithms of the latest version of Java, the one we used to develop our system explained in Chapter 4.

We begin by looking at the syntax of the policy files written by the administrators or the owners of the resources to be protected. Following that, we look at how Java performs access control. When an application is started, it first loads the security manager and the policy files in memory and then when sensitive operations are attempted, Java checks the policy files loaded in memory to see if they allow the operation to take place. If so, the operation can be executed, otherwise the operation is denied.

3.2 Java Policy Files

This section is a description of the contents of a Java policy file. First, we explain what the policy files looks like and what its contents mean. Following this, we describe how the Java virtual machine reads the file and uses it to perform access control.

The Java policy file is a file written by an administrator of the system. This file contains information on how the Java virtual machine performs access control on the current machine [1]. The contents of the file can be divided into two parts.

- The keystore entry. The keystore is the location where the public keys used to decrypt signatures of incoming code is located. There can be zero or one keystore. They contain a series of public encryption keys that are referred to in the policy file using aliases.
- The grant entry. The grant entry is the code source of users, services, or code that is affected by the current policy along with the list of permissions for this grant entry. The code source can contain an URL of the origin of incoming code, the aliases of the possible signatures of that code as well as a list of principals. All three parts of the code source are optional. There can be zero, one, or many grant entries.

Together, these two parts of the policy file tell the Java virtual machine how to create the policy for this system. Using this information, the system decides which

sensitive operations are permitted and which are not. When the system starts, it loads these permissions in memory and when a sensitive operation is attempted, it checks its memory to see if the operation is permitted or not. If it is permitted, the operation is executed, otherwise an exception is thrown and the operation is never executed. This process is explained later in this chapter.

The syntax of the grant entry is as follows.

```
keystore "keystoreURL", "keystoreType";
```

The keystore's URL specifies the location of the keystore in URL form. The URL can be absolute or relative to the policy file's location. It can be at a remote location or on the current machine. The keystore type parameter is optional, the most used type of keystore in the policy files is "jks[3]". If no paramater is given as the keystore type, the system assumes that the one specified in the security properties file[4] is used.

If no `keystore` parameter is present then it means that the security policy file does not contain aliases that refer to public encryption keys. This normally means that there is no way to verify the signatures of downloaded code. In some systems, it is not necessary to verify the signature so this is why the `keystore` entry is optional.

Following the optional `keystore` entry, the `grant` entry is present. It is composed of the code source and the permissions for that code source. The code source is composed of the `codeBase`, `signedBy`, and `principal` parameters. The permissions are a series of `permission` parameters.

The code source is a way for the system to identify who is affected by the permissions part of the corresponding grant entry. Each parameter of the code source is optional, while at least one `permission` parameter must be present if a grant entry is present in the policy file. When a sensitive operation is to be attempted, the system checks that the code source of the user or service trying to execute the operation implies the code source of the policy files. If it does, then the grant entry that corresponds to the code source is used for performing access control. More information on how the system compares code sources and how the system verifies if the permission should be given or not is given later.

Now, we look at each parameter of the grant entry of the policy file.

[3]Java keystore. It is the traditional type of keystore available from Sun Microsystems.

[4]The security properties file is the file containing various security properties used by the Java security classes. It is normally called `java.security`.

- codeBase. This is an URL that indicates the possible origin of code. If a codeBase is present then the security policy only affects code that comes from this URL. If no codeBase is present then the system reads it as "any codeBase". This means that code downloaded from any location can be affected by the current security policy.

- signedBy. This is a series of one or more aliases, which correspond to the public keys found in the keystore. These are the possible code signers of the code that is downloaded. If aliases are present then the downloaded code's signatures are verified using the public keys in the keystore that correspond to those aliases. The security policy only affects the code if all the signatures of the code can be decrypted using all the public keys associated with all the aliases specified here. If no signedBy parameter is given then the system reads it as "any signer". This means the signatures of the downloaded code is ignored.

- principal. This is present in JAAS and in SDK 1.4. It refers to the identity of who is currently executing the code. It is divided into two parts, principalClass (the type of the principal) and principalName (the name of the principal). If no principal is given then no authentication is necessary and access control is not performed using the user's identity to find the permissions.

- permission. There are one or many permission entries in the grant entry. These are the permissions associated with the code source.

The syntax of the grant entry is as follows.

The `codeBase` parameter is given as an URL:

```
codeBase "http://origin.com/*"
```

The `codeBase` parameter can be local or remote. If it is local, the protocol is file, otherwise it is http. The ending of the URL is important. If it ends with a "/" then it means that all the class files of that directory are included. If it ends with a "*" then it means that all the class files and JAR[5] files are included. If it ends with a "-" then it means that all the class files and JAR files of that directory as well as all the class files and JAR files of the subdirectories are included. The codeBase can also end with a JAR filename in which case the codeBase is only valid when that file is downloaded. If no

[5]Java archive file. Java class files that are to be downloaded together are normally put in a JAR file, which can be signed zero, one, or many times.

codeBase is given, the system reads it as any codeBase. In this case, code downloaded from any location is included.

The `signedBy` parameter is given as one or more aliases:

```
signedBy joe99, bob19
```

The `signedBy` parameter is a series of one or more aliases. If more than one signer is given, downloaded code must be signed by each signer to be affected by the current grant entry. The aliases are names that correspond to the entries in the keystore. For example, the public key that corresponds to `joe99` in the keystore file and the public key that corresponds to `bob19` in the keystore file are used to verify the signature of the downloaded code. If the keys successfully decrypt the signatures then the downloaded code can be affected by the current grant entry. Otherwise, it cannot.

The `principal` parameter is given as one or more principal names.

```
principal principal.class.Principal "Bob"
```

There can be many different principals in one grant entry. The subject currently executing the code should be identified as all of them to be affected by the security policy of this grant entry. The first part of the principal is the name of the class used to read the principal. Different classes exist for different types of principals. Some principals for example are identified by a name, others with a public encryption key. The principal class must be able to read the type of principal that the writer of the security policy file has entered. The name of the principal follows the name of the principal class. It can be anything as long as the principal class can read it.

Using the examples above, the code source parameter would look like this:

```
grant codeBase "http://origin.com/*",
signedBy "joe99, bob19",
principal principal.class.Principal "Bob"
```

Following the code source, the grant entry contains one or more permissions associated with that code source. The permission parameter first contains the permission class for that permission. This is the class that can read the permission entry. The

system creates an object of that type when reading the policy file to later check the permissions when necessary. The permission parameter can also contain a target and one or more actions that the administrator authorizes the code source to perform. The target is the name of the resource that the permission is restricting access to and the actions are the specific operations that this permission authorizes. The targets and action parts of the permission parameter depend on the permission itself. Some permissions require them while other permissions do not.

For example, the permission parameter can look like this.

```
permission java.io.FilePermission "/exams/*", "read";
```

In this case, the permission lets the code source read all the files in the `exams` directory. There are many different types of permission classes so we do not look into them individually here. It is also possible to create custom permission classes when access control is necessary on other types of actions.

Figure 3.3 shows an example of the standard version of the Java policy file. The figure shows all the parameters that can be found in a policy file. The keystore's location is first. The keystore is located on a remote location and is of type "jks". Next, the grant entry is present; it contains the code source and its permissions.

The code source entry is composed of the codeBase, the signedBy, and the principal parameters. There are two permissions for that grant entry. One is FilePermission, which is very common in policy files. The second permission is a custom permission class that we used in our system in the next chapter. It permits the multiplication of two integers.

```
keystore "http://keystores.com/.keystore", "jks";

grant codeBase "http://origin.com/*", signedBy "joe99, bob19",
principal principal principal.class.Principal "bob"

{
      permission java.io.FilePermission "/exams/*", "read";
      permission djvm.ArithmeticPermission "int, int","* ";
}
```

Figure 3.3: Example of a Standard Java Policy File

By using the policy files described in this section, administrators can write their policy files to restrict access on protected resources that are present on their systems. When the application starts, it verifies the syntax of the files to make sure that they are correct. Following that, the system loads them in memory. The next section explains how the system uses those policy files to perform access control.

3.3 Access Control

Once those policy files are written and loaded in memory, they are used by the system to perform access control. Access control refers to the action of deciding which operations are permitted and which operations are not permitted. The user or the program attempts to execute an operation and before it is executed, the virtual machine launches the access control mechanisms. These mechanisms examine the requested operation against the policies that exist at this moment. If those policies are interpreted as allowing the operation, the operation is executed, otherwise the operation is not executed. In this section, we explain how the policy files are read by the system and how they are stored in memory. The next section looks into the Java data structures used to store the policies in memory and how Java uses them to perform the access control.

3.3.1 Storing Permissions in Memory

The system security policy is represented by the Policy object and only one Policy object can be in effect at a time. Policy is an abstract class so the system security policy is actually represented by a class that is a subclass of Policy. The standard version used in Java and described in this chapter is called sun.security.provider.PolicyFile. We created our own policy provider, which is an improvement over the standard one and is called XMLPolicy. It is the subject of the next chapter.

PolicyFile stores and analyzes the permissions from one or more policy files. These files associate permissions to the class' origin and signers. This is done at the time when PolicyFile is created and initialized. Using the information in the policy files, PolicyFile creates a set of Permission objects to represent the permissions of the policy files.

By default, the system loads the policy files of the `java.security` file. This file contains security configurations for the system. It is possible to make the system load

other policy files when the application is started. This is done on the Java command line with the `-Djava.security.policy` parameter. For example:

```
java -Djava.security.policy=client.policy javaApplicationTest
```

This will start the Java application called "javaApplicationTest" with the security policy file "client.policy" loaded in memory. That file contains the security policy that will control access for that application. Notice that the example uses only one equal sign. This means that the policy file "client.policy" is loaded along with another policy file specified in the security configuration file `java.security`. It is possible to override that file by using two equal signs. With two equal signs, the policy file specified in `java.security` is ignored and only the one passed as a parameter in the command line is used.

PolicyFile reads the policy files at startup and when the application requests a refresh of the policy files. If the files are modified after the system has read the policies, the changes do not effect the execution of the application unless the `refresh()` method of PolicyFile is used. That method can be called by authorized users and services to reload the policy file to update the policies that are in memory so that changes can be taken into consideration during access control.

When the system starts, the PolicyFile object is instantiated and it reads the policy file. The data is stored in memory and ready to be used by the system to perform access control.

3.4 Permission Classes and the `implies` Methods

The permissions contained in the policy files are given using permission classes. The permission classes represent the access to the system's resources. The java.security. Permission class is an abstract class that is subclassed to represent more specific types of permissions such as file permissions or socket permissions. Those classes are used to give a permission in the policy file. For example, to give to a user access to files in a directory, we use the FilePermission class. The entry in the policy file looks like this:

```
permission java.io.FilePermission "/exams/*", "read";
```

The meanings of the `permission` parameters are explained in Section 3.2. In this example, the user, code, or service of the grant entry is permitted to read all the files in the exams directory and all its subdirectories.

When the policy file is read, the system instantiates a FilePermission object with the given parameters. The object is stored in memory and when an access to a file is attempted, it is used in the authorization process that decides whether permission should be given or not.

All the permission classes that exist in Java are subclasses of the class Permission. Custom permission classes can be created by extending the Permission class. One of the requirements of the subclasses is that they must implement an `implies` method. The `implies` method is used to decide if the granted permissions imply a desired permission. When a sensitive operation is attempted, the needed permission is passed to the `implies` method, which returns true if the granted permission implies the desired permission passed as a parameter. The `implies` method returns false if the granted permission does not imply the desired permission passed as a parameter.

The `implies` methods are extremely important. They are present in many places in the access control mechanisms of Java. Section 3.4.6 explains exactly how they are used. But first, we look at the classes used to store and analyze the permissions in memory.

