Security Mechanisms in Distributed Component Models

Sicherheitsmechanismen in verteilten Komponentenmodellen

Studienarbeit im Fach Informatik

vorgelegt von

Fabius Klemm

geb. am 20.04.1977 in Mainz

Angefertigt am

Institut für Informatik
Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	Prof. Dr. F. Hofmann Dr.-Ing. J. Kleinöder Dipl.-Inf. B. Schnitzer
Beginn der Arbeit:	01.01.2001
Abgabe der Arbeit:	30.04.2001

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 30.04.2001 _____

Abstract

Component models allow programmers to reuse pre-developed pieces of application code, so-called components. In distributed component models, applications are built from components that reside on different computers in a network. Usually not every network user is allowed to access all distributed applications in the network. Distributed component models therefore provide security mechanisms and policies to enforce, for example, access control or message encryption. Otherwise, an application developer would be forced to implement security mechanisms in every component.

In this work, I examine and compare security mechanisms in distributed component models. First, I describe general security mechanisms necessary to secure distributed applications. In the following chapters, I examine the security models in EJB, Jini, COM, and CORBA. I conclude the work with a comparison of the different security models.

Kurzfassung

Komponentenmodelle ermöglichen Programmierern die Wiederverwendung von vorgefertigtem Anwendungscode, sogenannten Komponenten. In verteilten Komponentenmodellen können Anwendungen erstellt werden, deren einzelne Komponenten auf unterschiedlichen Computern in einem Netzwerk installiert sind. Üblicherweise darf nicht jeder Benutzer auf alle in einem Netzwerk installierten verteilten Anwendungen zugreifen. Verteilte Komponentenmodelle bieten deshalb Sicherheitsmechanismen, wie zum Beispiel Zugriffskontrolle oder Nachrichtenverschlüsselung. Anderenfalls wären Anwendungsentwickler gezwungen, für jede einzelne Komponente Sicherheitsmechanismen zu implementieren.

In dieser Studienarbeit untersuche und vergleiche ich Sicherheitsmechanismen in verteilten Komponentenmodellen. Zuerst beschreibe ich allgemeine Sicherheitsverfahren, die für die Absicherung von verteilten Anwendungen benötigt werden. In den darauf folgenden Kapiteln untersuche ich die Sicherheitsmodelle von EJB, Jini, COM und CORBA. Abschließend gebe ich einen zusammenfassenden Vergleich über die verschiedenen Sicherheitsmodelle.

Table of Contents

1 Introduction

Software components are “binary units of independent production” [Szyp98], which encapsulate their implementation. They interact with their environment through an interface defined by the *component model*, which allows programmers to independently develop software components for the same model. When building new applications, application developers can combine mature components from independent vendors, as well as self-programmed components. Altogether, component models allow programmers to develop application software in less time.

In *distributed component models*, application assemblers can build applications from components that execute on different computers in a network, thereby using the advantages of distributed computing: Instead of running an application on a high-end mainframe computer, it is possible to distribute the application to a group of workstation at a fraction of the cost. Other advantages are increased fault tolerance through replication of components and better extensibility.

On the other hand, new security problems arise in a network environment because every computer must be open for remote access. Applications running in the network should be accessible, but only by authorized users.

Distributed component models therefore provide security support for application programmers. Application programmers can use security mechanisms form the component model, instead of implementing security for every application, which is a very arduous and error-prone task.

In this work, I examine how distributed component models solve common security problems. In chapter 2, I begin with an overview of security mechanisms that are required to secure distributed applications, such as secure communication, identification and authentication, authorization, client protection, security auditing, and non-repudiation. As examples for secure communication and client protection, I give a short overview of the *Secure Sockets Layer (SSL)* and the Java 2 Security Model. In chapter 3, I examine security mechanisms in four component models, starting with *Enterprise JavaBeans (EJB)* and *Jini*, both based on the Java programming language, followed by the *Component Object Model (COM)*, and the *Common Object Request Broker Architecture (CORBA)*. As Jini provides no security additional to the Java security, I outline possible security extensions. In chapter 4, I give an overall comparison of the security models of EJB, COM, and CORBA. I complete this work with a conclusion in chapter 5.

2 Overview of Security Mechanisms

A security system prevents unwanted disclosure, modification, or destruction of information in a distributed system. It should provide the following security mechanisms: secure communication, identification and authentication, authorization, client protection, security auditing, and non-repudiation. I discuss these mechanisms in the following sections.

2.1 Secure Communication

Interaction between two applications is often over insecure lower layer communications. Secure communication requires protection of the content of a message, achieved with *encryption*, and protection of the *integrity* of a message. In the next two sections, I give an overview of cryptographic algorithms used for encryption and integrity protection.

2.1.1 Encryption

Encryption mechanisms ensure that communication over an open network is kept private by scrambling the content of a message. The original content is called *plaintext*, whereas the encrypted result is known as *ciphertext*. There are symmetric and asymmetric encryption algorithms.

Symmetric Encryption Algorithms

When symmetric key cryptography is used, sender and receiver share the same secret key, which is used to encrypt and to decrypt messages. A popular symmetric key mechanism is the *Data Encryption Standard (DES).* It was published in 1977 (updated in 1993) by the U.S. *National Bureau of Standards (NBS),* now renamed to *National Institute of Standards and Technology (NIST).* DES uses a 56 bits long key to encode 64 bits long plaintext chunks. If DES is considered too insecure, it is possible to apply DES multiple times with different keys, taking the output from one encryption step as the input for the next. For example, *triple-DES (3DES)* applies DES three times.

One difficulty of symmetric encryption is that the communicating parties need to share a common secret key. This problem is solved by asymmetric encryption algorithms as described next. [KuRo00]

Asymmetric Encryption Algorithms

Asymmetric encryption allows communicating securely without having a shared secret key in advance. The first algorithms were introduced by Diffie and Hellman in 1976 [KuRo00]. Asymmetric encryption algorithms use two keys: a *public key,* which is available to everyone and a secret *private key*. The sender uses the public key of the receiver to encrypt a message. Then, only the receiver has the suitable private key to decrypt the message. This means applying the receiver's public key, pub_R, then the receiver's private key, $priv_R$, to a message *m* gives back *m*: $priv_R(pub_R(m)) = m$

A popular asymmetric key algorithm is the *RSA* algorithm, which is named after its founders, Ron *Rivest,* Adi *Shamir,* and Leonard *Adleman.* Asymmetric encryption algorithms need much more computing resources than symmetric encryption algorithms. Asymmetric mechanisms are therefore often used to secure the exchange of a symmetric session key, which is then used for ongoing symmetric key encryption. [KuRo00]

One problem with asymmetric encryption algorithms is that of obtaining someone's true public key. This problem is solved using a *trusted intermediary,* also called *Certification Authority (CA).* A CA signs[1] *certificates* with its private key. A certificate is a statement, which confirms that the public key of an entity, such as a person or a network entity, has some particular value. The public key of the CA, which is necessary to verify a certificate, is known by every communication party (for example, it could be distributed with standard software packages). [KuRo00]

2.1.2 Integrity Protection

Integrity protection is the second requirement for secure communication. Sender and receiver want to ensure that a message is not modified during the transmission without detection. The sender assures integrity by digitally signing the message. The client can then verify that the message was not altered. *Digital signatures* can be created with asymmetric key encryption algorithms. The sender uses its private key $priv_S$ to encrypt a message *m*, thus creating a digital signature, $pirv_S(m)$, of *m*. The receiver, which gets *m* together with $priv_S(m)$ uses the sender's public key, pub_S, to decrypt the digital signature and compares the result with *m:* $pub_S(priv_S(m)) = m$?

Asymmetric encryption is computationally expensive. For this reason, instead of the whole message, the sender signs only a much smaller *message digest* of *m*. A message digest algo-

[1] Digital signatures are described in the next section.

rithm is a one-way hash function that takes *m* as input data and generates a fixed-size output, called a digest, hash, or digital fingerprint, *h(m).* The message digest algorithm must assure that given a message digest, it is computationally almost infeasible to find another message that will generate the same digest.

To check the integrity of a message, the receiver applies the sender's public key to the digital signature to recover the message digest. The receiver then computes the digest of the plaintext message and compares it with the decrypted digest: $h(m) = pub_S(priv_S(h(m)))$?

There are two major message digest algorithms in use today: the *MD5* by Ron Rivest, which produces a 128-bit message digest and the *Secure Hash Algorithm (SHA-1)*, which produces a 160-bit message digest. [KuRo00]

Another requirement for integrity protection is to assure that each message is *unique*. Uniqueness prevents that a message is captured by an intruder and then reused later maliciously. The sender therefore appends a timestamp or a sequence number to each message before digitally signing it. The receiver can thus detect intercepted and reused messages by a wrong timestamp or sequence number. [KuRo00]

2.2 Identification and Authentication

The process of proving the identity of someone else is called authentication. Bidirectional authentication is referred to as *mutual authentication*. Every user in a secure system is mapped onto an *identifier*, also called security *principal*. Users prove their identity with *credentials*, which can be in the form of a password, a swipe card, a fingerprint, or a certificate. [Alla00]

Communicating parties can use a *challenge-response protocol* to perform authentication. For example, when a client wants to authenticate a server, the server first sends its certificate to the client. As described in section 2.1.1, the certificate contains the server's public key. The client then generates a *nonce*, which is random number used only once, encrypts it with the server's public key, and sends it to the server. The server decrypts the nonce with its private key and sends it back to the client, which can thus be sure that it is communicating with the right server. Instead of a nonce, the client can also choose a random session key, which can be used for ongoing secure communication with a symmetric key algorithm. Figure 2.2-1 shows the steps in a challenge-response protocol. [KuRo00]

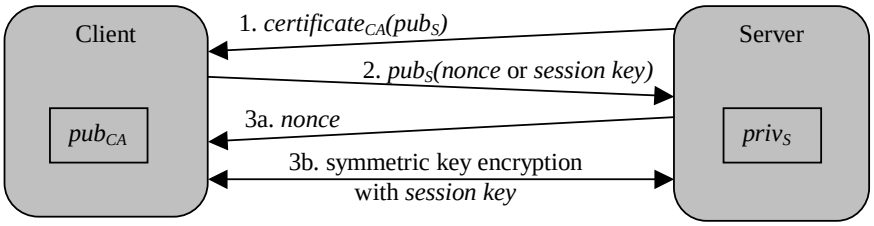

Figure 2.2-1 A client authenticates a server with a challenge-response protocol

2.3 Authorization

Authorization, which is also called *access control*, is required to protect network resources, such as files or applications against illicit access. Authorization is done using the *identity*, *role*, or *group* of a principal and the *control attributes* of the target component. Control attributes specify which principals can access which components, usually in the form of *Access Control Lists (ACLs*). [Alla00, OMG01a]

2.4 Client Protection

Client protection consist of two parts: A client should be able to control which of its rights are delegated to other components and which of its machine resources are open for component access.

2.4.1 Delegation

Clients call components to perform operations. A component often does not completely perform all operations itself and therefore calls further components. These calls can be under the client's identity or under the component's identity. The client should be able to control which of its rights are delegated and where and how long these rights can be used. When a component makes calls under the client's identity, it *impersonates* the client.