3.4.1 The UnresolvedPermission Class

Recall that when the system starts, it reads the policy file and instantiates the permissions found in that file. Sometimes the code needed to instantiate the class is not yet available. It is not always possible to have access to the code of the classes of the policy file at the time when the system is started. It may be downloaded later for example. When this happens, the systems loads the permissions in an UnresolvedPermission object. The object stores the permission until code for it is absolutely necessary. This happens when the user attempts a sensitive operation that needs the permission that is still unresolved. At that point, the system should have the implementation of the related permission class so it uses it to resolve the permission. The permission object is then created and used to verify the permission and decide if the operation should be executed or not. If the code is still unavailable at this point, the system acts like the permission is not granted and the operation is not executed.

3.4.2 The PermissionCollection Class

The system does not simply load and instantiate a series of Permission objects to later call their implies methods randomly. They must be grouped in memory in a certain way if they are to be analyzed efficiently. To group the permissions, Java instantiates some PermissionCollection objects. The PermissionCollection class is used to represent a collection of permissions of the same type. Permissions are added to the PermissionCollection object in any order and the PermissionCollection class contains its own implies methods that calls the implies methods of the Permission classes when necessary.

3.4.3 The ProtectionDomain Class

The ProtectionDomain class is used to define the set of permissions granted to classes. When a class is loaded, it is associated with a set of permissions granted to that class. This is the ProtectionDomain class. Basicly, ProtectionDomain encapsulates a series of PermissionCollection objects that have the same code source. If the class being loaded in memory comes from a new code source, a new protection domain is created, otherwise the loaded class is associated with an existing protection domain that corresponds to the code source.

While the PermissionCollection class puts together permissions of the same type, the ProtectionDomain class puts together a series of PermissionCollection objects of the same code source. This is done to improve the efficiency of access control.

3.4.4 The Security Manager

The class SecurityManager in Java implements a component of the language called the security manager. In the past, the security manager was the only one responsible for the managing of the permissions. This was changed and permissions are now checked using the access controller. The security manager is still used for compatibility with previous versions of Java.

The class contains a series of `check` methods, which each create a Permission object of the type of the needed permission. They are called when a sensitive operation is taking place. The Permission objects represent the requested access. Those methods

use that Permission object to call the method `checkPermission(Permission)`. This calls the method `AccessController.checkPermission(Permission)`, which performs the acutal access control as described in the next section.

3.4.5 The Access Controller

The access controller is called from the security manager when a sensitive operation is being executed. The method `AccessController.checkPermission(Permission)` looks at the protection domains associated with the classes of the current thread of execution and creates the access control context, which is a series of protection domains of code source relevant to the current run time. The requested permission is tested againsts the all the protection domains of the context. The permission must be present in each protection domain for the permission to be valid.

The access controller is also used for stack inspection. This is a security feature of Java where special permissions are given at special times during the execution of classes where the permission should normally not be given. This is where code is marked as "priviledged" so that if it calls methods that do not have the needed permissions, the code will still be executed. We do not go more into this subject here because the subject is beyond the scope of this document.

3.4.6 Performing Access Control

The previous sections explained the different components of access control and in this section, we summarize the steps in performing access control. All the different components explained above are used together to finally make decisions about whether to let an operation take place when it is requested by code, services, or users.

The first step is the creation of security policies using the syntax described in Section 3.2. The administrator decides how he or she wants to protect the resources. The policy files contain a link to a keystore and a grant entry that contains a code source and some permissions. The code source is composed of a codeBase entry, which identifies the origin of code, a signedBy entry, which contains a series of aliases that refer to public keys in the keystore, and a principal entry, which refers to the authenticated principals from the JAAS authentication process. Finally, the grant entry contains a series of permissions that specify what the code source is permitted to do.

After writing the policy file, the administrator can start the system. This is when the policy files are loaded in memory and to do that, the administrator writes the name of the file in the `java.security` configuration file or he or she can write it in as a parameter on the Java command line. When the system starts, it verifies that the policy file has the correct syntax and reads the file to store it in memory. The steps towards performing access control are summarized in Figure 3.4.

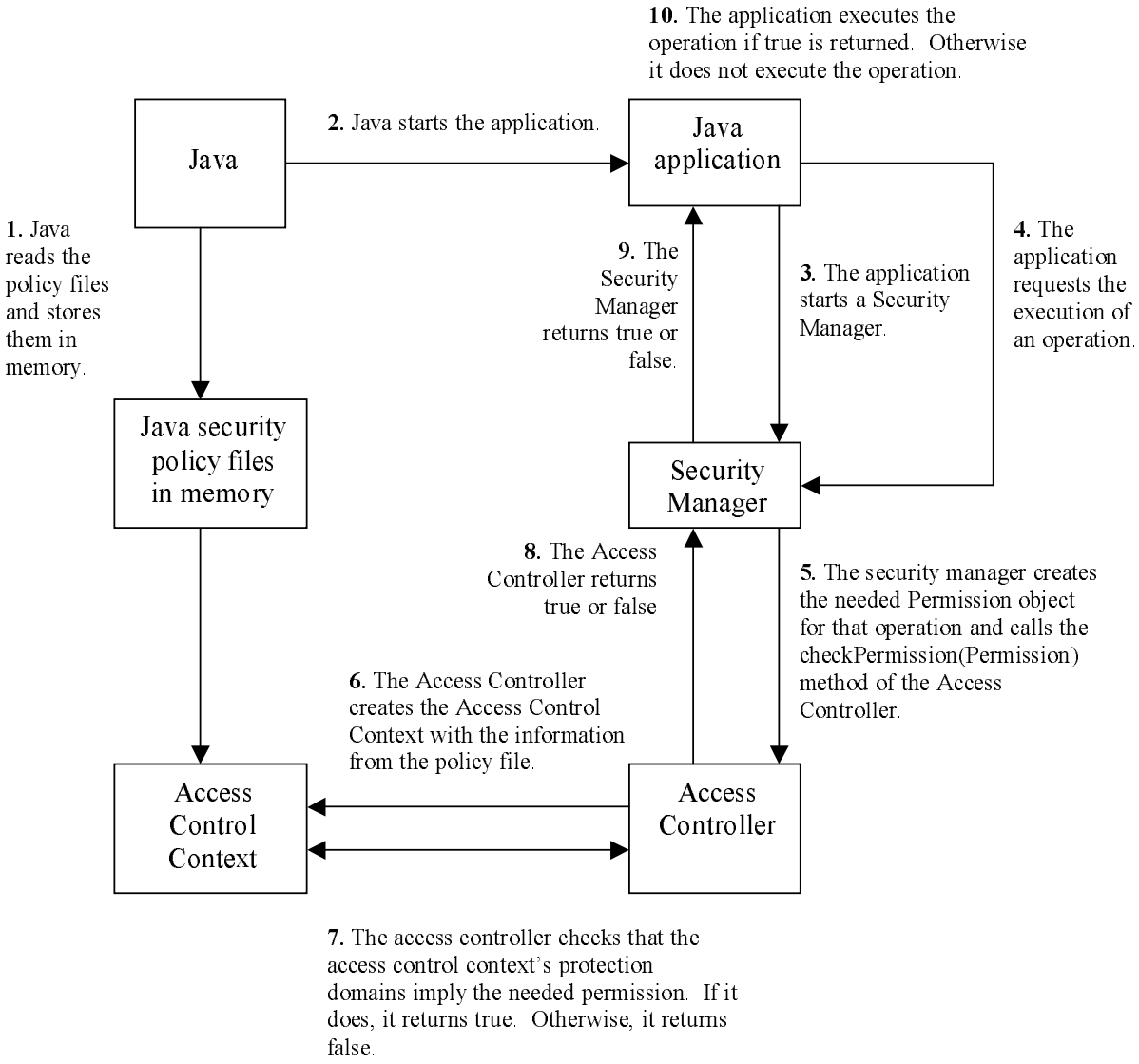

Figure 3.4: Access Control Operations

Step one is loading the policy file in memory. The system instantiates the permission objects from the permission entries of the policy file. For example, the java.io. FilePermission object can be created with the parameters "/exams/*" and "read". A CodeSource object is also created to represent the code source of the grant entry of

the policy file as well as the classes used to represent the principals in the code source. If those classes are not available to the virtual machine at the time when the system starts, the system will instantiate them when they become available.

The permissions are grouped together using their types in a PermissionCollection object. For example, all the permissions of the type java.io.FilePermission with the same code source are put together in a PermissionCollection object. This makes the search for types of permissions more efficient when access control must be performed.

The PermissionCollection objects are also grouped together by their code source. All the PermissionCollection objects of the same code source are put into a ProtectionDomain object.

The system can now begin the execution of the application (step two), which must start a security manager (step three). If the code is a downloaded Java applet then this is done automatically. When the application downloads code and tries to execute it (step four), the security manager verifies which permission is needed by the downloaded code to be executed. It then creates a Permission object of that type (step five) and calls the access controller's `checkPermission(Permission)` method (step seven) where the parameter is the Permission object that has just been created. This is where the access controller checks all the ProtectionDomain objects associated with the current code.

To do that, it creates the access control context. This is step six in Figure 3.4 and that step is described in more detail in Figure 3.5. It is created by checking the code source of the currently executing code against the code source of all the protection domains. All the protection domains that have a code source that implies the code source of the currently executing code are added to the access control context. In step seven of Figure 3.4, the access controller uses the access control context to verify that the needed permission is present in the protection domains. The access controller can then return true or false to the security manager (step eight), which sends the result to the Java application (step nine). The application then executes the operation if true is returned. If false is returned, the application does not execute the operation.

Figure 3.5 shows the creation of the access control context (see Section 3.4.5). This is done by first calling all the ProtectionDomain objects' CodeSource `implies` methods. This verifies that the ProtectionDomain objects' CodeSource object implies the code source of the downloaded code. If the `implies` method returns true, then that ProtectionDomain is added to the access control context. If the `implies` method returns false then the protection domain is ignored. If all of the ProtectionDomain objects'

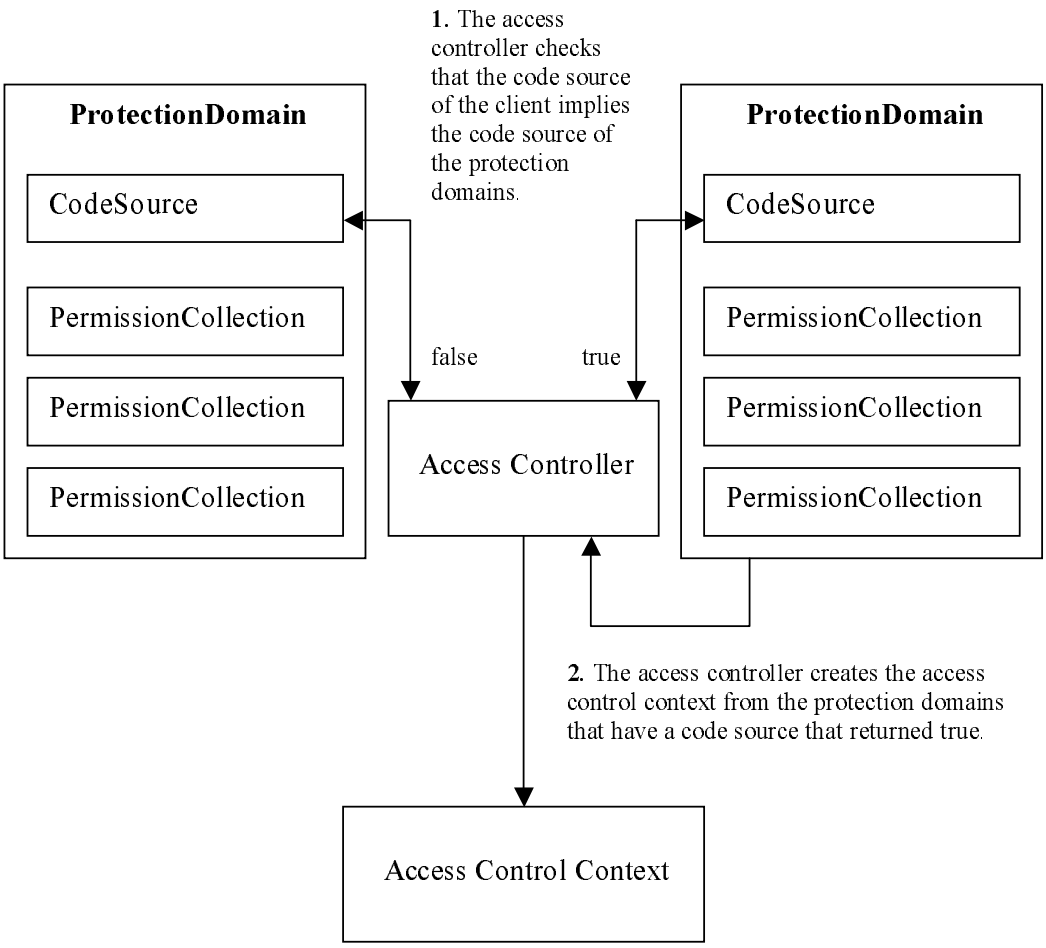

Figure 3.5: Creation of the Access Control Context

CodeSource objects are ignored then the access controller denies permission to the code because the access control context is empty.

On the ProtectionDomain objects that have been added to the access control context, the system calls the `implies` method of the PermissionCollection objects of each ProtectionDomain object. This verifies that the PermissionCollection implies the needed permission. To make this verification, the PermissionCollection object calls each of its permissions' `implies` methods. If they each return false then the PermissionCollection's `implies` method returns false. If one of them returns true then the PermissionCollection's `implies` method returns true.

The ProtectionDomain's `implies` method returns true if one of its PermissionCollection objects returns true. If all the PermissionCollection objects of the ProtectionDomain object return false then the ProtectionDomain's `implies` method returns false.

The access controller permits the action to be executed if the `implies` methods of each ProtectionDomain object return true.

Since the introduction of JAAS, the verification also considers the identity of the user or service trying to execute the code. The authentication must be done beforehand using any authentication protocol. The authentication protocol is programmed in a login module that is executed to verify the identity of the user. This is explained in Section 2.4.4. After this is done, the system has a Subject object that refers to the authenticated user or service. This Subject is attached to the Access Control context that is created by the access controller just before the verification of the ProtectionDomain objects described above. It contains the ProtectionDomain objects that are to be verified because they have the corresponding code source.

The sensitive operation that is to be executed is executed inside a `doAs` method that includes the name of the Subject that is trying to perform the operation. The subsequent steps in the access control verifications described above include verifications of the authenticated Subject against the principal parameters included in the code source of the policy files. If the authenticated Subject's principals are in the code source then the code source of the code can imply the code source of the policy file and therefore the ProtectionDomain object is verified. Otherwise, the ProtectionDomain object is ignored.

3.5 Conclusion

In this chapter, we looked at the Java access control mechanisms. We first studied the history of the security of the language. The first versions of Java offered less freedom in how to implement security in Java applications. JDK 1.0 for example only divided code into two types, remote code and local code. Remote could would be very restricted while local code was not restricted. This meant that it was difficult for developers to give more permissions to remote code or to impose restrictions on local code.

Improvements came with JDK 1.1 where remote code was divided into two types, signed remote code and unsigned remote code. Local code kept its unrestricted access and remote code was treated the same as local code if it was signed by a trusted signature, otherwise it was treated with the same restrictions as in JDK 1.1.

With JDK 1.2, there were some major improvements in the Java security model. Remote code was no longer divided into only two parts while local code could also be restricted. This was done with the introduction of protection domains. Protection domains are like custom sandboxes that restrict the code in different ways depending on their configuration. Local code and remote code could be put in the protection domains by creating Java security policy files. The policy files are written by the administrator or owner of protected resources. They contain a series of permissions that code can have depending on its origin and signers. When the system starts, those policy files are loaded in memory and then used to create the protection domains that impose the restrictions on the code.

Soon after JDK 1.2 was introduced, an optional package called the Java Authentication and Authorization Service (JAAS) was created. This added to the JDK 1.2 security model by adding the possibility of restricting the code not just on who signed the code or where it came from but also on the identity of who is currently executing the code. JAAS was then completely integrated in SDK 1.4, the latest version of Java.

We then studied the syntax of the policy files that are used to configure how access control is performed in the system. The policy files contain a series of parameters to do this. The keystore and the grant entry are the two main parts of the policy file. The keystore refers to the location where public keys used to verify the signature of the code are located. The grant entry contains the code source, which is used to identify the code, service, or user that this policy file will restrict followed by how it will restrict it. The code source contains a codeBase entry to mark the origin of the code, a signedBy entry to mark the list of one or more code signers. The signers are referred to by using

aliases that correspond to public keys in the keystore. Finally, the code source can contain a series of principals that refer to the authenticated users. The grant entry then contains a list of one or more permissions that tell the system which permissions this code source has.

Finally, we looked into how the system uses all these tools and information to perform the actual access control. It begins by storing the permissions in memory and creating a series of Permission, PermissionCollection, and ProtectionDomain objects, which work together with the Java security manager and access controller to verify the permissions at run time. A series of `implies` methods are called to verify the permissions.

Chapter 4

A New Security Policy Provider

After studying the important security properties that a distributed system must support, we looked into different distributed systems that could be used in our research. We chose to use the Jini network technology. We then looked into different tools that we can use to improve the security of distributed systems. Knowing that Java was created with security in mind, we looked at how the language enforces security and which security properties it can ensure in distributed systems. Now that we know how Java imposes security, we can find ways to improve this. We use the tools that we studied to find ways of improving the security of the language, especially in a distributed environment like Jini. This chapter explains our improvements to the security of the Java language, to the security of the Jini network technology, and to the security of distributed systems in general.

4.1 Introduction

To improve the security of Java and of Jini networks, we developed a new Java security policy provider. The standard security policy provider is described in the previous chapter. We improved the provider by adding support for new security properties that are not supported in the standard version. In addition to the already supported positive authorization security property, our new security policy provider supports negative authorization, exceptions, constraints, and delegation. We begin by describing each property and then we explain how they were implemented. Finally, a summary of the system that was developed in this research is given as well as a description of an example application that we developed to show the capabilities of our system. The goal was to

improve security in the Jini distributed system. We did that, but as it will be shown in this chapter, we are not limited to Jini only.

After studying the standard Java security policy provider and after studying various security properties from the Ponder Specification Language, we were ready to improve the original Java security policy provider with the security properties chosen from the Ponder language. This chapter explains the system that was developed. In addition to the authorization security properties, we also implemented a JAAS authentication login module to perform SSL authentication in our system.

4.2 New Security Properties

After studying how the Java security provider works in the standard Java virtual machine, we were able to find ideas to improve it. To do that, we studied the work of other research groups in the improvement of security in distributed systems. We studied SDSI / SPKI (Section 2.6.1) and the Ponder Specification Language (Section 2.6.2). We used parts of both specifications to create our system. SDSI / SPKI gave us the idea of the local name space. Instead of using global names on a network, we use local names so that clients on the network can decide by themselves which name they will use for themselves. More on this is given later when we describe our system in more detail. The Ponder language describes new security properties and we decided to implement some of them in our system. This section describes those properties. Section 2.6.2 explained Ponder security properties that we did not implement in our system and this section explains the Ponder security properties that we decided to implement in our system.

4.2.1 Positive Authorization

Positive authorization is already present in the standard version of the Java security policy provider. It is used to explicitly give permissions to users in a policy file. When a sensitive operation is about to be executed, the system's access controller checks to see if the user has a positive authorization entry granting him or her permission to perform this action. If the system finds one, then permission has been granted and the user may perform the action. Otherwise, the user does not have this permission and the action is not executed. Chapter 3 explains how this is done in the original Java policy provider. Our policy provider works in the same way, but by adding support for more specific positive authorization policies as explained in this chapter.

4.2.2 Negative Authorization

The negative authorization security property is the opposite of the positive authorization security property. It is used to explicitly deny the execution of certain actions. If the system's access controller finds that an attempted action on the system has a security policy entry that is a negative authorization, then the access controller denies the action to be executed. This means that the user trying to perform this action receives an error message and the action does not occur.

At first, this may look like this is the same as simply not putting a positive authorization for an action. It is similar, but having negative authorizations makes the system even more secure. For example, if an administrator wants to deny a user access to a hard drive, it would be possible to do that in the standard policy provider by not putting an authorization for that user on accessing the hard drive. Doing that would mean that the access controller, when seeing that the user wants access to the hard drive, will look for a positive authorization giving this user access to it and, by not finding one, would deny the access.

This can work, but if the administrator enters a negative authorization on accessing the hard drive, the access controller will see it immediately and denying access will be done faster. Also, if the administrator knows that this user is not permitted to access the hard drive and he or she puts this negative authorization in the policy file then it prevents the administrator from accidently giving the access later in the policy file. This is because negative authorizations have priority over positive authorizations. If both a negative authorization and a positive authorization exist for the same permission, then the positive authorization is ignored and the access is denied by the negative authorization.

4.2.3 Exceptions

Exceptions enable administrators to make permissions more precise. This idea is implemented in the Ponder Specification Language, but it is also an idea that was implemented in another system called the Java Secure Execution Framework (JSEF) [14]. The idea is to make positive and negative authorizations less general. In the standard Java policy provider, it is possible for example to give a user permission to read an entire directory. If the administrator still wants to keep some files in that directory confidential, he or she must either remove those files or give the user a positive permission on each of the nonconfidential files. This can be a lot of work, but exceptions

make it a little simpler.

The administrator can give a user permission to access a directory and tell the system which files in that directory are exceptions to that permission. The permission could be that the user is permitted to access all files in directory "example" except for files with the extension "txt". So instead of having hundreds of lines of code for the permission, the administrator can write just two. This also works for negative permissions.

It must be noted that exceptions are not the same as negative permissions. One could think that by putting the exceptions for the "txt" extensions, it is the equivalent of putting a negative permission on files with the "txt" extension. It is different. A negative authorization in the security policy completely denies the user from doing that action. An exception simply takes away the permission. It means that it is still possible for the user to perform the action if another positive authorization later gives it to him with no exceptions. The same is true for negative permission with exceptions. If the negative permission denies access to a directory but there is an exception on files with the "txt" extension then the user is still not permitted to access those files unless a positive authorization later on explicitly gives him access. Access is only given if a positive permission explicitly gives access with no valid exceptions.

4.2.4 Constraints

Our system also implements constraints. Constraints, like exceptions, enable administrators to make permissions more precise. While exceptions let the administrator be more precise on the target of the permission, constraints let the admintrators decide if a permission is valid or not based on outside factors valid at the time that the access is attempted. It is not always possible for an administrator to be present at every moment to add or to take away permissions. Constraints exist so that permissions become valid or invalid whenever necessary without outside intervention. In our system, we restrict ourselves to the current system date and time.

A constraint can make a permission valid or invalid at anytime based on the current system date and time. An administrator can already write a positive permission to access all files in a directory. But what if the permission should only be given to a user during weekdays? It forces the administrator to update the policy files on Monday morning to give permissions and then again on Friday night to delete permissions. To avoid this, it is possible for the administrator to write a policy so that it is only valid when the "Day of week" parameter is between "Monday" and "Friday" inclusively. So

if the user attempts to access files between Monday and Friday, it will work but if the user tries to access the files on Saturday or on Sunday, the access will be denied. Constraints can be implemented to control permissions on any type of time and date parameters. For example, constraints could be "anytime between 8 AM and 5 PM", "anytime during the year 2004", "only during the months of February and September", and "never when the day of the week is a Sunday".