2.4.2 Client Machine Protection

When a client makes a call, the component either executes on the server or it is downloaded and runs on the client's machine. In the latter case, the client should have the possibility to control which system resources such as the file system or network downloaded components are allowed to access. There are two common ways to perform client machine protection: A component can run under a specific *user account* of the operating system and is thus restricted by the security policy settings of the account. The Java 2 Security Model[2] performs

[2] Chapter 2.7.2 gives an overview of the Java 2 Security Model.

a different approach. Java applications execute in the *Java Virtual Machine (JVM)* restricted by the Security Manager and Access Controller.

2.5 Security Auditing

Security auditing is used to record security related events and sensitive operations in a distributed system such as success and failure of authentication and object invocation. Security auditing is also required to make users accountable for their actions. An auditing service must be able to identify a user correctly, even after a chain of calls through multiple components. [OMG01a]

2.6 Non-repudiation

Non-repudiation (NR) is also called *accountability* and is used to generate and check irrefutable evidence about a claimed event or action. As an example, a NR service can generate an *evidence of creation* of a message. When a sender attempts to falsely deny the creation of a specific message, the recipient can prove the creation of the message with an evidence from the NR service. Another common NR type is the *evidence of receipt* of a message to protect a sender from falsely denying recipients. [OMG01a]

2.7 Examples

In the following sections, I give two examples for security mechanisms used in praxis: With *SSL*, two parties can enforce secure communication, for example between a web browser and a web server. The second example is the *Java 2 Security Model*, which implements client machine protection.

2.7.1 SSL

The *Secure Sockets Layer (SSL)* protocol was originally developed by Netscape and is the basis of the *Transport Layer Security (TLS)* protocol from the *Internet Engineering Task Force (IETF).* The SSL protocol runs above the TCP/IP layer and below application layer protocols such as HTTP. It allows mutual authentication of the communicating parties, and encryption and integrity protection of the data sent over the connection. When an SSL connection is established, client and server perform a series of actions, called a *handshake*. During the handshake, client and server first exchange their SSL version numbers and information about available cryptographic algorithms. The SSL protocol supports a large number of cryptographic algorithms, such as DES, 3DES, RSA, MD5, or SHA-1. Client

and server then exchange their certificates to perform mutual authentication with a challenge-response protocol and to agree on a symmetric session key for ongoing encryption. [Ipla98]

2.7.2 Java 2 Security Model

The Java language is designed to make it easier for a programmer to write safe code. It is strongly typed, provides automatic memory management, garbage collection, and range checking on strings and arrays. The *Java Virtual Machine (JVM)* performs language type safety and range checks at run time. The *Class Loader*, which loads Java classes into the JVM, defines a local name space to assure that an untrusted Java program cannot access other Java programs running on the same machine. It uses the *Java Bytecode Verifier* to check the integrity of Java bytecode. Access to system resources is mediated by the JVM and checked by the *Security Manager* and *Access Controller*. [GoMu97]

The *Java 2 Security Model* allows to run applications as well as applets restricted by a security policy. Figure 2.7-1 gives an overview of the Java 2 Security Model. I describe some of its components in the following subsections.

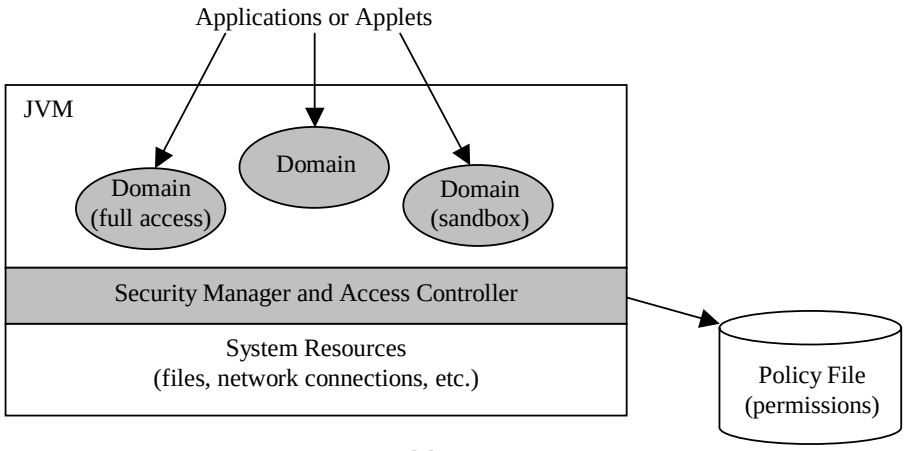

Figure 2.7-1 Java 2 Security Model

Permission Classes

Permission classes represent access to system resources. A permission consists of a *target* and one or more *actions*. For example, the `java.io.FilePermission` class represents access to a file or directory and is associated with read, write, execute, and delete actions. The following code produces a permission to read and delete all files in */tmp*:

```
FilePermission p = new FilePermission("/tmp/*", "read,delete");
```

It is not possible to deny an action to a specific target. [Gong98]

Policy Object and Policy File

A policy object represents a *system security policy*, which specifies permissions for code from different sources. The system security policy may be stored as an ASCII policy file, a serialized binary file of the policy object, or in a database. In the default implementation one or more ASCII policy files are used.

A policy file consists of a *keystore* specification entry and *grant entries*. A keystore is a database that stores private keys and their associated digital certificates. A grant entry grants a set of permissions to a specified CodeBase. Listing 2.7-1 shows a typical policy file. The first line specifies the location of the keystore. The following grant entry will grant write permission to the local */tmp* directory to any code originating from the CodeBase URL *http://www.uni-erlangen.de* signed by both *Lisa* and *Mary*. [Gong98, Li01]

```
keystore "/jdk1.2/mykeystore"
grant codeBase "http://www.uni-erlangen.de/*", signedBy "Lisa,Mary"{
   permission java.io.FilePermission "/tmp/*", "write";
};
```

Listing 2.7-1 A policy file

Protection Domains

Classes are grouped into individual protection domains. A protection domain is uniquely defined by a *CodeSource*, which consists of a CodeBase and a set of certificates. Thus, all classes that belong to the same protection domain are signed by the same keys and originate from the same URL. A class can be member of only *one* protection domain. Each protection domain is associated with a specific set of permissions that is granted to the classes of the domain.

There are two distinct categories of protection domains: *system domains* and *application domains*. All Java 2 SDK code belongs to a unique system domain whereas an applet or application runs in its appropriate application domain. [Gong98]

3 Security in Distributed Component Models

In the next four sections, I examine EJB, Jini, COM, and CORBA. Each section begins with a short overview of the component model, followed by the discussion of the security model.

3.1 EJB

3.1.1 Overview of EJB

Enterprise JavaBeans (EJB) is part of the *Java 2 Platform, Enterprise Edition (J2EE)* from Sun Microsystems. It "defines a model for the development and deployment of reusable Java server components" [Thom98, 1]. EJB components, which are also called *enterprise beans*, implement only the business logic. Middleware services, such as memory management, network access, transactions, or security are provided by the EJB server and the EJB container. Enterprise beans do not provide a *Graphical User Interface (GUI)*. [Sun01a; Sun01b]

3.1.1.1 Architecture of EJB Applications

EJB technology is based on the Java programming language. EJB applications consist of four parts: enterprise beans, EJB containers, EJB servers, and clients. Figure 3.1-1 gives an overview.

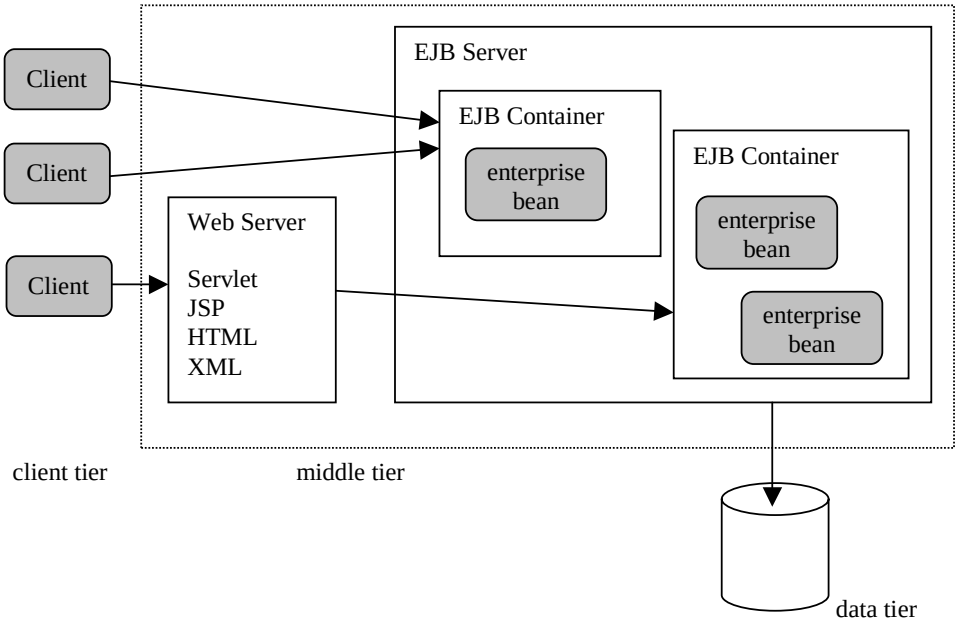

Figure 3.1-1 J2EE Environment [Thom99]

Enterprise Bean

Enterprise beans are server components and run in an EJB container. One or more EJB containers run in an EJB server. EJB servers and EJB containers provide all middleware services. An EJB application is built with multiple enterprise beans. There are standard contracts between the EJB container, the EJB server, and the enterprise bean to assure that an enterprise bean can be deployed in any EJB-compliant container. However, at the time being, enterprise beans are not fully portable because many vendors develop EJB servers and containers with proprietary extensions [GrTh00].

EJB technology supports *session beans* and *entity beans.* Session bean instances exist only for the duration of a single client/server session. A session usually spans multiple method calls, for example, to read or change data in a database. A session bean instance is terminated after the session. There are *'stateful'* session beans that retain the *conversational state* across method calls and transactions. The conversational state of a sessions bean instance consists of its field values and the field values of all instances reached by following object references [DeYa00, 61]. *Stateless* session beans do not maintain conversational state. Instances of *entity beans* represent persistent data and are maintained in a permanent data store, typically a database [Thom98]. Clients and enterprise beans communicate with the Java *Remote Method Invocation (RMI)* API.

Every enterprise bean has two interfaces:

- The *home interface* provides methods to create, find and destroy enterprise bean instances.
- The *remote interface* provides the business methods of an enterprise bean.

The Bean Provider, the Application Assembler, and the Deployer[3] use the *Deployment Descriptor* to specify information about an enterprise bean. This information concerns lifecycle, persistency, security, and other middleware services. The Deployment Descriptor is in *Extensible Markup Language (XML)* format. The EJB specification defines the XML *Document Type Definition (DTD)* for the Deployment Descriptor.

[3] The EJB roles are described in chapter 3.1.1.2.