4.2.5 Delegation

Delegation is the final new security property that we have implemented in this research. It enables administrators to let some users give to others some permissions that they have received from the administrator. This lets permissions be given without the administrator always updating the security policy file. This is a dangerous permission because, if not given to the right users, it could lead to some permissions being given to anyone. The administrator must be very careful with this permission.

There are three basic settings of delegation permissions. Firstly, there is the "no delegation" setting. This is the equivalent of a negative authorization on a delegation. When the access controller sees this, it is not possible for the user to perform the delegation even if elsewhere in the security policy, it is written that this user can delegate this permission. Secondly, there is the absence of a delegation permission. This does not permit the delegation to take place, but it does not deny it either. This means that if the access controller sees this, it will not let the user perform delegation unless elsewhere in the security policy, it is written that the user can do it. If a positive delegation permission on this permission is given later on, then the user is permitted to perform delegation on this permission. This is the third delegation setting. It means that a user has the permission to perform a delegation. It is divided into two parts. One is "single delegation" and the other is "multiple delegation". In single delegation, it lets the user perform one delegation on the permission and the users who receive this delegation can use it but cannot delegate the delegated permission. It is a one-step delegation. Multiple delegation lets a user delegate a permission and lets the user who has received the delegated permission delegate it too. In this case, it can be delegated any number of times.

Finally, delegations are always permissions that are not permanent. The administrator decides for how long a delegation can be valid once it has been delegated. A user delegating a permission sets the number of seconds that the permission will be valid for the next user. The number of seconds must be less than or equal to the number of seconds chosen by the administrator. Once the time has passed, the delegation is no

longer valid and the user no longer has those permissions.

4.3 Implementing Authentication

The authentication component of the system is based on JAAS. As described in Section 2.4.4, JAAS is a framework based on Java for creating and using different authentication and authorization protocols in a system. It is possible to use the already built-in login modules or to create our own login modules. We decided to create our own login module because JAAS was not built to support the local name space that we are using. As said earlier, we do not use a global name space in our system. Each client is free to choose its own name even if the name already exists elsewhere on the network. Authentication protocols that are regularly used in JAAS such as Kerberos are not fit for that kind of system but the SSL authentication procotol could be used so to implement SSL authentication in our system, we created a new JAAS login module based on that protocol.

The SSL protocol algorithm is described in Section 2.5.2. As explained in that section, the SSL protocol does not have to perform authentication on both the client and the server. The only requirement is that the client authenticates itself on the server. In our system, we chose to make authentication bidirectional. This means that the client authenticates itself on the server and the server authenticates itself on the client.

The result of the SSL authentication using the JAAS login module is that the client knows for sure the identity of the server and the server knows for sure the identity of the client. This is typical authentication. After the process is over, the two parties can communicate with each other absolutely confident that they know who the other participant is. Another result of the authentication protocol of SSL in our system is the creation of a symmetric encryption key. This is explained in more detail in Section 2.5.2. During the SSL authentication mechanisms, the client and the server exchange data and from that data, they are able to create a key that only they know. They both use this key to encrypt and decrypt the data that they exchange. To use the encryption key, the client and the server create secure sockets. Sockets are Java objects that other objects use to write information on them. When it becomes possible, the information is sent to whoever it supposed to receive this information. The secure version of sockets in Java encrypts all the data that is written on them using an encryption key. The encryption key is given to the sockets that are created immediately after the authentication process is over. That key is the SSL symmetric key created during authentication.

By using the symmetric key and the SSL protocol, we ensure the authentication security property along with the confidentiality security property.

To perform authentication using SSL, the clients and the servers need a public and a private key. In our system, we use the Java key generation tool. The Java package that we download from Sun Microsystems includes a program called "keytool" that can be used to create public key pairs of any size using the RSA algorithms. Each client and server on the network that uses our system is required to generate a public key pair (a public key and a private key). This is the key that can uniquely identify participants on the network. Obviously, only one participant on the network can own each public key pair.

Those keys are used to perform authentication and the authentication algorithm results in a symmetric key that is used to encrypt all the data that is transferred on the network. But the SSL protocol requires more than just a public and a private key to fonction. It needs certificates. The certificate is what is transferred between the client and the server and between the server and the client to perform authentication. The certificate contains among other things the public key of the owner of the certificate. The SSL protocol in our system requires a standard type of certificate. We chose the X.509 standard for certificates, which contain the following:

- The version number of the X.509 certificate.
- A unique serial number for the certificate.
- The name of the algorithm used to create the certificate (RSA).
- The certificate issuer's name. In our case it is SPKS (see below).
- A validity period. This is the date and time when the certificate begins to be valid and the date and time when the certificate stops to be valid.
- The subject's name. This is not important in our system, but it is a good idea to include some valid information here anyway. We include a common name (CN), an organizational unit (OU), an organization (O), and the country (C).
- The subject's public key with the algorithm used to create this key.

We chose X.509 because it is a standard. It is simple to create and then to use with SSL. To create the certificates, we use an external program that we put on the network as a service we called "Signed Public Keys Service" (SPKS). Different programs are

used together to start an SPKS. As shown in Figure 4.1, the idea is to send a public key to the SPKS, have the SPKS create and sign a public key certificate in the X.509 format, and then return that certificate to the owner of the public key. We do not go into details about how the SPKS works in our system specifically because it does not matter what system is used to create the certificates, as long as they are of the X.509 format. The only thing to make sure of is that the public key of the SPKS is correctly and securely distributed across the network because when clients and servers on the network receive a certificate and want to verify that it is valid, they check the dates of the certificates but also the signature and the only way to check the signature is to have a valid copy of the SPKS public key. Ways to distribute this key are numerous, as long as it is secure. Ideas include sending the key by email, by fax, or by telephone. It is not always convinient, but it is secure. Once the client or server has received this key, it can verify any SPKS certificate it receives. If the certificate is found to be invalid, authentication fails.

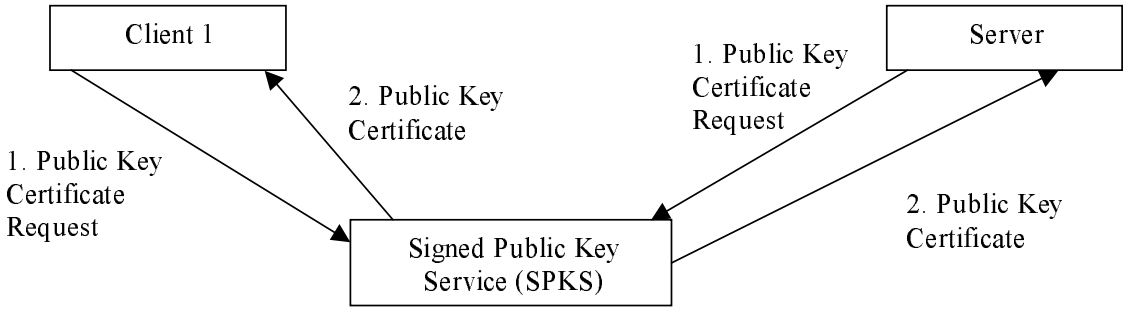

Figure 4.1: Signed Public Key Service's Communication

In summary, authentication is one of our four basic security properties that a distributed system must implement to be secure. In our system, it is performed by the client on the server and by the server on the client. Only after authentication has occured can the desired communication begin. The protocol used for authentication is the Secure Sockets Layer (SSL). The latest version of Java includes a tool called the Java Authentication and Authorization Service (JAAS), which offers a method of performing authentication in Java applications. JAAS offers built-in login modules and offers the possibility of creating our own login modules. The login module is a description of the interface of the chosen authentication protocol. We created a login module that implements the authentication protocol that we have chosen.

The clients and servers of our system begin by generating their public key pair. They send their public key to the SPKS. The SPKS receives the public key, creates an X.509 certificate from it, and sends it back to the client or server. The SPKS is external to our system, it can be anything as long as it is possible to receive public keys and generate certificates from it. The system must also make sure that the SPKS public key

is securely distributed so that all the participants on the network can verify certificates signed by the SPKS. After the client and the server have done this, they can begin the authentication process. The algorithm described in Section 2.5.2 is executed. The result from this is that the client and server know with certainty the identity of the participant with who they are communicating and they both have a secret symmetric key that only they know. With that key, they create secure sockets and all the information that is to be transmitted between the two participants passes through those sockets for confidentiality.

4.3.1 Global Name Spaces vs. Local Name Spaces

As we mentioned above, our system uses a local name space instead of a global name space. We explain here what we mean by local name space and why this decision was made.

In typical networks, participants are known as a unique global name. For example, Alice communicates with Bob, who sends this information to Charlie. This is interesting for the administrator of the network as well as for each participant. It is simple for the participants on the network to find other participants simply by using their names. Administrators can create simple security policies by using the participants' names. The problem with this method is that each name must only appear once on the network. If the network already contains a participant called Bob, another Bob cannot suddenly appear. This can be a problem in certain types of networks because it creates a dependence between the clients of the network. Before Bob can call himself Bob, he must look at every other participant on the network to make sure that no one else is already using that name. Only after this verification can Bob use the name.

The SDSI / SPKI specification proposes the use of a local name space. This makes the participants of the network more independent from one another. Each client and service is free to choose whatever name it wants, even if the name is already on the network. This is interesting because it eliminates the need for each participant to know the names of all the other participants. It also enables clients from other networks to join our network without changing anything in their configuration. We chose this method to make each participant on the network as independent as possible from all the other participants on the network.

Even with the local name space, we still have a method to uniquely identify each participant. This is necessary in the creation of the security policies. We use the public key of the participants to identify them in the security policies. An administrator

writing a security policy uses the public key of the participant that is to be restricted by the current security policy to identify who must be affected by that security policy.

The use of a local name space made the choice of SSL as an authentication protocol to use in our system more obvious. Other security protocols depend on the global names of participants for identification. For example, they often use user names and passwords to perform authentication. SSL does not use that. The protocol uses keys for performing authentication so it was an obvious choice for authentication in our system of local names.

4.4 The System's Architecture

This section explains how the system works and the order of the normal operations of the system. As shown in Figure 4.2, the first step in the system is normally the creation of the security policy. The security policy is written using the syntax described in Section 4.4.1.

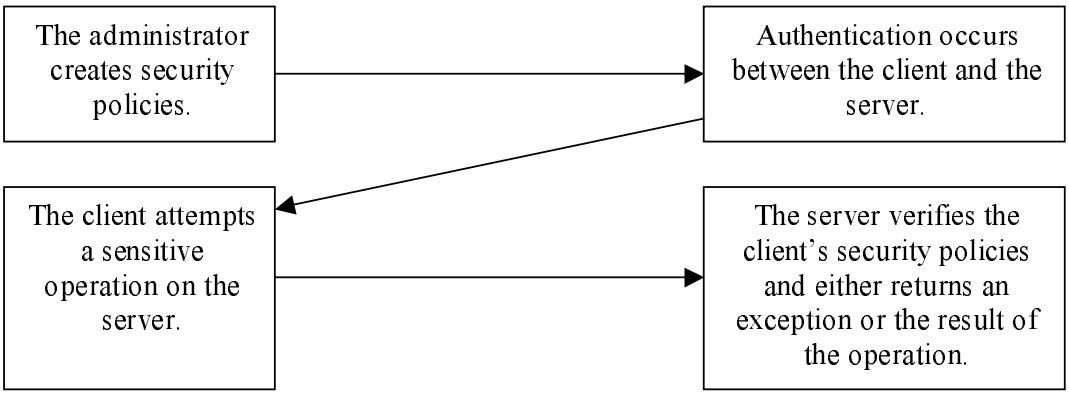

Figure 4.2: The System's Normal Operations

Once the policy has been created, the system is started, the security policy is loaded in memory and then clients can connect to the server. Before they can use the server, authentication is performed between the client and server as described in Section 4.3. This is normally followed by the client requesting a particular action from the server. If this action is restricted, the server checks the security policy to make sure that the client is permitted to perform this action. How this is done is explained in Section 4.4.2. If permission is granted then the server executes the operation and returns the result. Otherwise, the action is not executed and the server returns an exception.