EJB Container

An EJB container provides an execution environment for one or more enterprise beans. It reduces the complexity of developing applications by managing middleware services, such as life cycles, transactions, concurrent access, security, and persistence of the included enterprise beans. To provide these services, the EJB container intercepts every method call, that means clients cannot access enterprise beans directly. Clients use methods provided by the *home interface* and the *remote interface* to access enterprise beans. The EJB container uses the values specified in the Deployment Descriptor to perform the middleware services. Figure 3.1-2 depicts an EJB container. [Thom98, 15]

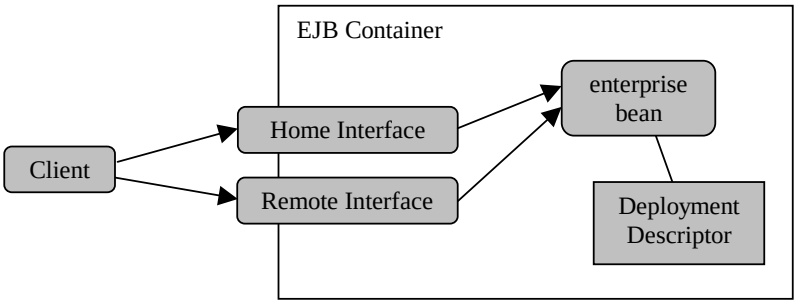

Figure 3.1-2 EJB Container

EJB Server

An EJB server may host one or more EJB containers. It provides services for the management of system resources, such as threads, memory, or database and network access. EJB server and EJB container are provided by the same vendor.

Client

A client contains only presentation logic. It can use *Java Naming and Directory Interface (JNDI)* to get the home interface of an enterprise bean. All interceptions from the EJB container are transparent to the client. Possible clients are Java applets, Java applications, other enterprise beans, or CORBA applications.

3.1.1.2 EJB Roles

The EJB specification defines seven roles that participate in the development and deployment process of an EJB application [DeYa00]:

- **Enterprise Bean Provider:** Develops enterprise beans. The Enterprise Bean Provider uses the infrastructure provided by the EJB container and the EJB server and can therefore concentrate on the business logic.
- **Application Assembler:** Assembles multiple enterprise beans to a complete EJB application. The Application Assembler can be the same party as the Enterprise Bean Provider.
- **Deployer:** Installs and configures enterprise beans and EJB applications in the operational environment. The Deployer usually does not have to know every detail of an EJB application.
- **EJB Server Provider:** Develops EJB servers.
- **EJB Container Provider:** Develops EJB container. “The current EJB architecture assumes that the EJB Server Provider and the EJB Container Provider roles are the same vendor” [DeYa00, 35].
- **Persistence Manager Provider:** Manages the persistent state of the entity beans installed in an EJB container.
- **System Administrator:** Configures and administers the running EJB application server and infrastructure network.

3.1.1.3 Packaging

The Bean Provider and the Application Assembler use the *ejb-jar file* format for the packaging of enterprise beans and assembled EJB applications. An ejb-jar file contains one or more enterprise beans, the Deployment Descriptor, and assembly information. [DeYa00, 485]

3.1.2 Security in EJB

The security examination in this section is based on the Enterprise JavaBeans™ Specification, Version 2.0, Proposed Final Draft [DeYa00].

3.1.2.1 Secure Communication

The Container Provider is responsible to provide secure communication mechanisms. The EJB specification does not define a secure communication protocol. Usually, the SSL protocol is used. The Deployer is responsible for configuring EJB containers to protect the communication between enterprise beans. “The Deployer should configure containers to reject call requests or responses with message content that should be protected but is not protected” [Kass00, 235].

3.1.2.2 Identification and Authentication

Identification and authentication is not addressed by the EJB specification. Application servers must authenticate users through proprietary means. One possibility is to “require a client application to provide a user-name parameter and a password parameter in the JNDI initial context” [Alla00, 1051]. The EJB container can also use the identity provided by an implementation of the SSL protocol. Another possibility is that the EJB server uses the *Java Authentication and Authorization Service (JAAS)* to authenticate the user’s identity. Once the EJB sever identifies and authenticates the client, it will map the identity onto a logical *security role* (see next section). [Alla00, 1051]

EJB containers can build trust relationships to other EJB containers installed in different EJB servers if the network is physically secure. The Deployer or System Administrator configure the trusted containers in *Trust Container Lists (TCL)*. [DeYa00, 400]

3.1.2.3 Authorization

The EJB specification defines what each EJB role has to do to enforce authorization.

Bean Provider

The Bean Provider should not hard-code any security policies and mechanisms, unless in less frequent situations in which access to *security context* information is absolutely necessary. In such situations the Bean Provider can use the following two methods provided by the `javax.ejb.EJBContext` interface to access security context information:

```
public interface javax.ejb.EJBContext {
      java.security.Principal getCallerPrincipal();
      boolean isCallerInRole(String roleName);
}
```

With `getCallerPrinicpal()`, the Bean Provider can obtain the name of the caller principal. The name may be used, for example, as a key to access information in a database. The `isCallerInRole(String roleName)` method tests whether the caller has been assigned to the *security role* `roleName.` Security roles are defined by the Application Assembler (see next subsection). The Bean Provider should declare and describe all the security role names used in the enterprise bean code in the `security-role-ref` elements in the Deployment Descriptor. [DeYa00]

Application Assembler

The Application Assembler has detailed knowledge of the EJB application and appropriate access control restrictions. He defines which groups of users are allowed to invoke which groups of methods. He provides this *security view* with the definition of *security roles* and *method permissions.*

Security Roles

The Application Assembler does not know the user names and groups in the target operational environment. He therefore defines *security roles* with the `security-role` elements in the Deployment Descriptor. Security roles are logical groups of users with the same access rights. Listing 3.1-1 shows an extract of the Deployment Descriptor with the definition of two security roles: *employee* and *payroll-department*. The Application Assembler also describes each security role with the `description` element. [DeYa00]

```
…
<assembly-descriptor>
      <security-role>
            <description>
                  This role includes the employees of the
                  enterprise who are allowed to access …
            </description>
            <role-name>employee</role-name>
      </security-role>

      <security-role>
            <description>
                  This role includes the employees of the
                  payroll department.
                  This role is allowed to view and update
                  the payroll entry for any employee.
            </description>
            <role-name>payroll-department</role-name>
      </security-role>
      …
</assembly-descriptor>
…
```

Listing 3.1-1 Definition of security roles in the Deployment Descriptor [DeYa00, 440]

The Application Assembler must also link all *security role references* declared by the Bean Provider to security roles. Listing 3.1-2 depicts how the security role reference *payroll*, defined and used in the enterprise bean code by the Bean Provider, is linked to the security role *payroll-department*, defined by the Application Assembler. [DeYa00]

```
…
<enterprise-beans>
      …
      <entity>
            <ejb-name>Payroll</ejb-name>
            <ejb-class>com.payroll.PayrollBean</ejb-class>
            …
            <security-role-ref>
                  <description>
                        This role should be assigned to …
                  </description>
                  <role-name>payroll</role-name>
                  <role-link>payroll-department</role-link>
            </security-role-ref>
            …
      </entity>
      …
</enterprise-beans>
…
```

Listing 3.1-2 Linking security role references to security roles in the Deployment Descriptor [DeYa00, 444]

```
…
<method-permission>
      <role-name>employee</role-name>
      <method>
            <ejb-name>EmployeeService</ejb-name>
            <method-name>*</method-name>
      </method>
      <description>
            …
      </description
</method-permission>

<method-permission>
      <role-name>employee</role-name>
      <role-name>payroll-department</role-name>
      <method>
            <ejb-name>Payroll</ejb-name>
            <method-name>findByPrimaryKey</method-name>
      </method>
      <method>
            <ejb-name>Payroll</ejb-name>
            <method-name>getEmployeeInfo</method-name>
      </method>
      …
</method-permission>
…
```

Listing 3.1-3 Assignment of method permissions to security roles in the Deployment Descriptor [DeYa00, 443]

Method Permissions

Method permissions specify the methods of the home and remote interface that each security role is allowed to invoke. Listing 3.1-3 on page 15 illustrates the definition of two method permissions. The first definition allows all users of the group *employee* to invoke all (*) methods from the enterprise bean *EmployeeService*. The second method permission allows all users of the groups *employee* and *payroll-department* to invoke the methods *findByPrimaryKey* and *getEmployeeInfo* from the enterprise bean *Payroll*.

Deployer

"The Deployer assigns principals and/or groups of principals (such as individual users or user groups) used for managing security in the operational environment to the security roles defined in the security-role elements of the Deployment Descriptor" [DeYa00, 446]. Figure 3.1-3 shows the assignment of the security roles defined in Listing 3.1-1.

There are deployment tools, which help the Deployer to read the *security view* of an application. He also assigns principals for the run-as identities[4] and mappings for resource manager access.

"This mapping of a user's actual security identity onto a logical security role is the key to understanding Enterprise JavaBeans security" [Alla00, 1052].

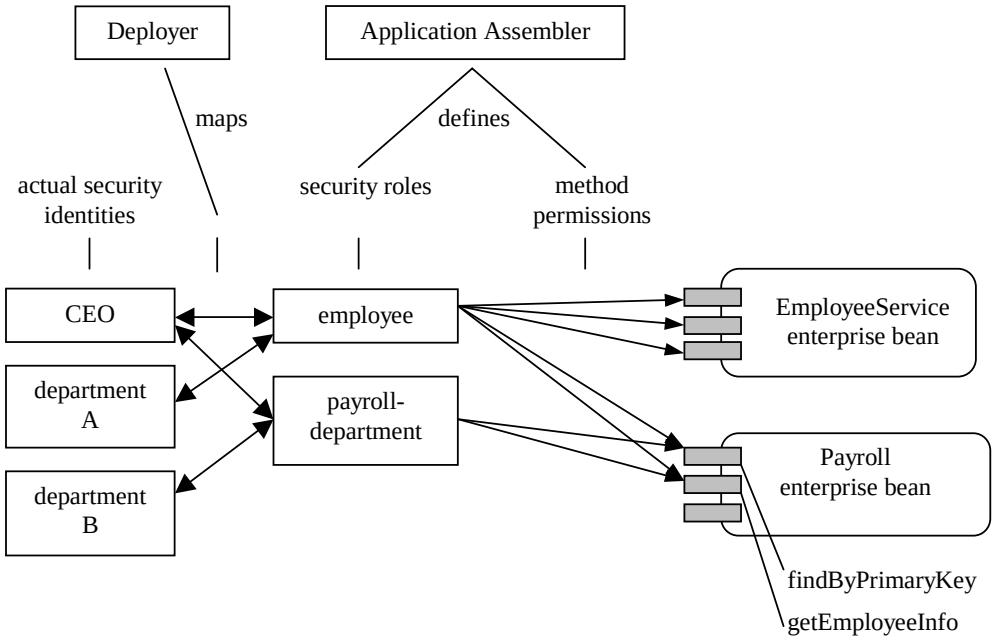

Figure 3.1-3 Mapping of actual security identities onto security roles

[4] See section 3.1.2.4 Client Protection.

Container Provider

The EJB container enforces authorization according to the specifications in the Deployment Descriptor and the mappings of the Deployer. If an enterprise bean access is illegal, the EJB container must throw a `java.rmi.RemoteException`.
The EJB Container must also make the identity and role information of callers available to enterprise beans.