4.4.1 XML Policy File Syntax

Before going into detail about how our Java policy provider works to enforce security properties in our system, we look at the contents of a Java policy file written for our system. Appendix A shows the complete structure of the policy file. In this section we look at the different parts of the file to explain what they mean and how they are used.

We made the decision to use the eXtensible Markup Language (XML) [29] for the creation of the policy files in our system. The original Java policy file does not use XML but we decided it was ncessary to make our policy files more structured because it contains more information. While the original policy file permitted the creation of positive authorizations, our policy file permits the creation of security policies using all the different security properties described at the beginning of this chapter.

Another reason to use XML is the fact that it is relatively easy to verify the syntax of XML documents. Our security provider code contains Document Type Definition (DTD) information to describe the syntax of the file. Before the file is read, the provider checks the file's syntax using the DTD to make sure that it is correct. If it is incorrect, the file is not read and the system stops its execution. If it correct, the policy can be placed in memory. The verification of the syntax is done very quickly using XML with DTD.

The XML policy file contains a series of tags used to describe the security policy. It begins with the tag `Policy` used to note that this is a policy file. Next, the keystore of the system is given if necessary using the tag `keyStore`. This is the same as the original version described in Section 3.2 except that it is written in XML using the tags `keyStoreLocation` for the URL of the location of the keystore and `keyStoreType` for the type of the keystore.

Next in the policy file, we include information on groups if necessary. Groups are a way to put together a series of principals. During the creation of the policy file, we may want to say that the current policy applies to many different users and not just one. To do this, we create a group, which contains a series of users and later use that group to tell the system that the policy applies to all the members of the group. Many groups may be specified in one policy file, they are written in the tag `groups`. That tag contains one or many `group` tags, each representing a different group. The `group` tag contains one `groupName` tag, which specifies the name that will be used to refer to this group and a `members` tag, which contains the list of members for that group.

The members of the group are called principals. A principal represents a user's

identity or a system's identity. One user or one system may have many identities and each of them is a principal. The `principals` tag inside the `group` tag is used to include the series of principals that are part of the group. Each principal is written inside a `principal` tag, which contains the `principalClass` tag and the `principalName` tag. The `principalClass` tag contains the name of the class that is used to read this type of principal and `principalName` is the name of the principal read by that class.

Next begins the `grant` entry. This is divided into two parts, the `subject` and the `permissions`. The subject is information regarding who is to be affected by the security policy of the `grant` entry. As explained in Chapter 3, the system knows who is affected by a security policy using three criteria. They are refered to using the tags `codeBase` (the origin of code using an URL), `signedBy` (the downloaded code signers using aliases found in the keystore mentioned above), and `principals` (user identities). This is the same as explained in Section 3.2 except that it is written using XML. More than one signer can be present in the signedBy tag. We use one or many `signer` tags to specify the aliases of each possible signer. Our system adds support for groups so the `subject` tag may contain a `groupName` tag, which contains the name of a group defined above. The `grant` entry will then affect all members of the group.

The second part of the `grant` entry is the `permissions` tag. This is where the permissions for the subject are given. All the positive and negative permissions with their exceptions, constraints, and delegation information are given here. We have a `permissionPos` tag for positive permissions and a `permissionNeg` tag for negative permissions. They have the same syntax except that the `permissionPos` tag contains information on delegations and `permissionNeg` does not. Both tags contain a `permissionClass` tag to specify the name of the class of the current permission. One `target` tag can be present to represent the target that is controlled by the permission if necessary. One `actions` tag can be present to represent one or more actions each represented with an `action` tag. The tag `constraint` can be present when a permission contains constraints. This tag is described in more detail later. Finally, the `permissionPos` and `permissionNeg` tags can contain an `exception` tag when the permission has an exception. The `exception` tag has the same syntax as `permissionPos` and `permissionNeg`.

The last part of the `permissionPos` tags is the information on `delegation`. The `permissionNeg` tags do not contain delegation information because negative authorizations cannot be delegated. This is to prevent users from giving to other uses some negative permissions that would take away their positive permissions since negative authorizations have priority over positive authorizations.

The `delegation` tag contains a `deleg` tag, which is either set to true or to false. This tells the system whether delegation should be permitted or not. Next the `deleg Delegations` tag is present to tell the system whether the delegated permission can be delegated. This is recursive delegation. If this is set to true, a user receiving a delegated permission may be able to delegate it also. Delegations also include a time limit. This is the `timeLimit` tag and is given in seconds and refers to how long a delegation can be valid once it has been delegated to someone else. A user cannot delegate this permission for longer than the amount of time written here. Finally, delegation may contain constraints using the `constraint` tag. This specifies when a delegation permission is valid.

The `constraint` tag is more complex. It is present in the `permissionPos`, `permissi onNeg`, `exception`, and `delegation`. This supports the constraint security property mentioned at the beginning of this chapter. It tells the system when the permission, exception, or delegation should be ignored and when it should not be ignored. To do that, it uses a series of tags. They are `dateTime`, `comparator`, `year`, `month`, `day`, `dayOfWeek`, `hour`, `minute`, and `second` as well as the Boolean operations `and`, `or`, and `not`. Together the tags make it possible to write powerful contrainsts to give to the administrator a lot of control on when a permission, exception, or a delegation should be considered or not.

To write one constraint, the administrator uses this `constraint` tag with the `date Time` tag. That tag contains only the information that the administrator wants to control. For example, the administrator may want a permission to be valid only on Mondays so the tag looks like this:

```
<constraint>
   <dateTime>
      <comparator>==</comparator>
      <dayOfWeek>Monday</dayOfWeek>
   </dateTime>
</constraint>
```

All the parameters that are not present in the `constraint` tag are ignored. The comparator is used as follows:

- The symbol "==" means "Equal".
- The symbol "!=" means "Not equal".

- The symbol "<" means "Less than".
- The symbol ">" means "Greater than".
- The symbol "<=" means "Less than or equal to".
- The symbol ">=" means "Greater than or equal to".

The first day of the week is Sunday and the first month of the year is January so the following example means that the permission is valid on Monday and Sunday in February and January. The permission is ignored when the current system time does not corrspond to these two constraints.

```
<constraint>
   <and>
      <dateTime>
         <comparator>&lt;</comparator>
         <dayOfWeek>Monday</dayOfWeek>
      </dateTime>
      <dateTime>
         <comparator>&lt;</comparator>
         <month>February</month>
      </dateTime>
   </and>
</constraint>
```

Notice the `and` tag is used to say that there are two constraints to consider. The `or` tag works in the same way to specify that one constraint or the other must be true for the permission, exception, or delegation to be valid. We can also use the `not` tag to specify that the constraint must be false for the permission, exception, or delegation to be valid.

Constraints like these can be written for any combination of dates and times as necessary.

4.4.2 XMLPolicy: A New Java Policy Provider

To implement the new security properties explained at the beginning of this chapter, we first write security policies that use them using the syntax presented in the previous

section. Then we start our system that reads those policies to load them in memory to finally use them to make authorization decisions when the client requests the execution of sensitive operations. This is the part explained in this section and the next one. We first look at how we created the policy provider and the next section looks into how the provider works to make authorization decisions.

When the system starts, the first step is to load the XMLPolicy Java policy provider. This is our system. We call it XMLPolicy because we use XML for writing the policy files used in the system. Figure 4.3 shows the first steps of the execution of XMLPolicy. It begins by loading the appropriate security policies using the PolicyParser object created by XMLPolicy. PolicyParser is a class that we created that is called by XMLPolicy. It examines the syntax of the policy files to make sure that they are correct. If the syntax is correct, it then loads its contents in PolicyEntry objects in memory.

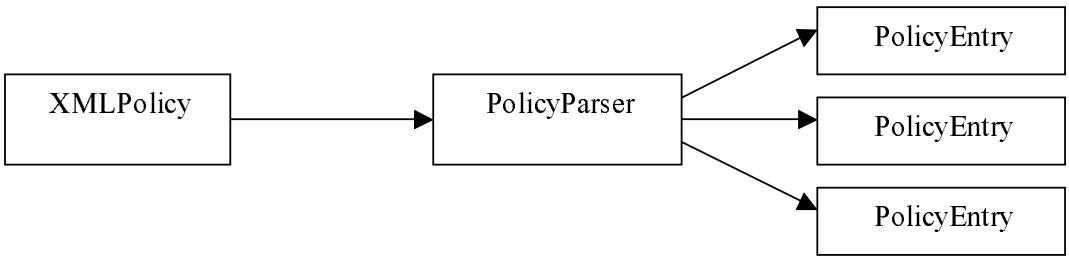

Figure 4.3: Summary of the Steps in Creating the System's Data Structures

The PolicyEntry objects contain the information of the policy files. Each Policy Entry object represents a different `grant` tag in the policy file. As shown in Figure 4.4, the PolicyEntry object contains two main objects, GroupSubjectCodeSource and PermissionCollections. Both of these objects contain other objects to describe the `grant` entry of the policy file.

This is the contents of a GroupSubjectCodeSource object:

- CodeBase: This is zero or one URL objects that represent the origin of downloaded code. If an URL is present, only code downloaded from that URL is affected by the security policy. If no URL is given then the system interprets this as code from any origin.
- Signers: This is an array of Certificate objects that contain the public keys of possible code signers. PolicyParser reads the aliases from the policy file and from the keystore, it finds the corresponding public keys and enters them here. If signers are given then the downloaded code signatures are verified using the

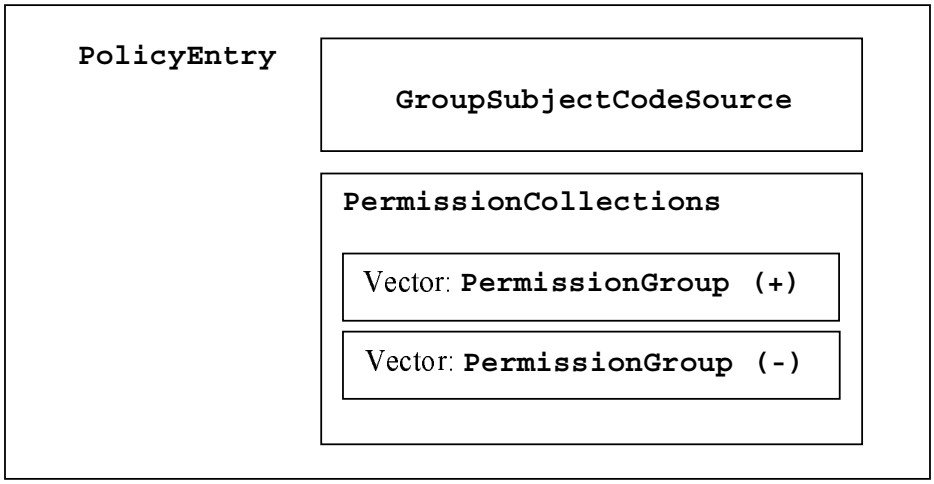

Figure 4.4: The Contents of the PolicyEntry Object

public keys in the keystore that correspond to those signers. The security policy only affects the code if all the signatures can be decrypted with all the public keys associated with the signers specified here. If no signer is given then the system ignores the signature of the downloaded code.