3.1.2.4 Client Protection

Delegation

Delegation in EJB is solved with *security identities*. Enterprise bean method calls can run under two different security identities: The *caller's security identity*, or a *specific run-as identity*. If the caller's security identity is used, the caller principal is propagated from one enterprise bean to another. With run-as identity, a specific run-as principal is used on any calls that the enterprise bean makes. The Deployer specifies security identities in the Deployment Descriptor. [DeYa00, 447]

Client Machine Protection

EJB applications always run in an EJB container on the server and not on the client's machine. However, some clients download Java applets or applications to access EJB applications. In this case, the Java 2 Security Model restricts Java programs according to the client's security policy.
The server machine can also restrict enterprise beans with the Java 2 Security Model.

3.1.2.5 Security Auditing

Security auditing in EJB is optional. An EJB container may provide a security audit trail mechanism, which logs all `java.security.Exceptions` and all denials of access to EJB servers and EJB containers. The System Administrator is responsible for security auditing management. [DeYa00, 451]

3.1.2.6 Summary

EJB divides the responsibility of security between those who *develop* enterprise beans and EJB applications, and those who *deploy* EJB applications [Kass00, 238]. The Bean Provider and the Application Assembler should be relieved from details of security mechanisms. They specify their security requirements in the Deployment Descriptor at an abstract level external to EJB applications [Alla00, 1052]. The Deployer and the System Administrator then select suitable security mechanisms, which are implemented by the EJB Container and the EJB Server. An EJB application can thus be installed in environments with different security requirements and mechanisms. However, the EJB specification defines Deployment Descriptor entries only for authorization and delegation.

3.2 Jini

3.2.1 Overview of Jini

Jini, developed by Sun Microsystems, provides a simple infrastructure to federate clients and services in a network without installation or human intervention. Services, such as applications, devices, or storage can be spontaneously connected to a Jini network. A service automatically registers itself to one or more directories, so-called lookup services. Such lookup services manage registered service providers and act as brokers between clients and services. A client searches the lookup service to get a proxy object, which runs on the client's machine and is used to access the service [Sun99]. While EJB is designed to build a relative static distributed enterprise system, Jini can be used to connect services in a "plug and play" manner.

3.2.1.1 Architecture of Jini

Jini Services and Clients

A Jini service can be an application, a software component, or a hardware device. Every Jini service provides a service proxy, which runs in the client's JVM and does all the communication work between client and service. Jini does not dictate a specific communication protocol.

A client calls the methods provided by a proxy object, which can be a simple RMI stub for talking to its remote service. It is also possible that a proxy object performs parts of the service or the whole service by itself. In the latter case, the term "downloaded software component" would be more suitable than "proxy".

Discovery

Before a service can connect with other services it must first find a Jini *community*. Jini provides *discovery protocols* by which a service can find *lookup services.* Lookup services, which are the only Jini services that must be started explicitly by administrators, manage the services in one or more communities and act as brokers between clients and services. It is also possible that a community has multiple lookup services. Jini supports several discovery protocols [Edwa99]: Services that already know a particular lookup service can request it directly with the *unicast discovery protocol*. Lookup services are named in URL

syntax and listen on default port number 4160[5]. Here is an example: `jini://uni-erlangen.de:4160`

Another possibility is the *multicast request protocol*, which is used by a service to find all lookup services running in the network. When a new lookup service starts up, it can use the *multicast announcement protocol* to publish its presence. Lookup services answer to discovery requests with serialized proxy objects that implement the `ServiceRegistrar` interface.

Join

A Jini service that wants to make itself available to a Jini community performs a process called *joining.* The service first makes a discovery to find out about lookup services in a Jini community. Each lookup service answers with a `ServiceRegistrar` object, which provides a `register()` method. The service then calls the `register()` method to upload its proxy object as well as its service description to the lookup service.

Lookup

The `ServiceRegistrar` object can also be used to search the lookup service for registered services. A client uses the `lookup()` method, which returns the proxy objects of the requested services. A proxy object is then executed in the client's JVM and used to make calls to the service implementation. "This idea of downloadable service proxies is the key idea that gives Jini its ability to use services and devices without doing any explicit driver or software installation" [Edwa99, 70]. Figure 3.2-1 gives an overview of the Jini architecture and shows the steps of a client-service provider connection.

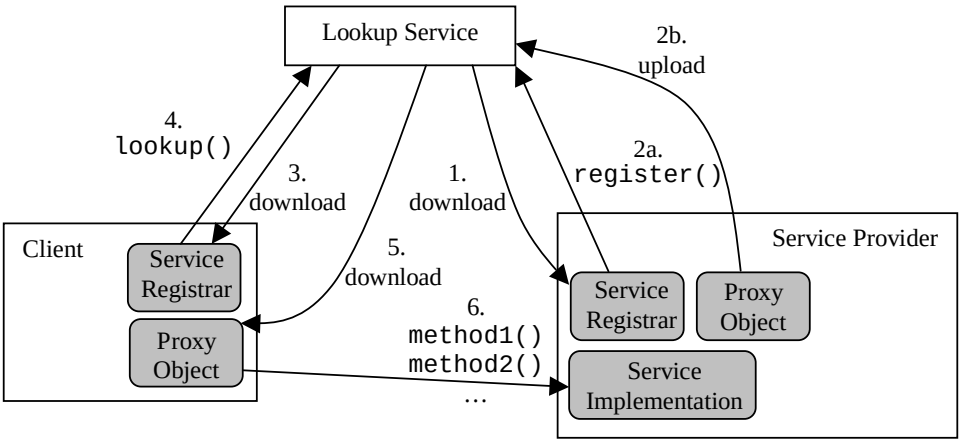

Figure 3.2-1 Overview of the Jini architecture

[5] $4160_{(10)} = CAFE_{(16)} - BABE_{(16)}$

3.2.2 Security in Jini

The Jini architecture does not provide any security mechanisms additional to the Java security. In this section, I therefore first describe security requirements and then outline security extensions for a Jini network.

3.2.2.1 Security Requirements

There are security requirements between proxy object and service implementation, client and proxy object, and client and service implementation.

Proxy Object-Service Implementation Interaction

Authentication: The proxy object must assure that it is talking to the right service implementation. The service implementation cannot authenticate the proxy object because it is impossible to conceal an encryption key inside the proxy object.
Secure Communication: Encryption and integrity protection is necessary for all messages with confidential content exchanged between proxy object and service implementation.

Client-Proxy Object Interaction

Proxy object integrity and authenticity: In a traditional client-server system, the client would authenticate the server. A Jini client however contains no code for communicating and cannot authenticate a service without the help of the proxy object. A client therefore must be able to prove integrity and authenticity of the proxy object to assure that it has downloaded the “genuine” proxy object of the service provider. The proxy object then authenticates its service implementation.
Client runtime environment protection: The client must have the possibility to control which local resources a proxy object can access. The Jini system already fulfils this security requirement by using the Java 2 Security Model.

Client-Service Implementation Interaction

Authentication: The service implementation requires authentication of the calling client to enforce access control. The client cannot give its private key to the proxy object, because the proxy object might misuse it. Client authentication must therefore go through the proxy object. Figure 3.2-2 on page 21 summarizes the security requirements in a Jini system.

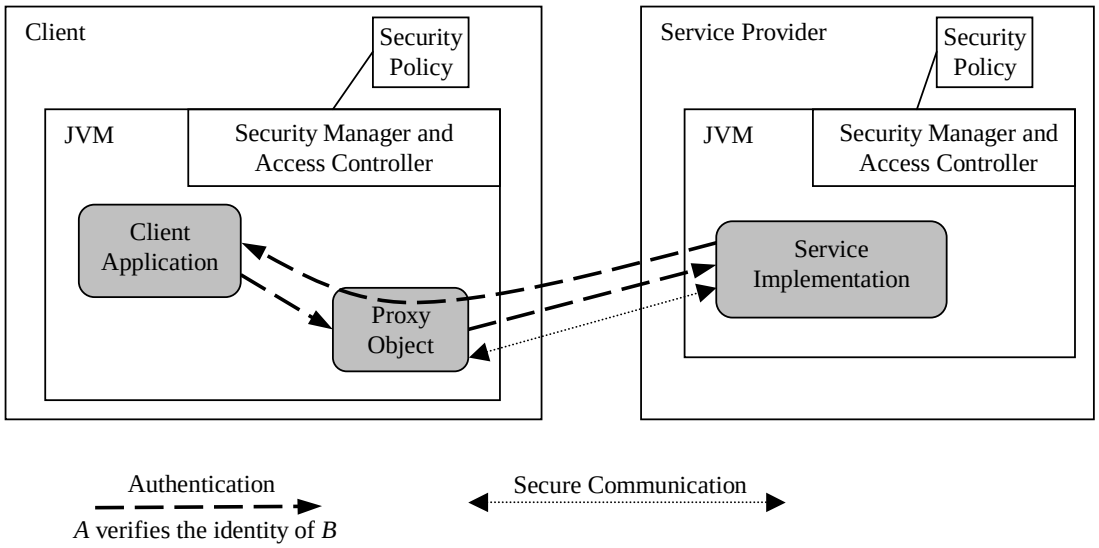

Figure 3.2-2 Security requirements in a Jini system

3.2.2.2 Jini Security Extensions

In this section, I outline possible Jini security extensions, that meet the requirements described in the last section. My extension are similar to those in [HaKe00] and [ErNi01].

Authentication

One possibility to perform authentication in a Jini network is with a *Certification Authority (CA)*. It is assumed that every participant in the Jini network knows the CA's public key.
Integrity and authenticity of a proxy object is guaranteed with a digital signature by its service provider. The client downloads a signed proxy object together with the CA certificate that contains the service provider's public key.
The proxy object contains the service implementation's public key and can thus authenticate the service implementation by using a challenge-response protocol.
The service implementation also uses a challenge-response protocol to authenticate the client and to negotiate a session key. As the client cannot give its private key to the proxy object, the proxy object must hand all messages of the challenge-response protocol to the client. After the challenge-response protocol, the client gives the negotiated session key to the proxy object for ongoing communication.

Authorization

The service implementation can enforce access control with *capabilities* or with ACLs. A capability is a signed statement that certifies certain access rights to a client. It can be

signed either by the service provider or by a trusted *Capability Manager (CM)* and is usually stored on the client machine. The client sends its capability to the service provider before making a call. Capabilities signed by a CM should be used when there are many clients with medium security demands, because such capabilities are only as trustworthy as the CM. Generally, capabilities should have an expiration date, otherwise it is difficult for a service provider to revoke access rights.

Another possibility is that a service provider manages ACLs. An ACL contains entries that either grant or deny access rights to a certain client, which can be identified by its public key. ACLs are more flexible than capabilities and the service provider does not have to rely on CMs or CAs. ACLs with many entries are difficult to manage. For service providers with a large number of clients it is therefore easier to use capabilities.

Secure Communication

After the authentication process, proxy object and service implementation share a secret session key, which is used for secure communication with a symmetric encryption algorithm. Proxy object and service implementation can also exchange further session keys of different length to realize multiple security levels.

3.2.2.3 Related Work

Hasselmeyer et al. [HaKe00] developed a secure lookup service, which enforces client authentication and secure communication. Certificates are managed by a CA. The lookup service supports special security groups, which grant access only to authorized clients. A CM administers access rights. There is also a group without any access control to maintain compatibility with “legacy” Jini applications.

3.2.2.4 Summary

Sun Microsystems developed the Jini technology without security mechanisms additional to the Java 2 Security Model. Generally, security extensions reduce the spontaneity of a Jini network. For example, a system administrator must first set authorization policies before a client can use a service.

Building trust in the proxy object can be easily achieved by signing the proxy object. Secure communication between client and service implementation can be realized with symmetric key encryption. More complex to achieve are extensions to enforce client authentication and authorization.

3.3 COM

3.3.1 Overview of COM

The *Component Object Model (COM)* from Microsoft is a standard for integration between binary software components. It defines means for developing components and specifies how these components and their clients communicate. COM specifies how components *interact*, not how they are *structured*. The internal structure depends on the programming languages and development environments used [WiKi94]. The COM-library provides API functions to facilitate the creation of components.

In its initial releases, COM could not be used to connect components residing on different computers. Since the release of Windows NT 4.0, COM also provides an infrastructure to support communication with components over a network. This *update* to COM is referred to as *Distributed COM* or *DCOM*. [Msdn99]

With the release of Windows 2000, Microsoft introduced COM+. COM+ combines enhancements to COM with the *Microsoft Transaction Server (MTS)*. It handles infrastructure services that COM developers have to program, such as thread allocation, object activation, load balancing, transactions, events, and security. [Sdk01]

3.3.1.1 Architecture of COM

Components

Microsoft does not define exactly what a component is [Szyp98, 194]. According to [Msdn99, 155], a component is a "collection of COM classes packaged into an executable unit, such as a DLL[6] or EXE". An instance of a COM class is called a *COM object.* A COM object is "some piece of compiled code that provides some service to the rest of the system" [WiKi94]. The internal implementation of a COM object is completely autonomous and therefore programming language independent. COM classes are identified with *Class Identifiers (CLSIDs)*. A CLSID is a G*lobally Unique Identifier (GUID)*, a 128-bit number guaranteed to be globally unique.

All COM objects combined in one component are grouped to the same *Application Identifier (AppID)*, which is also a GUID. The COM library provides the function `CoCreateInstance`[7] to create a new COM object.

[6] Dynamic Link Library
[7] All COM library function names start with *Co*.

COM Servers

A COM server contains the implementations of multiple COM classes. There are two types of COM servers: *in-process* and *out-of-process.* In-process COM servers are implemented as DLL. Out-of-process COM servers are implemented as EXE [Msdn99, Sdk01]. Each COM server implements a *factory object* for each class to create COM objects [Szyp98]. A COM server with two classes and factories is depicted in Figure 3.3-1.

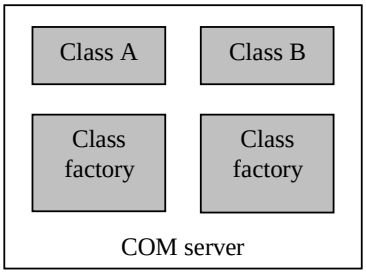

Figure 3.3-1 COM server with two classes, each with a factory [Szyp98, 205]

Interfaces

An interface is a "defined set of functions that are grouped together under one name" [EdEd98, 20]. It defines the name, parameter types, and return type for each function. An interface is a typed contract between COM object and client. A COM object implements an interface when it implements each member function. Every interface has an unique *Interface Identifier (IID)* to eliminate name conflicts.

Every COM object must support the interface `IUnknown`[8]. Clients use this interface to control the lifetime of the COM object and to query a COM object whether it supports a predefined interface. Clients always use an *interface pointer* to call functions of a COM object. [WiKi94]

An interface must not be changed. A changed interface must be published as a new interface with a new IID. As a COM object is able to support multiple interfaces it can support interfaces in different versions. Clients that are aware of the new interface can use it, whereas older interface versions are still available for older clients. Functions that are defined in two or more interfaces of a COM object can share the same internal implementation. The payroll object in Figure 3.3-2 on page 25 supports the old interface `IPayroll` and the new interface `IPayroll2`, as well as the mandatory `IUnkown` interface.

[8] Interface names begin with *I* by convention.

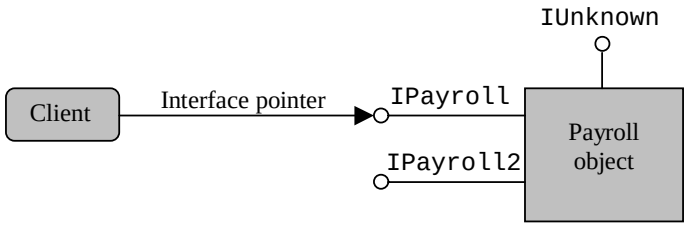

Figure 3.3-2 COM object with multiple interfaces

Cross-process Communication

COM allows clients to transparently communicate with other COM objects regardless of where those COM objects are running. *In-process* COM objects run in the address space of the client and allow very fast function calls. A client can access in-process COM objects directly through interface pointers. *Local* and *remote* COM objects are *out-of-process.* They run in their own memory space and cannot be accessed directly by a client. Instead, the client uses a local or remote object proxy, which generates *Remote Procedure Calls (RPCs).* Function calls to out-of-process COM objects are significantly slower than in-process calls because they require a process switch and the copying of parameters. *Local* COM objects run on the same machine as their clients whereas *remote* COM objects run on a separate machine connected via a network. Figure 3.3-3 shows a client calling functions from in-process, local, and remote COM objects.

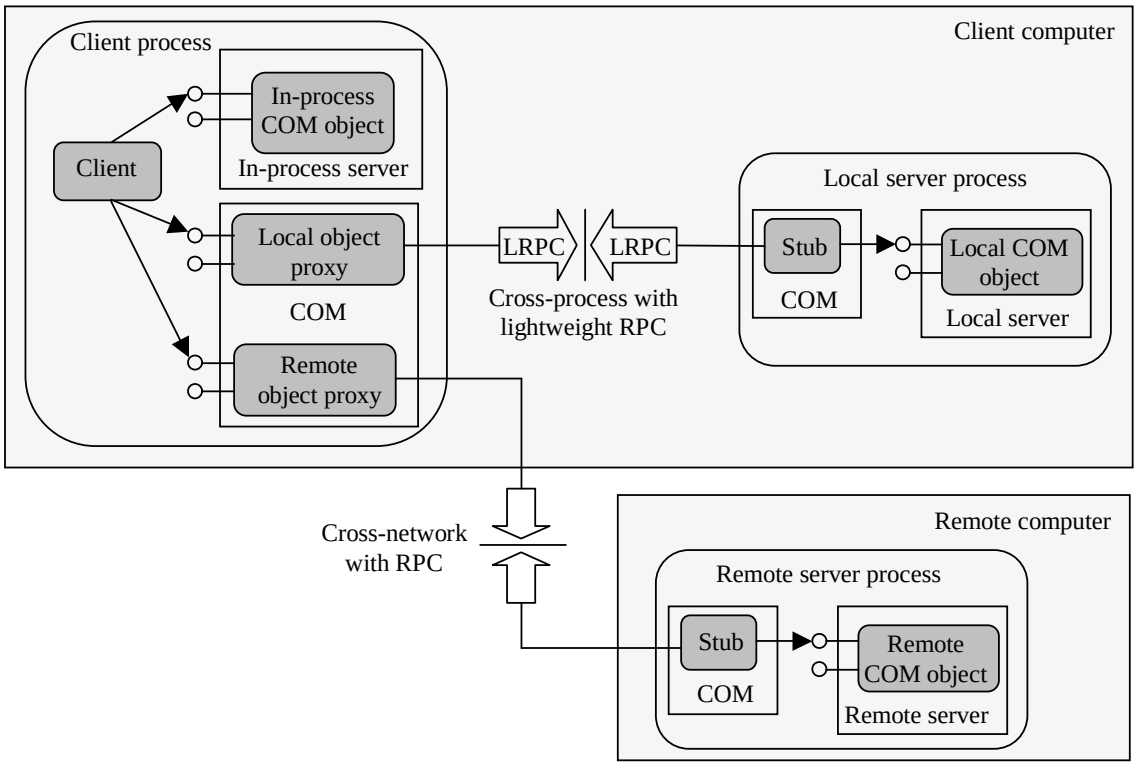

Figure 3.3-3 In-process, local, and remote COM objects [WiKi94]

3.3.2 Security in COM

In the following sections, I describe the COM security model. I examine all security mechanisms introduced in chapter 2, except non-repudiation. I also give a short overview of the COM+ security model.

3.3.2.1 Overview

The COM security model distinguishes between *declaratively* and *programmatically* configured security settings.

Declarative Security

Declarative security is configured externally and thus transparent to both the client and the component. It provides more flexibility at deployment time because the same component can be used with different security policies. Declarative security settings can therefore also be used for components that were developed without security concerns. The system administrator can define external security settings in the *registry* with the help of configuration utilities such as `dcomcnfg.exe`. COM will automatically enforce all necessary security checks according to the settings in the registry. There are *default* and *component* declarative security settings. COM applies default, or machine-wide security, to all components running on the local machine that do not override these default settings. Component security allows the system administrator to configure security specifically for each component. More fine-grained security settings for single COM objects or functions can be specified only programmatically. [Dule01, EdEd98, HoKi97]

Programmatic Security

Some features of the COM security model, for example different access control settings for each function in a COM object can be specified only programmatically. Programmatic security overrides declarative security. Usually, declarative security and programmatic security are combined [EdEd98]. COM provides several functions and interfaces for programmatic security:

COM servers and clients can call `CoInitializeSecurity` to initialize the security infrastructure on a per-process basis with their own values. For applications that do not call this function explicitly, COM will call it with default values from the registry. [Sdk01]

Clients can use the `IClientSecurity` interface to control the security settings for a particular connection to an out-of-process COM object. Every out-of-process COM object has a proxy manager, which implements this interface. [Sdk01]

When a client has invoked a function of a COM server, the COM server can use the `IServerSecurity` interface, which is implemented by the COM server stub. With

`CoGetCallContext` the COM server can get a pointer to `IServerSecurity`, which is valid for the duration of the client's function call. [Sdk01]

3.3.2.2 Identification and Authentication

Security Provider

COM uses a *security provider* to identify and authenticate a security principal. Different platforms support different security providers. Every security provider must implement the *Security Support Provider Interface (SSPI),* a standard API to insulate the developer from different security providers. Windows 2000 offers multiple security providers, such as the Windows *NT LAN Manager Security Support Provider (NTLM SSP)* or a SSP that implements the *Kerberos* network authentication service version 5 [Eddo99]. When COM asks the security provider to authenticate a user it receives an *access token.* The access token is used for ongoing authentication. It contains a *Security Identifier (SID),* which uniquely identifies the user, and the user groups to which the user belongs. [EdEd98, 400]

Authentication Levels

COM defines seven authentication levels, which are listed in Table 3.3-1. Levels 5 and 6 also specify packet encryption and integrity for secure communication.
Authentication levels are used by the system administrator to set default and component security in the registry and by applications that override the declarative security settings with `CoInitializeSecurity`.