- Principals: This is zero, one, or many principals that represent authenticated users. The principals are grouped in a PrincipalCluster object that we created. If principals are present, the security policy only affects the users that have been authenticated as each principal in the principal cluster specified here. If no principals are given then users do not need to be authenticated to be affected by the current `grant` entry's security policy.

- Groups: This is zero, one, or many groups that each represent a series of principals. The system reads the name of the group written in the policy file and gets the name of the principals that are part of the group and puts them here. Each group is put in a PrincipalGroup object that we created. They each contain a name and a PrincipalCluster object that contains the names of the principals that are members of the group. All members of the group are affected by the same security policy. If no group is given then no group can be affected by current `grant` entry's security policy.

The second part of the PolicyEntry object is a PermissionCollections object. That object contains the permissions of the current `grant` entry with two PermissionGroup objects, one for the positive permissions and one for the negative permissions. The positive and negative permissions are separated to efficiently perform access control.

This is the contents of the PermissionGroup objects:

- Permission: The permission is represented by an object that extends the Java class Permission. The object represents the actual permission of the PermissionGroup object.
- Exception: This is an array of Permission objects, which represent each exception that is to be applied to the current PermissionGroup object. If there are no exceptions, this parameter can be empty.
- Delegation information: This is a DelegationInfo object that we created. It represents the permissions on delegation for the current PermissionGroup object. These are permissions on delegation, on recursive delegation, on the maximum amount of time in seconds that a delegation can be valid, and on the constraints related to this delegation. This parameter may be empty when the policy file did not specify any delegation permissions for this permission.

The above description does not take into account all the improvements of our security provider. The original version of the Java policy provider supports positive authorization. In our version, we support positive authorization as well as negative authorization, exceptions, delegation, and constraints. This means that permissions and exceptions may contain information on constraints, in which case we cannot use the standard Java Permission object to represent them. Therefore, we created a ConstraintPermission object that contains the standard Permission object with a Constraint object to represent permissions and exceptions that have constraints. This can be used in the place of the permissions and the exceptions above.

We also created an UnresolvedPermission class for permissions that must be loaded in memory but cannot be resolved yet because the implementations of those classes have not been loaded in memory by Java. When this happens, our system places the permission in the UnresolvedPermission object. When the permission must be checked later, our system finds the permission in this object and instantiates it at this moment to verify the permission. This is explained in Section 3.4.1 because it is present in the original Java specification. Our class works in the same way.

This section looked into the data structures that we created to store in memory the security policies written by the administrator. The information in the policy file is stored in memory to be checked when necessary when performing access control. These classes work together to store the policy files' data and then to perform the access control. The next section explains how this access control is performed.

4.4.3 Security Policies Verification

This section explains how our system performs access control when a sensitive operation is attempted by a client on the server. The procedure for access control in our system is similar to the procedure for access control in the original Java policy provider explained in Chapter 3. To support the new security properties, we created our own Java policy provider called XMLPolicy. The Java access controller calls XMLPolicy when it must check for a permission before letting an application execute an operation.

The policy provider that Java will use with an application is specified in the `java.security` file. This file contains security configurations for the system. The option to choose which policy provider to use is called `policy.provider`. The full name of the policy provider that we created is `ca.ulaval.lsfm.djvm.XMLPolicy`. So that name must be entered in the security configuration file for the option `policy.provider`. When Java sees this, it will not use the original policy provider to load the policy file in memory and to perform access control on the sensitive operations performed on the system. Instead it uses XMLPolicy.

As described in Section 3.3.1, the policy file to load in memory is specified in options either at the command line or in the `java.security` configuration file. In our system, the PolicyParser object verifies that the syntax of the policy files is correct and then loads the information of the policy files in memory in the objects described in the previous section. Now a client can request the execution of an operation and the security policies verifications can begin.

In Java, a security manager can be installed when the application starts. In applications running with our system, the security manager is always started. It verifies that the operations being executed are permitted to be executed before they start. To do that, it calls the access controller that checks the permissions currently loaded in memory. More precisely, the security manager creates a Permission object of the type of the needed permission. It then calls the access controller's `checkPermission` method with that Permission object as a parameter. The access controller can then perform the access control by calling the `implies` methods of the classes of the XMLPolicy policy provider with that Permission object.

What happens is that the access controller verifies each PolicyEntry's Group SubjectCodeSource object to see if it implies the GroupSubjectCodeSource of the user or service requesting the operation. If none of the GroupSubjectCodeSource objects return true then the permission is denied. Otherwise, the system continues its checks in the PolicyEntry objects that have a GroupSubjectCodeSource that returned true.

For each GroupSubjectCodeSource that returns true, the system calls that Policy Entry's PermissionCollections `implies` method. That method uses the following algorithm.

```
1.  Negative permissions are checked...
   2.  If a negative permission implies the permission...
      3.  Check that negative permission's exceptions...
      4.  If no exception imply the permission, return false.  End.
      5.  If one of its exceptions implies the permissions continue (1).
6.  Positive permissions are checked...
   7.  If a positive permission implies the permission...
      8.  Check that positive permission's exceptions...
      9.  If one of its exceptions implies the permissions continue (6).
   10.If no exception imply the permissions, return true.  End.
11.  Return false.
```

As the algorithm shows, the negative permissions are checked first. This is because negative authorizations have priority over the positive authorizations. If the same permission is present in the policy file as a negative permission and as a positive permission, the permission must be denied.

When the negative permissions are checked (step 1), the algorithm calls each negative permission's `implies` method. If the negative permission does not imply the needed permission, it is ignored and the system checks the next negative permission. If all the negative permissions are ignored, the system begins its checks of the positive permissions. If it finds a negative permission that implies the needed permission (step 2), it checks its exceptions (step 3). If none of its exceptions imply the needed permission then the algorithm returns false (step 4) and the permission is denied because the system has found a negative permission that explicitly denies the execution of the desired operation. If the exception implies the needed permission then it means that the negative permission is to be ignored. The system then continues its checks with the other negative permissions present (step 5). When they have all been checked and found not to explicitly deny the execution of the desired operation, the system continues its checks with the positive permissions.

When the positive permissions are checked (step 6), the algorithm works in the same way. It looks for positive permissions that imply the needed permission. If none is found then the system returns false (step 11) and the operation is denied. When it finds one (step 7), the algorithm checks the positive permission's exceptions (step 8). If

one of the exceptions imply the needed permission (step 9) then the algorithm ignores the positive permission and continues its checks with the other positive permissions. If none of the exceptions imply the needed permission, the algorithm returns true (step 10) because it has found a positive permission that explicitly gives permission to execute the operation.

If all the positive permissions that imply the needed permission have at least one exception that implies the needed permission, then the system returns false (step 11) and the operation is not executed.

The result of this algorithm is either true or false. If PermissionCollections finds that its PermissionGroup objects imply the permission then it returns true, otherwise it returns false. That result is sent to the PolicyEntry's `implies` method, which returns the result to the access controller. The access controller receives the result of each PolicyEntry's `implies` methods. If one of them returns true then it returns true to the security manager and the operation is executed. If all of the PolicyEntry objects return false then the access controller returns false to the security manager, which denies the operation from being executed. An exception is then thrown in the application.

Using our system, the security policy can be much more powerful than with the original Java policy provider. The series of `implies` methods can check much more than simple positive permissions. They check that the positive permissions do not have exceptions or constraints on them that make it so that they should be ignored. The negative permissions enable the administrator to specify which operations should be explicitly denied. They make the creation of secure policies easier.

Our system also can support delegation. Clients can send to other clients some of their permissions. When this is done, a new PolicyEntry object is created from the original one with the delegated user's GroupSubjectCodeSource information. The system checks that PolicyEntry object along with the others during access control.

4.5 An Example: Secure Calculator Application

After developing the XMLPolicy Java policy provider, we created an example application to demonstrate the capabilities of our system. The application is a distributed calculator that can perform simple arithmetic operations. This section describes this application and how to use it without becoming a user's manual for the application. We look into the different components and how they communicate together. We look

at the requirements for running the distributed calculator. We do not go into details about how to start and use the application.

4.5.1 Prerequisites

The calculator works with one server and at least one client. The server has a public key to uniquely identify it on the network but we call it Alice. It is a local name but on our network, it is still unique because there are only three participants. Two clients, called Bob and Charlie, were created to communicate with Alice to request the execution of some arithmetic operations and to delegate permissions. The original goal being to make the Jini network technology more secure, we created the application to work with that distributed system but the security improvements that we created in our Java policy provider are not limited to Jini.

Because it is a Jini network, the network running the calculator application must have a lookup service with at least one client and one service. All three participants can run on the same machine or on different machines. The first step is to create encryption keys on each participant. We do not go into detail about how we did this because the method used is not important as long as it generates valid and unique RSA keys. The Java key generation program called "keytool" can be used to do this. It is the one that we used.

Next, each participant must validate their public key with the Signed Public Keys Service (SPKS). The participants create a certificate request certificate using their public key. Again, it does not matter which tool is used to do this as long as valid certificate request certificates are created. The Java program called keytool can do this as well. It is the one that we used. The certificate request certificate is sent to the SPKS, which extracts the public key from the certificate to create a new certificate called a public key certificate and signs it. That certificate is used to transmit public keys between the participants of the network. The participants receiving public key certificates can verify that those certificates are valid using the certificate's signature.

Somehow, the public key of the SPKS must be shared among all the participants so that the participants can verify the signatures. We can do this by email, telephone, or any other secure method. In our case, we transmitted that key in person.

The SPKS is not something that we programmed. Networks running our system can use any type of SPKS as long as it can receive public keys, sign them, and return them to their owner. In our case, we used a combination of three programs called Apache

Ant [2], Enterprise Java Beans Certificate Authority (EJBCA) [9], and JBoss [25] to create an SPKS service. We do not go into detail about how they work here.

4.5.2 Running the Calculator Application

After the keys have been created on each participant and a public key certificate has been acquired by each participant, communication can begin. First, the lookup service is started along with an HTTP server. We use the built-in lookup service and the built-in HTTP server from the Jini packages. Once the lookup service is started, the clients and services can communicate with it. The HTTP server is used by clients and services that want to download code from the lookup service.

The first step after that is to start Alice, the service. Alice offers to the network a calculator to execute simple arithmetic operations. An HTTP server must be started on the server as well. This is because sometimes clients may want to download the code to their machine to execute it. When Alice is started, it creates a service proxy. That proxy is registered on the lookup service and later used by clients to connect to the service. Alice also loads our policy provider (XMLPolicy) along with the security policy designed to control the actions on it. This is the policy provider that is used to load the policy in memory and then to perform access control on the actions of the clients.

After starting the server, the client (Bob) can start its execution. The first thing it will do is connect to the lookup service and ask for access to the calculator application. If the service has correctly registered itself on the lookup service, it will be possible for the lookup service to either give the proxy to the client or tell the client where to find it. Using the proxy, the client contacts Alice.

The first step in the communication here is authentication. The client authenticates itself on the service and the service authenticates itself on the client. This is done using our own JAAS login module programmed with the SSL authentication protocol. In this case, Bob authenticates itself on Alice and Alice authenticates itself on Bob. If both if these authentications are successful then communication can continue. Otherwise, it is halted. If the authentications are successful then the client can begin to use the calculator as shown in Figure 4.5. Bob can now use the calculator because he has been authenticated.

As shown in Figure 4.5, the client may choose which types of numbers to use in the arithmetic operations. The choice is either integers (Integer), long integers (Long), real

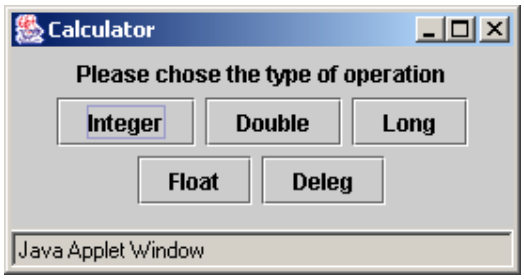

Figure 4.5: The Start of the Calculator Application

numbers (Float), and long real numbers (Double). It is also possible for the client to delegate one of his or her permissions (Deleg). We look into this option later.