Value	Authentication Level	Flag	Description
0	Default	`RPC_C_AUTHN_LEVEL_DEFAULT`	Currently maps to connect level authentication.
1	None	`RPC_C_AUTHN_LEVEL_NONE`	No authentication.
2	Connect	`RPC_C_AUTHN_LEVEL_CONNECT`	Authenticates the client only when the client first connects to the COM server.
3	Call	`RPC_C_AUTHN_LEVEL_CALL`	Authenticates the client at the beginning of each remote call.
4	Packet	`RPC_C_AUTHN_LEVEL_PKT`	Authenticates that all of the data received is from the expected client.
5	Packet integrity	`RPC_C_AUTHN_LEVEL_PKT_INTEGRITY`	Authenticates all of the data and verifies that it has not been modified when transferred between the client and the COM server.
6	Packet privacy	`RPC_C_AUTHN_LEVEL_PKT_PRIVACY`	Authenticates, verifies, and encrypts the arguments passed to every remote call.

Table 3.3-1 Authentication levels [EdEd98, 405]

The following registry entries are used for default and component authentication security (components are distinguished by their AppIDs):

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OLE\LegacyAuthenticationLevel =` *value*

`HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID\{`*AppID*`}\AuthenticationLevel =` *value*

A COM object can find out about the principal name and authentication level of the client with the `IServerSecurity::QueryBlanket` function. [Sdk01]

3.3.2.3 Authorization

COM distinguishes between launch security and access security.

Launch Security

Launch security, also called *activation security*, specifies which security principals are permitted to launch components. Launch security is enforced by the *Service Control Manager (SCM)*, which is responsible for locating COM class implementations and running them. The SCM uses ACLs stored in the registry. Launch security is always configured declaratively, because components themselves are not involved.

Default launch permission is used for components that do not provide their own ACL. The following registry entries set default and component launch permissions [Sdk01]:

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OLE\DefaultLaunchPermission =` *ACL*

`HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID\{`*AppID*`}\LaunchPermission =` *ACL*

Access Security

Access security specifies which security principals are allowed to call which COM components. ACLs for default and component access security are set under the following registry entries:

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OLE\DefaultAccessPermission =` *ACL*

`HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID\{`*AppID*`}\AccessPermission =` *ACL*

A COM server can manage more fine-grained access control to its COM objects with the `IAccessControl` interface provided by the COM library. This interface provides functions to set and revoke access rights for individual COM objects. [Sdk01]

A COM object can also use the `IServerSecurity::QueryBlanket` function to determine the security credentials of the calling client and then take special actions depending on the user identity. [EdEd98]

3.3.2.4 Client Protection

Impersonation Levels

COM allows COM objects to impersonate their callers. A caller can specify what actions the COM object is allowed to perform under its identity. COM defines four impersonation levels as listed in Table 3.3-2:

Value	Impersonation Level	Flag	Description
1	Anonymous	`RPC_C_IMP_LEVEL_ANONYMOUS`	The object is not allowed to obtain the identity of the caller.
2	Identify	`RPC_C_IMP_LEVEL_IDENTIFY`	The object can only detect the security identity of the caller, but can not impersonate the caller.
3	Impersonate	`RPC_C_IMP_LEVEL_IMPERSONATE`	The Object can impersonate the caller and perform local operations on the machine where the object is running. The object can not call other objects on behalf of the caller.
4	Delegate	`RPC_C_IMP_LEVEL_DELEGATE`	The object can impersonate the caller and call other objects using the security identity of the caller.

Table 3.3-2 Impersonation Levels [HoKi97]

The default impersonation level for all clients running on the system is specified under:
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OLE\LegacyImpersonationLevel =` *value*
A client can set a process-wide impersonation level with `CoInitializeSecurity`. A client can also use the `IClientSecurity::SetBlanket` function to specify the impersonation level for specific COM objects. [Sdk01]

Client Machine Protection

A component executes under a specific *security identity*, i.e. a user account. The security identity is associated with certain privileges, which define the access rights of a component. The security identity of a COM component is always configured via the registry:
`HKEY_LOCAL_MACHINE\SOFTWARE\Classes\APPID\{`*AppID*`}\RunAs =` *value*
The *value* specifies a user name and must be in the form *username* or *domain**username*. [Sdk01]

3.3.2.5 Security Auditing

The Win32 API provides functions for auditing security-related events[9]. However, using Win32 API security functions ties a component to the Windows NT platform [EdEd98,

[9] See [Msdn01] for further details.

413]. As far as I know, the COM security model provides no interfaces or functions to realized platform-independent security auditing.

3.3.2.6 Overview of COM+ Security

In COM+, the *catalog* contains the declarative security settings. These settings can be changed with the *Component Services administration tool* [Eddo99].

Access control is based on security roles and privileges. COM+ offers two methods for programmatic access control:

- `IObjectContext::IsSecurityEnabled:` Checks whether role-based security is enabled.
- `IObjectContext::IsCallerInRole:` Checks whether a user is assigned to a specific role.

COM+ uses the *authenticode technology* to sign code, which makes is possible to identify the code provider and to assure that the code has not been changed. COM+ also uses the concept of *zones*, which allows classifying code sources in different zones. For each zone, there is a specific security policy. [Kirt97]

3.3.2.7 Summary

COM provides declarative mechanisms to specify machine-wide and component-wide security properties. Thus, security settings can be defined also for components that were developed without any security concerns. On the other hand, declarative security is very coarse-grained as all COM objects in a component share the same declarative security settings.

Programmatic security allows more fine-grained security settings. Components have access to detailed information concerning authentication, authorization, and impersonation of the caller. However, components with hard-coded security settings are often tied to a specific operational environment.

Security auditing is possible only with Win32 API functions. As far as I know, the COM security model provides no non-repudiation service.

COM+ introduces further security concepts, such as the specification of security roles, which allow greater flexibility and simplify the deployment and administration of components.

3.4 CORBA

3.4.1 Overview of CORBA

The *Common Object Request Broker Architecture (CORBA)* is a vendor-independent specification [OMG01b] for an architecture and infrastructure to connect applications in a heterogeneous network environment. These applications can be written in different programming languages and run on different operating systems. CORBA, which is standardized by the *Object Management Group (OMG)*, automates common network programming tasks such as object registration, location, and activation, error-handling, and parameter marshalling and demarshalling.

3.4.1.1 OMA

CORBA applications are composed of objects. The *Object Management Architecture (OMA)* classifies objects into four categories: the CORBA services, CORBA facilities, CORBA domain objects, and application objects. [OMG01c]

CORBA Services

A CORBA service provides *fundamental, domain-independent* functionality to build distributed object applications. Examples are naming, event, lifecycle, transaction, or security services.

CORBA Facilities

CORBA facilities are useful across business domains but are not as fundamental as CORBA services. CORBA facilities, which are also called *horizontal facilities*, include the Printing facility, the Secure Time facility, the Internationalization facility, and the Mobile Agent facility. [OMG01c]

CORBA Domain Objects

CORBA domain objects are services for specific application domains such as finance, healthcare, manufacturing, telecommunication, e-commerce, or transportation.

Application Objects

Application objects are typically customized for an individual application. This category identifies objects that are not affected by OMG standardization efforts. [OMG01c]

3.4.1.2 OMG IDL

Object implementations specify their interfaces in the *OMG Interface Definition Language (OMG IDL).* IDL is programming language independent. OMG has standardized mappings for C, C++, Java, COBOL, Smalltalk, and Ada. IDL is used to generate client *stubs* and object implementation *skeletons.* Stubs and skeletons serve as proxies. A client can access an object only through its interface. The implementation of an object is hidden from the rest of the system. Interfaces are stored in an *interface repository* service. The ORB may use the interface repository information to perform requests. Moreover, the interface repository is used to store additional information associated with interfaces, such as debugging information.

3.4.1.3 ORB

The *Object Request Broker (ORB)* is the communication backbone of a CORBA system. It provides middleware services, which allow clients to perform operations on objects. Objects register at the ORB to provide their services. The ORB is responsible to find an object implementation, to prepare it to receive a request, and to manage necessary control and data transfers. The interface a client sees is independent of the object location, the programming language that was used to implement the object, and the operating system of the server.

A client can use an OMG IDL *stub* or a *Dynamic Invocation Interface (DII)* to make a request. The DII is used to construct requests on interfaces that were not known at compile time. The ORB calls an object implementation either through an OMG IDL generated *skeleton* or through a *Dynamic Skeleton Interface (DSI).* The DSI is the server-side analogue of the DII. It handles method calls for objects that do not have compiled IDL skeletons.

An object implementation uses an *object adapter* to access services provided by the ORB. Such services include the generation and interpretation of object references, security of interactions, mapping of object references to implementations, or registration of implementations. An ORB may have multiple object adapters to target particular groups of object implementations that have similar requirements. Each ORB must support the standard *Basic Object Adapter (BOA).*

The ORB needs information to locate and activate object implementations. Such information is stored in the *Implementation Repository*. It is also used for further information concerning administrative control, resource allocation, and security. Figure 3.4-1 on page 33 gives an overview of an ORB. [OMG01c]

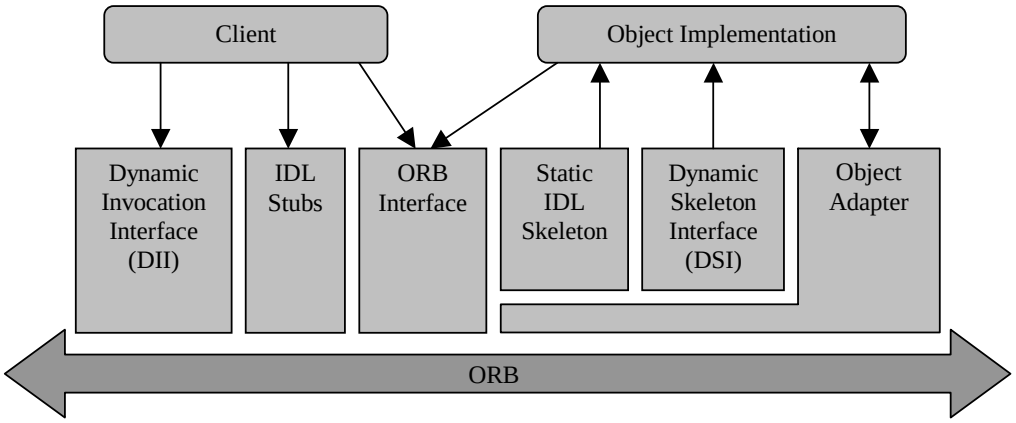

Figure 3.4-1 Structure of an Object Request Broker [OMG01b]

3.4.1.4 GIOP and IIOP

The *General Inter-ORB Protocol (GIOP)* specifies standards for communication between ORBs. It defines transfer syntax and message formats. It is designed to work over any connection-oriented transfer protocol. The *Internet Inter-ORB Protocol (IIOP)* specifies how GIOP messages are exchanged using TCP/IP.

3.4.2 Security in CORBA

3.4.2.1 Overview of the CORBA Security Service

The *CORBA Security Service*, which is also called *CORBAsec*, specifies security architecture and interfaces for a CORBA environment. CORBA security is structured into several feature packages. There are two *security functionality packages* which enforce security on two levels. *Level 1* provides security for applications which are *unaware* of security or which have only limited security requirements. *Security aware* applications use security facilities provided with the *level 2* security functionality package. [OMG01a]

One of the major design principles is the *Common Secure Interoperability (CSI)* between ORB products and Security Services on the one hand, and different Security Services on the other hand. Interoperability between different Security Services is standardized with the *Secure Common Inter-ORB Protocol (SECIOP)*. [AlLa00]

Security Reference Model

The security reference model provides the framework for CORBA security. It describes "how and where a secure system enforces security policies" [OMG01a]. It is a "*meta-policy* because it is intended to encompass all possible security policies supported by the OMA" [OMG01a]. In the following sections, I describe the security mechanisms of the CORBA security reference model.

3.4.2.2 Identification and Authentication

"A principal is a human user or system entity that is registered in and is authentic to the system" [OMG01a]. Principals that initiate activities are called *initiating principals*. Each principal is associated with several *security attributes*, such as an *identity attribute* or *privilege attributes*, which are used to control access rights of a principal. Privilege attributes may have duration limits and controls on where and when they can be used. The security attribute *Public* is available to any principal without authentication. Security attributes of a principal are collected in its *credential* as depicted in Figure 3.4-2 on page 35. A credential object is created by a *principal authenticator* object, which can be called, for example, by a user login application. [OMG01a]

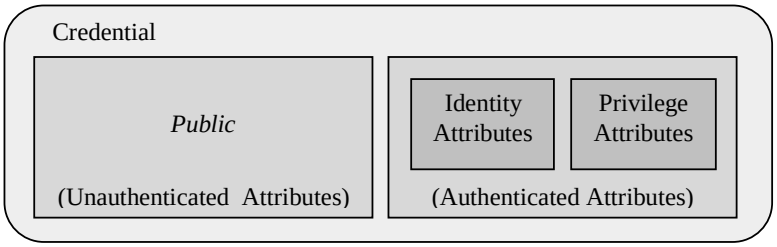

Figure 3.4-2 A credential containing security attributes [OMG01a]

3.4.2.3 Security Domains

A security domain is a distinct scope with common rules and characteristics. There are three types of security domains [OMG01a]:

Security Policy Domains

All objects in a *security policy domain* share a common security policy. A *security policy* concerns access control, authentication, secure object invocation, delegation and accountability. There is a *system security policy*, which is enforced automatically by the ORB and the Security Services, whereas application objects with special security requirements can enforce their own *application security policy*.

A *policy domain manager* provides means to add and remove members for each security policy domain. It is also possible to delegate security policies to *subdomains* thereby forming *policy domain hierarchies*, which can reflect organizational subdivisions and separate administrators' duties. Security policy domains may also be federated.

Security Environment Domains

A security environment domain specifies the scope over which the enforcement of security policies is achieved by means local to the environment. All objects running on the same machine, for example, may trust each other and are therefore in the same security environment domain. The Security Service specification considers two types of environment domains: Objects in the same *message protection domain* do not need to perform integrity or confidentiality checks. Objects in the same *identity domain* share the same identity and thus do not need authentication when invoking each other.

Security Technology Domains

In a specific security technology domain, security demands are enforced with the same security technology. All objects in the same security technology domain use, for example, the same authentication services or the same access control mechanisms.

3.4.2.4 Authorization

Security Service authorization is based on *access decision functions*. An access decision function uses the initiator's *privilege attributes* and the *target control attributes* to enforce access control. Privilege attributes contain, among other things, the principal's access identity and its capabilities. Target control attributes, which may be ACLs, can be shared by categories of objects to avoid overhead on the administration.

The Security Service access control model consists of two layers:

The *object invocation access policy* is enforced automatically for all applications by the ORB and the Security Services on object invocation. A *client side access decision function* controls whether a client can invoke an operation on a target object. A *target side access decision function* defines the conditions that allow a target object to accept an invocation.

The *application access policy* is enforced by an application itself to extend the object invocation access policy. With its own access decision functions, an application can thus perform more fine-grained access control based on parameter values, or the data being accessed. Figure 3.4-3 shows the Security Service access control model.

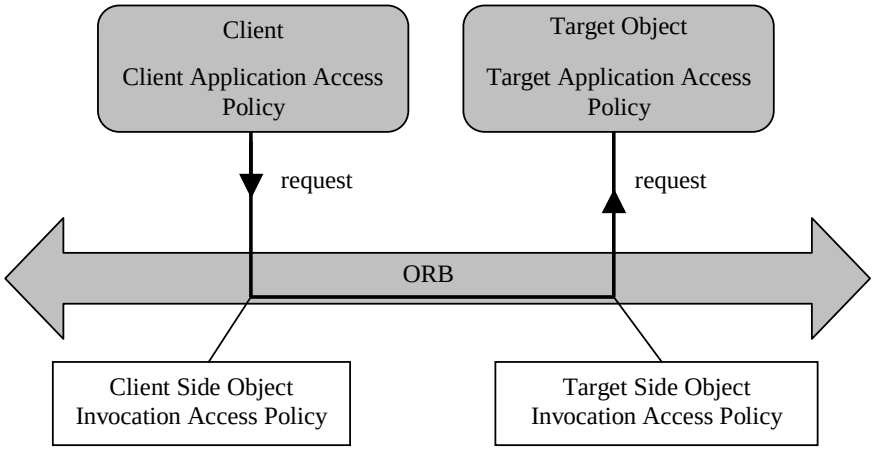

Figure 3.4-3 The Security Service access control model [OMG01a]

3.4.2.5 Secure Communication

When a client requests an operation, the ORB establishes a *secure association*, which is not perceptible to the client and target object. A secure association enforces identification and authentication of the participants. It also negotiates the minimum security level that is acceptable to both parties. The mechanisms used to establish the secure association depend on the individual security policies and the security mechanisms available between the

participants. Security technology, such as secret or public key cryptography, must be implemented by services outside the actual CORBA Security Service implementation. The Security Service specification provides only interfaces for setting secure association policies. [Chiz98]

3.4.2.6 Delegation

A client can delegate some or all of its privilege attributes to another object. The administrator specifies the default delegation, which is automatically performed by the ORB for applications unaware of security.

The Security Service specification [OMG01a] defines the following terms in the context of delegation:

- **Initiator:** The first client in a call chain.
- **Final target:** The final recipient in a call chain.
- **Intermediate:** An object in a call chain that is neither the initiator nor the final target.

When a client calls an object, the object can make calls to other objects, which results in a chain of calls as shown in Figure 3.4-4.

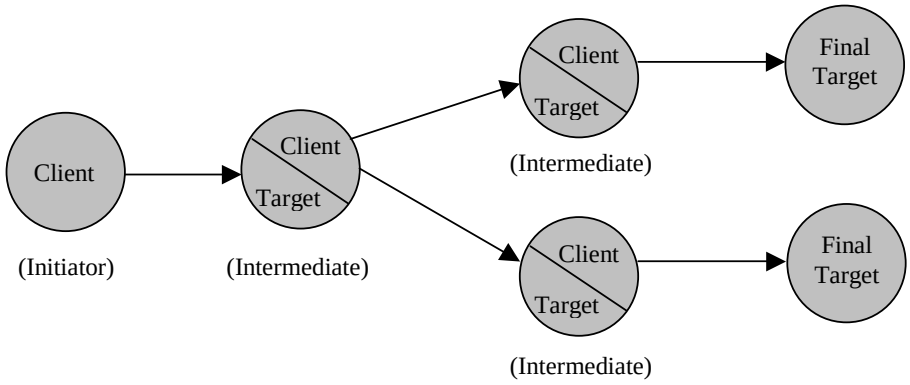

Figure 3.4-4 Chain of Calls [OMG01a]

The Security Service specification describes facilities to restrict delegation. Clients can control which of their privileges are delegated and where individual privileges can be used. The latter is also referred to as *target restriction*. A client can also specify how long or how many invocations a delegation is valid.

The Security Service specification describes the following delegation scenarios [OMG01a]:

Scenario 1 – *No Delegation:*

The client permits the intermediate to use its credentials only for access control decisions but does not permit them to be delegated.

Figure 3.4-5 No Delegation

Scenario 2 – *Simple Delegation:*

The client permits the intermediate to use and to delegate its credentials. The target is not aware of the intermediate object. When the client does not impose target restrictions, simple delegation is equivalent to impersonation.

Figure 3.4-6 Simple Delegation

Scenario 3 – *Composite Delegation:*

The client permits the intermediate to use and to delegate its credentials. The intermediate passes both credentials separately to the target.

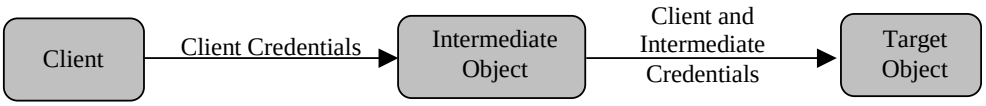

Figure 3.4-7 Composite Delegation

Scenario 4 – *Combined Privileges Delegation:*

The client permits the intermediate to use and to delegate its credentials. The intermediate combines the client's and its own privileges into a new credential. The target cannot distinguish which privileges come from which principal.

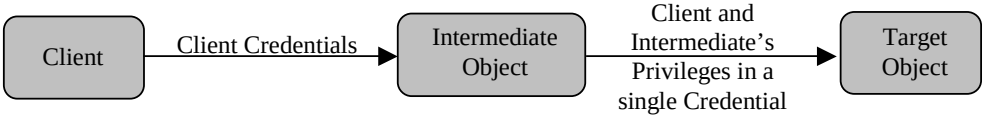

Figure 3.4-8 Combined Privileges Delegation

Scenario 5 – *Traced Delegation:*
The client permits the intermediate to use and to delegate its credentials. Each intermediate object adds its credential to form a chain of credentials.

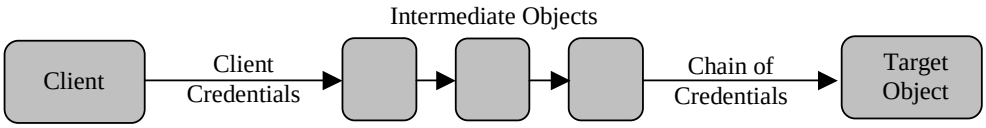

Figure 3.4-9 Traced Delegation

3.4.2.7 Security Auditing

Security auditing is used to record security relevant events in a CORBA system. It is used to detect actual or attempted security violations. Events that should be audited are specified with *audit policies*. The Security Service specification distinguishes two types: system audit policies and application audit policies. *System audit policies* are enforced automatically for all applications and record *system events*, such as authentication of principals, success or failure of object invocation, or administration of security policies. With *application audit policies*, applications can specify the *application events*, that should be audited. Application events depend on the specific application. [OMG01a]

3.4.2.8 Non-repudiation

The CORBA Security Service specification defines a *non-repudiation (NR)* service, which provides facilities to *generate* and *verify* irrefutable evidence about a claimed event or action. NR services are under the control of the applications, rather than being automatically enforced by the ORB.
The CORBA Security NR service is based on the *International Standards Organization (ISO)* non-repudiation model but provides only *Evidence Generation and Verification*. The ISO NR model additionally provides *Evidence Storage and Retrieval*, and a *Delivery Authority*. [Chiz98, OMG01a]

3.4.2.9 Summary

The CORBA Security Service specification tries to encompass all possible security requirements. The main goal is to provide a consistent security system that scales from small to large networks, is available as transparently as possible, independent from the underlying security technology, and interoperable between different implementations with and without security. Security domains allow to group objects under a common security policy. Client machine protection is not an issue of CORBA, as CORBA supports many programming languages. For example, a CORBA client may be programmed in Java and restricted by the Java 2 Security Model.