After choosing which types of numbers to use, the client is shown Figure 4.6. The client enters the first number of the arithmetic operation using the buttons on the calculator. Next the client chooses the type of operation to perform. This is either addition (+), substraction (−), multiplication (∗), or division (/). Finally, the client enters the second number of the operation and presses the equals (=) button to get the result.

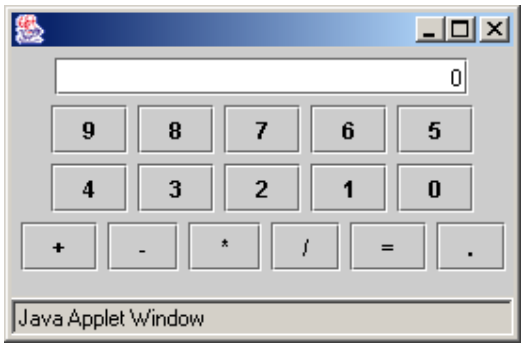

Figure 4.6: The Calculator

The result of the operation is only displayed if the client has the permission to execute the operation. This is where XMLPolicy is used. When the client presses the "=" button, the system verifies that the client is permitted to execute the chosen arithmetic operation with the chosen numbers. This verification is performed using the methods described in this chapter. If permission is denied then the result is not

displayed.

As shown in Figure 4.5, it is also possible for the client to decide to delegate a permission. By pressing the "Deleg" button, the client is shown a list of permissions that he or she is permitted to delegate. The system searches the client's current permissions and shows the permissions that can be delegated. This is shown in a window such as the one shown in Figure 4.7.

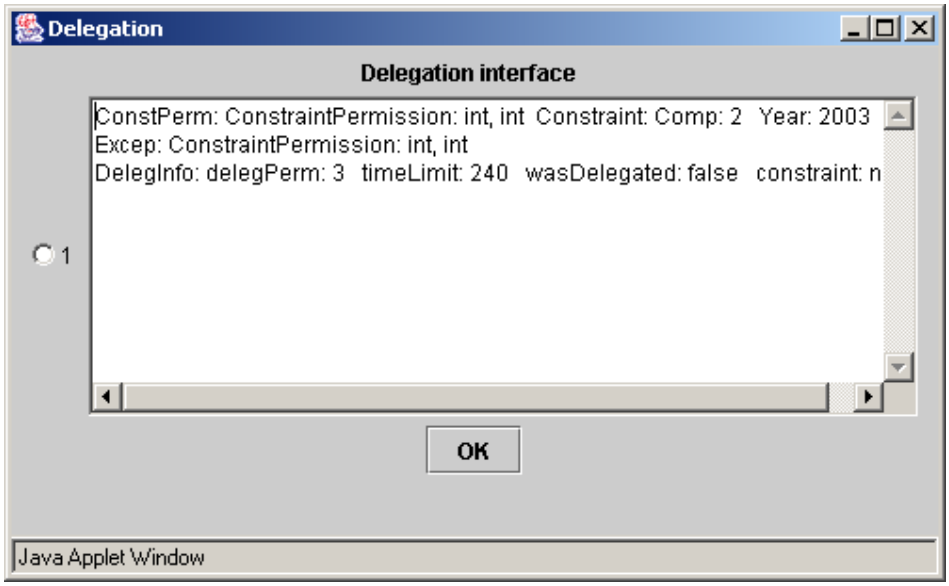

Figure 4.7: Delegating a Permission: Part 1

Using the buttons at the left of the window, the client chooses one permission to delegate. In the example, there is only one permission, but there can be many. Only those permissions the the client is permitted to delegate are shown. After choosing a permission to delegate, the client presses the OK button. The client must now answer some questions to specify how he or she wants the delegation to work.

The questions are asked in another window shown in Figure 4.8. The client must first tell the system who the receiver of this delegation is. This is done by entering the public key of that user. The client then decides which type of delegation this is going to be. The choices are:

- No Delegation: This means that whoever receives this delegation cannot delegate it.
- Delegation: This means that whoever receives this delegation can delegate it but those users who receive the delegation of the delegation cannot delegate it.

- Recursive Delegation: This means that the delegation can be delegated any number of times.

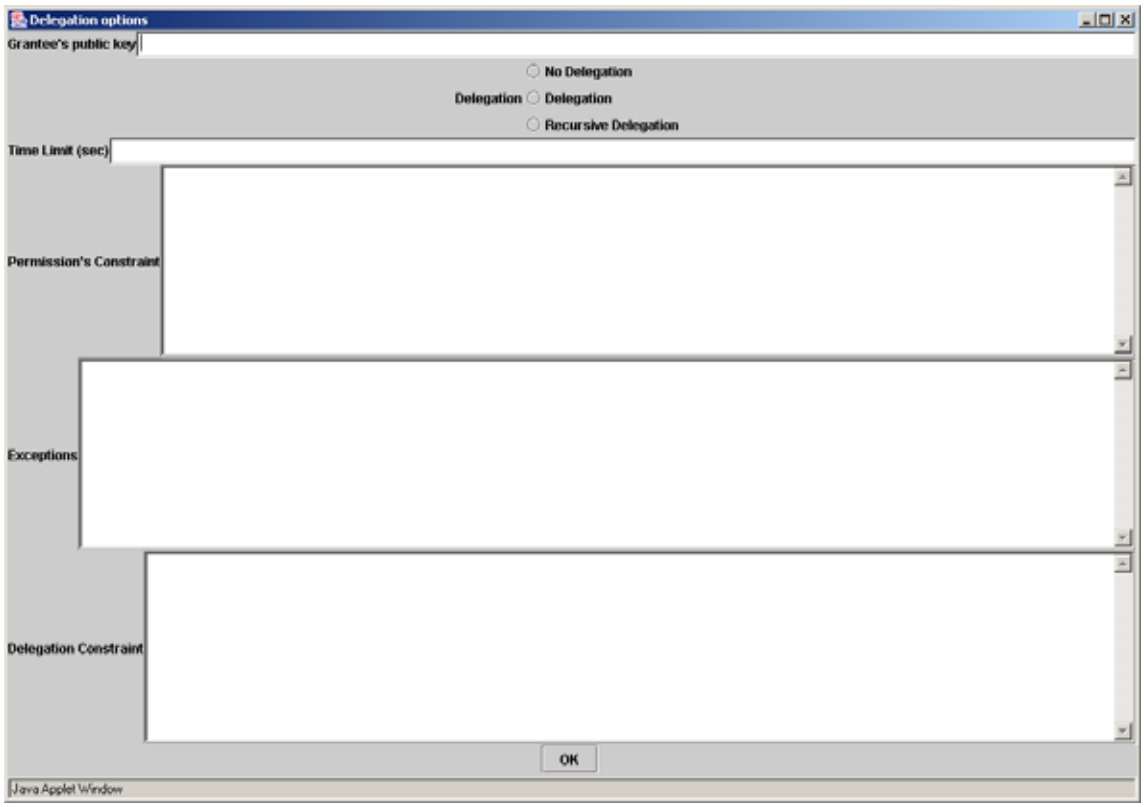

Figure 4.8: Delegating a Permission: Part 2

The client must also specify for how long a delegation can be valid. This is given in seconds. Once this time has expired, the delegation will no longer be valid for the user who has received this delegation. The client can also specify new exceptions and constraints to add to those already put in place by the administrator for the actual permission and for the delegation permission. The client cannot delete exceptions or constraints already put in place by the administrator or by the users who have already delegated this permission but it can add some new exceptions and constraints here. The users receiving this delegation will receive the same permissions that the original user has but with the added constraints and exceptions imposed by the client.

Once the client is finished, he or she presses the OK button. Again, XMLPolicy is used here to verify the permissions. In Figure 4.7, the system only displayed the permissions that could be delegated, but the system must still verify that the permission can be delegated. It verifies the permissions and checks that the permission can be

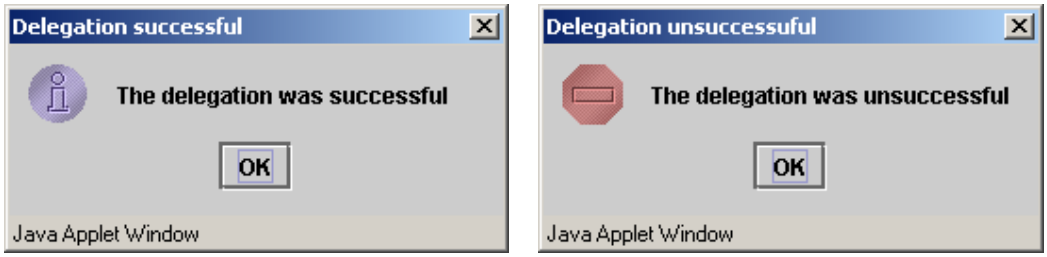

Figure 4.9: Delegation Results

delegated for the amount of time decided by the client. It also verifies that the type of delegation chosen by the client is valid.

If the system finds that the permission can be delegated using the options given by the client, then the delegation is successful and the user who is to receive this delegation now has that permission and can perform the operations but restricted by the new exceptions and constraints specified by the client. Otherwise, the delegation is unsuccessful. Figure 4.9 shows the possible results of the delegation requests.

4.5.3 Conclusion on the Calculator Application

The calculator application is a very simple application designed to show the different capabilities of our system. We used it with three participants. Alice (the server), Bob, and Charlie (the clients). The lookup service is also a participant on the network but it is only present because we use the Jini network technology in the example. Alice offers a calculator that can perform the basic arithmetic operations. It has a security policy that gives to Bob and Charlie some permissions.

Bob and Charlie connect to the lookup service to find Alice. They then connect to Alice using the Alice's proxy and start the authentication mechanisms. If authentication is successful, requests for some arithmetic operations can be made. When a request is made, Alice uses our system to find out if the permission to perform this operation has been given or not. The result is only shown to the client if permission has been given in the security policy.

Bob and Charlie can also perform some delegation. They ask Alice which permissions they can delegate and then select one of those permissions to delegate it. The clients are required to answer a series of questions about how this delegation will work on the user who receives this delegation. These answers are sent to Alice who verifies

that they are valid answers in the policy file. Our system checks for example that the time limit entered by the client is valid compared to the time limit entered in the policy file by the administrator. If the delegation is considered valid then the permission is delegated and the receiver can now perform the operation of the permission. Otherwise it is denied.

4.6 Conclusion

This chapter described the new Java policy provider that we created to improve the security of Java, Jini, and distributed systems in general. We begin by studying the security properties that were not present in Java but that we could implement in our security provider. Some of those security properties come from the Ponder Specification Language. We also base some of our ideas on the Java Secure Execution Framework and from SDSI / SPKI.

While the original Java policy provider offered support on positive authorizations, our policy provider improves this by adding support for negative authorizations, exceptions, constraints, and delegations. All these security properties are explained in this chapter. Authentication is another security property that we enforced and we explain how we implemented it in this chapter.

This chapter also explains our design decision of using a local name space instead of a global name space. A local name space offers more flexibility by letting participants of our network individually choose their names. Participants of our system are more independent from the other participants of the system than they are in other types of networks. We can still uniquely identify clients and services on our networks using their public key.

Next, this chapter explains how the authentication security property is enforced in our system using an SSL JAAS login module that we created. We also look at how all the security properties are enforced in our security provider that we called XMLPolicy. We explain the architecture of the security provider followed by a description of the syntax that we created for the policy files used in our system. The original policy files used in Java are described in Chapter 3. Our policy files are based on that but are written in XML. XML is the ideal language to write this type of structured document because the syntax of the files can easily be verified and reading and writing the file is very straightforward. Appendix A shows the entire structure of those files.