4 Comparison

In this chapter, I compare the security models of EJB, COM, and CORBA. I do not include Jini into this comparison, because it provides no security additional to the Java 2 Security Model.

Security Awareness

EJB: Enterprise beans should be developed without security awareness. Security settings for EJB applications are specified declaratively in the *Deployment Descriptor* and enforced by the EJB Container and the EJB Server.

COM: Declarative security settings are specified in the system registry for security unaware components. COM additionally provides several functions and interfaces to enforce security programmatically. Some requirements can be met only with programmatic security.

CORBA: Security for unaware applications is managed automatically by the ORB and the Security Service. The Security Service specification also describes interfaces for security aware applications.

Secure Communication

EJB: Communication security is EJB container-specific and set by the Deployer.

COM: Integrity and encryption of messages are set declaratively or programmatically with the *Authentication Level* flags.

CORBA: ORB and Security Service can establish transparent *secure associations* between communicating parties according to their individual security policies. The Security Service does not implement cryptographic algorithms.

Identification and Authentication

EJB: Identification and authentication is also EJB container-specific and not addressed by the EJB specification.

COM: Microsoft defines the *Security Support Provider Interface (SSPI)* to identify and authenticate security principals. COM also defines *Authentication Level* flags to specify authentication requirements.

CORBA: In a CORBA system, users are authenticated with a *Principal Authenticator*, which generates *credentials* containing security attributes.

Authorization

EJB: The EJB specification defines how *security roles* and *method permissions* are set in the Deployment Descriptor. Although not recommended by the EJB specification, authorization can also be enforced programmatically.

COM: COM distinguishes between *launch security* and *access security*. Declarative launch and access security can be enforced only on component level. More fine-grained access control is possible programmatically with functions and interfaces from the COM library. COM+ allows fine-grained declarative access control with security roles.

CORBA: Security Service authorization uses *access decision functions* on client and on target side. Target side access decision functions make it possible to control from which clients an object can accept calls. There is an *object invocation access policy*, which is enforced automatically, whereas the *application access policy* allows programmatic access control.

Delegation

EJB: EJB supports only two delegation scenarios, which the Deployer specifies in the Deployment Descriptor: the propagation of the *caller's security identity* and the propagation of a *specific run-as identity*.

COM: Clients can set delegation restrictions with four *Impersonation Level* flags that range from autonomous calls to full delegation.

CORBA: The CORBA Security Service allows clients to control carefully which of its rights are delegated to which objects and how long the delegated rights are valid.

Client Machine Protection

EJB: EJB is based on the Java technology, which provides the Java 2 Security Model.

COM: Components execute under a specific user account associated with corresponding permissions.

CORBA: Client machine protection is not an issue of CORBA Security, as CORBA is language independent.

Security Auditing

EJB: The EJB specification does not define any interfaces or Deployment Descriptor entries for security auditing. Security auditing may be optionally provided by the EJB container.

COM: Security auditing is possible only with Win32 API functions, which ties a component to the Windows NT platform.

CORBA: The Security Service specification distinguishes between *system audit policies* and *application audit policies.*

Non-repudiation

EJB, COM: As far as I know, non-repudiation is not addressed by EJB and COM.

CORBA: Applications can use a non-repudiation service, which generates and verifies evidence.

5 Conclusion

In this work, I described security requirements and solutions in distributed component models. Generally, there are the following security requirements for distributed applications: secure communication, identification and authentication, authorization, delegation, client machine protection, security auditing, and non-repudiation.

The EJB specification defines how application programmers can specify authorization and delegation requirements. It does not define how to specify secure communication, authentication, security auditing, and non-repudiation requirements for an EJB application. The EJB container and the EJB server must provide most of the work to secure a distributed application. Client machine protection is solved with the Java 2 Security Model.

Jini provides no security mechanisms additional to the Java 2 Security Model. Therefore, I outlined security requirements and extensions for secure communication, authentication, and authorization. Security extensions require major changes to the Jini architecture, which reduce the spontaneity of a Jini network. I did not propose extensions for delegation, security auditing, and non-repudiation.

COM provides only coarse-grained declarative security. Programmatic security is extensive but ties a COM application to a specific operational environment. COM+ introduces further security concepts and more fine-grained declarative means. Security auditing is based on the Windows NT platform. COM does not provide a non-repudiation service.

The CORBA Security Service reference model describes a general security model on an abstract level, which is interoperable between different ORP products and Security Services. It comprises all security mechanisms that I examined in this work, except client machine protection. I did not examine CORBA Security Service implementations.

Not included in this work is administration of security information, such as the management of user accounts and policy settings. This issue is also important, as security holes are more likely to arise when the administration of security mechanisms is too complex and time-consuming.

6 References

[Alla00]
Allamaraju, Subrahmanyam; et al.: *Professional Java Server Programming J2EE Edition.* Wrox Press Ltd, Birmingham 2000

[AlLa00]
Alireza, A.; Lang, U.; Padelis, M.; Schreiner, R.; Schumacher, M.: *The Challenges of CORBA Security.* http://www.cl.cam.ac.uk/~ul201/research.html, 2000

[Cetu01]
Cetus Links: *Distributed Objects & Components: CORBA.* http://www.cetus-links.org/oo_corba.html, 2001

[Chiz98]
Chizmadia, David: *A Quick Tour Of the CORBA Security Service*. Information Security Bulletin September 1998, http://omg.com/news/corbasec.htm

[DeYa00]
DeMichiel, Linda G.; Yalçinalp, Ümit L.; Krishnan, Sanjeev: *Enterprise JavaBeans™ Specification, Version 2.0.* Proposed Final Draft. Sun Microsystems, Inc., Palo Alto 2000

[Dule01]
Dulepet, Rajiv: *COM Security in Practice.* http://msdn.microsoft.com/library/techart/msdn_practicom.htm, Microsoft Corporation, Redmond 2001

[Eddo99]
Eddon, Guy: *The COM+ Security Model Gets You out of the Security Programming Business.* Microsoft Systems Journal, November 1999, http://www.microsoft.com/msj/

[EdEd98]
Eddon, Guy; Eddon Henry: *Inside Distributed COM.* Microsoft Press, Redmond 1998

[Edwa99]
Edwards, W. Keith: *Core Jini*. Prentice Hall, New Jersey 1999

[ErNi01]
Eronen, Pasi; Nikander, Pekka: Decentralized Jini Security. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS 2001),* San Diego 2001, pages 161-172

[GoMu97]
Gong, Li; Mueller, Marianne; Prafullchandra, Hemma; Schemers, Roland: Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java™ Development Kit 1.2. In: *Proceedings of the USENIX Symposium on Internet Technologies and Systems,* Monterey 1997

[Gong98]
Gong, Li: *Java™ 2 Platform Security Architecture.* Sun Microsystems, Inc., Palo Alto 1998

[GrTh00]
Gruhn, Volker; Thiel, Andreas: *Komponentenmodelle - DCOM, JavaBeans, Enterprise JavaBeans, CORBA.* Addison-Wesley, Reading 2000

[HaKe00]
Hasselmeyer, Peer; Kehr, Roger; Voß, Marco: *Trade-offs in a Secure Jini Service Architecture*. Department of Computer Science, Darmstadt University of Technology 2000

[HoKi97]
Horstmann, Markus; Kirtland, Mary: *DCOM Architecture.* http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomarch.htm, Microsoft Corporation, Redmond 1997

[Ipla98]
iPlanet™: *Introduction to SSL.* http://docs.iplanet.com/docs/manuals/security/sslin/index.htm, 1998

[Kass00]
Kassem, Nicholas; the Enterprise Team: *Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition, Version 1.0.1.* Sun Microsystems, Inc., Palo Alto 2000

[Kirt97]
Kirtland, Mary: *Object-Oriented Software Development Made Simple with COM+ Runtime Services.* Microsoft Systems Journal, November 1997, http://www.microsoft.com/msj/

[KuRo00]
Kurose, James F.; Ross, Keith W.: *Computer Networking A Top-Down Approach Featuring the Internet.* Addison-Wesley, Reading 2000

[Li01]
Li, Sing: *Java security evolution, Part 2.* http://www-106.ibm.com/developerworks/java/library/j-secevol2/index.html, IBM 2001

[Msdn01]
Msdn Online Library, Microsoft Corporation, Redmond 2001 http://msdn.microsoft.com/library/psdk/winbase/accctrl_0v5a.htm

[Msdn99]
Msdn Training: *Mastering Distributed Application Design and Development Using Microsoft® Visual Studio 6.* Microsoft Corporation, Redmond 1999

[OMG01a]
Object Management Group, Inc.: *Security Service Specification.* Version 1.7 March 2001, http://www.omg.org/technology/documents/formal/omg_security.htm

[OMG01b]
Object Management Group, Inc.: *The Common Object Request Broker: Architecture and Specification.* Version 2.4.2 February 2001, http://www.omg.org/technology/documents/formal/corbaiiop.htm

[OMG01c]
Object Management Group, Inc.: *Getting Started with OMG Specifications and Process.* http://www.omg.org/gettingstarted, 2001

[Sdk01]
Platform SDK Documentation, http://msdn.microsoft.com/library/, Microsoft Corporation, Redmond 2001

[Sun01]
Sun Microsystems, Inc.: *Java™ 2 Platform, Standard Edition, v 1.3 API Specification.* http://java.sun.com/j2se/1.3/docs/, Palo Alto 2001

[Sun01a]
Sun Microsystems, Inc.: *Enterprise JavaBeans™ technology.* http://java.sun.com/products/ejb/, Palo Alto 2001

[Sun01b]
Sun Microsystems, Inc: *Enterprise JavaBeans™ Frequently Asked Questions.* http://java.sun.com/products/ejb/faq.html, Palo Alto 2001

[Sun99]
Sun Microsystems, Inc: *Jini™ Architectural Overview.* Palo Alto 1999

[Szyp98]
Szyperski, Clemens: *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, Reading 1998

[Thom98]
Thomas, Anne: *Enterprise JavaBeans™ Technology - Server Component Model for the Java™ Platform.* Patricia Seybold Group, Boston 1998

[Thom99]
Thomas, Anne: *Java 2 Platform, Enterprise Edition – Ensuring Consistency, Portability, and Interoperability.* Patricia Seybold Group, Boston 1999

[WiKi94]
Williams, Sara; Kindel, Charlie: *The Component Object Model: A Technical Overview.* http://msdn.microsoft.com/library/techart/msdn_comppr.htm, Microsoft Corporation, Redmond 1994