Finally, we looked at the example application that we created to demonstrate the capabilities of our system. It is a simple application called the Secure Calculator Application and it shows the different type of operations and verifications that our system can do.

Chapter 5

Conclusion

The goal of this research was to make a distributed system that runs using the Java programming language more secure. The first step we took to reach this goal is detailed in Chapter 2. It was to find out what a secure distributed system must look like. These are the basic security properties that any distributed system calling itself secure must support. We decided that our secure distributed system must at least support the authentication, authorization, confidentiality, and data integrity security properties. We also looked at different models of access control mechanisms that could help us implement access control mechanisms. We studied the access control matrix, the Bell-LaPadula Model and the Chinese Wall Model.

Next, we looked for which distributed system we could use in our research. We had two main criteria to consider while studying the possible distributed systems. Firstly, we are implementing a distributed system that uses the Java programming language. This is because it is a language that was built with security in mind so it would be simpler to make a distributed system using a secure language. Java is also a language that offers interesting features such as code mobility that makes a distributed system more interesting. Secondly, our distributed system offers a traditional type of network. This is a client-server network, where clients create requests and send them to a server, which executes the request and returns the result to the client. This means that we needed a distributed system that was of the type client-server.

Using those criteria, we studied four different distributed architectures. The first one is Jini, a network technology created by Sun Microsystems. We also studied CORBA, which was created by the Object Management Group. Next we studied Microsoft .NET, which is similar to Java but created by Microsoft. We also studied JESSICA2, a research project at the Department of Computer Science and Information Systems at

the University of Hong Kong.

After carefully studying those systems, we decided to use the Jini networking technology. This distributed system is written in Java so it inherits its security properties. Jini is a client-server type of network so both criteria are satisfied. Jini also has features such as leasing and a lookup service, which are not found in the other distributed systems.

CORBA was interesting but lacked some of the features of Jini, while offering other features that we do not need such as support for multiple programming languages. Microsoft .NET was not used because it does not support Java. JESSICA2 is not a client-server type of network so it could not be used.

Once we knew which distributed system we would use in our system, we looked at different tools that exist to enforce security properties. We looked at encryption, message digests, digital signatures, and the Java Authentication and Authorization Service (JAAS). We found interesting features in each tool that we would later implement in our system. We also looked at authentication protocols to find out how our system would be able to perform authentication, one of our basic security properties. The Kerberos and SSL protocols were studied. We also studied specification languages to look for ideas on how our network could be more secure. In SDSI / SPKI we discovered the local name space that we implemented in our system and in the Ponder Specification Language we found more specific security properties that we could implement in our system.

In Chapter 3, we discuss the current Java security model. This is the standard way for the language to enforce its security properties. Administrators or owners of systems create security policy files that tell the system how it must protect its resources so we start by looking at how those policy files are created and which types of information are found in them.

After the policy files are written, the administrator or owner of the system can start the application. Java, before starting the application, reads the policy file and stores it in memory. It also loads a security manager. When a client attempts to execute a sensitive operation, the application calls the security manager to make sure that this client has permission to execute this operation. The security manager calls the access controller, which uses the security policy stored in memory from the policy files to find out if this client is permitted to perform this action. This is detailed in Sections 3.3 and 3.4.

Chapter 4 details the system that we created to enforce our security properties. This is the XMLPolicy provider. We first look at the security properties that we want to enforce in our system. We already looked at the four basic security properties, but we can do even better by making those properties less general. Negative authorizations permit the administrator to create more precise security policies by letting him or her create policies that tell the system not only what is permitted to do but also what is not permitted to do. We can make authorizations more precise also using exceptions and constraints, which specify when a permission should be ignored or not at run time rather than when the application is started. The delegation security property is also interesting because it lets ordinary users give to other users some of their permissions when the security policy permits it.

We also explain how we implemented authentication. After studying some security protocols, we decided to use SSL for authentication in our system. SSL is well adapted to the local name space that we decided to implement. We use SSL to enforce bidirectional authentication using a JAAS login module that we created. We look at why we use a local name space. It is because we needed our clients and services of our networks to be more independent from each other. When choosing its name, a client for example does not need to find out if its name has already been used on the network.

The architecture of our new security provider (XMLPolicy) is described in detail. This is our main contribution. To use our security provider, we created a new syntax for the security policy files. The new policy files are written in XML, which is an ideal language for this type of structured document. The new system can enforce the new security properties that are described earlier. Finally, we look at an example application that we developed to demonstrate the capabilities of our system. This is the secure calculator application.

5.1 Contributions

Our main contribution is the creation of a new Java policy provider to replace the standard Java policy provider. Our policy provider is called XMLPolicy and supports a series of security properties. They are:

- Positive authorizations
- Negative authorizations
- Exceptions

- Constraints
- Delegations

Using Java and some Java tools, we also support the following security properties:

- Authentication
- Confidentiality
- Data Integrity

Together, those security properties enable developers to create more secure networks than they could create using the standard Java policy provider.

To create this system, we developed a new syntax for policy files. This syntax is written in XML and is detailed in Annex A. Using XML, it is simple to understand how to create the security files. It is also simple to create a method to verify the files to make sure that they have the correct syntax before loading them in memory.

Another contribution is the creation of a new JAAS login module that implements authentication using the SSL protocol.

Our ultimate goal was to improve the security of a distributed system. We chose to make the Jini network technology more secure. The result is a policy provider and a JAAS login module that is used to make Jini more secure but it is not limited to that. Our system can be adapted to make other types of distributed systems that use Java more secure as well as applications not running on a network.

5.2 Future Work

In the future, more security properties could be added to our system. For example, Ponder offers refrains, filters, and obligations. These could be added to our system to make it even more secure. Some of the currently implemented security properties could be improved as well. For example, our version of constraints only support the implementation of constraints based on the current system date and time. We could

improve this to implement support for constraints based on other factors such as the speed of the network or how busy the server is at this moment.

It may be interesting to improve the scalability of the system. Currently, the system works well with a few users as shown with the calculator application but it could be improved to support a large number of users working together on the system. We could also formalize the semantics of the security policies languages that we used such as Ponder. This would conclusively prove the correctness of the verification process.

We could implement security checks at a lower level in Java. Currently, we check that the users have permissions to execute operations during run time. We could also improve the verification of the code before it is executed. Java already does some verifications using the Java Bytecode Verifier. Improving this verifier could enable us to make application dependent verifications. Some security policies could be created to make those checks at this lower level.

Bibliography

[1] Anne Anderson. Java Access Control Mechanisms. Technical Report TR-2002-108, Sun Microsystems, Inc, March 2002.

[2] Apache Group. *Apache Ant Manual*, 2001.

[3] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specifications Second Edition*. Addison-Wesley., 2000.

[4] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley., 2002.

[5] Matthew Burnside, Dwaine Clarke, Srinivas Devadas, and Ronald Rivest. Distributed SPKI/SDSI-Based Security for Networks of Devices. Technical report, MIT Laboratory of Computer Science, December 2002.

[6] Michael Cierniak, Ali-Reza Adl-Tabatabai, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler, May 1998.

[7] James Conard, Patrick Dengler, Brian Francis, Jay Glynn, Burton Harvey, Billy Hollis, Rama Ramachandran, John Schenken, Scott Short, and Chris Ullman. *Introducing .NET*. Wrox Press Inc., 2000.

[8] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder specification language. *Lecture Notes in Computer Science*, 1995:18–39, 2001.

[9] EJBCA. *Enterprise Java Beans Certificate Authority*, 2004.

[10] Pasi Eronen, Johannes Lehtinen, Jukka Zitting, and Pekka Nikander. Extending Jini with Decentralized Trust Management. In *Short Paper Proceedings of the 3rd IEEE Conference on Open Architectures and Network Programming (OPENARCH 2000)*, pages 25–29, Tel Aviv, Israel, March 2000.

[11] Weijian Fang, Cho-Li Wang, and Francis Lau. Efficient Global Object Space Support for Distributed JVM on Cluster. In *International Conference on Parallel Processing*, Vancouver, British Columbia, Canada, August 2002.

[12] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, 1997.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[14] Manfred Hauswirth, Clemens Kerer, and Roman Kurmanowytsch. A secure execution framework for java. In *ACM Conference on Computer and Communications Security*, pages 43–52, November 2000.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Second edition, 1999.

[16] John C. Mitchell. Finite-State Analysis of Security Protocols. In *Computer Aided Verification*, pages 71–76, 1998.

[17] B. Clifford Neuman and Theodore Ts'o. Kerberos : An authentication service for computer networks. Technical Report ISI/RS-94-399, USC/ISI, 1994.

[18] Scott Oaks and Henry Wong. *Jini in a Nutshell*. O'Reilly & Associates, Inc., 2000.

[19] Object Management Group. CORBA Specification v1.2, December 1998.

[20] David S. Platt. *Introducing Microsoft .NET, Third Edition*. Microsoft Press, 2003.

[21] Ronald L. Rivest and Butler Lampson. SDSI – A Simple Distributed Security Infrastructure. Presented at CRYPTO'96 Rumpsession, October 1996.

[22] Jeremy Rosenberger. *Sams' Teach Yourself CORBA in 14 Days*. SAMS, 1998.

[23] Thomas Schoch, Oliver Krone, and Hannes Federrath. Making jini secure. In *Proc. 4th International Conference on Electronic Commerce Research*, pages 276–286, November 2001.

[24] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification and Validation*. Springer-Verlag, 2001.

[25] Scott Stark and Marc Fleury. *JBoss Administration and Development*. Que/Sams, 2002.

[26] Sun Microsystems Inc. Secure Computing with Java: Now and the Future, 1997.

[27] Sun Microsystems Inc. Java^{TM} Authentication and Authorization Service v1.0 Specification, 1999.

[28] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 Protocol, November 1996.

[29] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000.

[30] Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.

Appendix A

The New Java Policy File Syntax

This is a summary of what the XML policy file looks like in our system. Section 4.4.1 describes the policy file in detail. We give the entire syntax of the file here.

```
<policy>

<keyStore>
   <keyStoreLocation> </keyStoreLocation>
   <keyStoreType> </keyStoreType>
</keyStore>


<groups>
   <group>
      <groupName> </groupName>
      <members>
         <principals>
            <principal>
               <principalClass> </principalClass>
               <principalName> </principalName>
            </principal>
         </principals>
      </members>
   </group>
</groups>
```

```
<grant>
   <subject>
      <codeBase> </codeBase>
      <signedBy>
         <signer> </signer>
      </signedBy>
      <principals>
         <principal>
            <principalClass> </principalClass>
            <principalName> </principalName>
         </principal>
      </principals>
      <groupName> </groupName>
   </subject>


   <permissions>
      <permissionPos>
         <permissionClass> </permissionClass>
         <target> </target>
         <actions>
            <action> </action>
         </actions>
         <constraint> </constraint>


         <exception>
            <permissionClass> </permissionClass>
            <target> </target>
            <actions>
               <action> </action>
            </actions>
            <constraint> </constraint>
         </exception>
         <delegation>
            <deleg> </deleg>
            <delegDelegations> </delegDelegations>
            <timeLimit> </timeLimit>
            <constraint> </constraint>
         </delegation>
```

```
        </permissionPos>


        <permissionNeg>
           <permissionClass> </permissionClass>
           <target> </target>
           <actions>
              <action> </action>
           </actions>
           <constraint> </constraint>


           <exception>
              <permissionClass> </permissionClass>
              <target> </target>
              <actions>
                 <action> </action>
              </actions>
              <constraint> </constraint>
           </exception>
        </permissionNeg>


     </permissions>


  </grant>


  </policy>
```