administrators. Administrators have to establish the credentials of principals and control the attributes of objects in such a way that only those principals that should be allowed to request operation executions can actually do so. The security mechanisms of object-oriented middleware have to provide the primitives that support administrators in implementing these access control policies.

Access rights define the modes of access that principals have to server objects.

Access control policies determine the principals that are allowed to execute a particular operation based on *access rights*. Access rights are usually formulated in terms of access modes. An access mode might be the ability to get an attribute value or to set an attribute value or to use a particular operation. There are three parts to an access control policy. First, access rights have to be granted to a principal and stored in the principal's credentials. Secondly, a set of access rights for being able to execute an operation has to be defined. Finally, the policy has to define how the rights in the set of required access rights have to be combined in order to determine whether or not access is granted. A principal who wishes to execute a particular operation must previously have been granted those rights that are required for this operation, as determined by the policy.

Access rights are often defined for types rather than objects.

Administrators may wish to set access rights for individual objects, but this may be too impractical if there are many instances of a type. The definition of access rights is therefore often determined based on the types of object. This means that all instances of the object type then share the same set of required access rights. It is also desirable to be able to determine access rights for object types that have not been designed to have access rights. This means that definition of access rights should be isolated from the interfaces and be done by a separate required rights object.

The *required rights object* can be seen as an interface to a database of access rights. The database contains, for the operations in each type of object or even in individual objects, the access rights that are required. Object-oriented middleware products will provide user interfaces so that administrators can set up and maintain this database. At run-time, the middleware will utilize the database in order to perform access control checks.

12.3.4 Non-Repudiation

Non-repudiation makes principals accountable for their actions.

Secure object-oriented middleware often include non-repudiation services, which make users and other principals accountable for their actions. They achieve this accountability by the collection of evidence

1. that a principal has requested a particular operation, which prevents a requester falsely denying a request for an operation, and
2. that a server has received a request for a particular operation, which prevents a servers falsely denying receipt of a request.

It is important that the non-repudiation service collects the evidence in such a way that the principals using the client or server cannot repudiate the evidence later, and that the evidence can be used by an *adjudicator* to settle disputes.

Figure 12.8 shows an overview of the different components of a non-repudiation service according to the ISO 7498-2 standard [ISO 7498-2, 1989]. The most important of those components generates verifiable evidence about creation and receipt of object requests. A

Consider a funds transfer between two accounts. In a conventional setting, a customer walks into a bank branch and asks for a funds transfer of $1,000 from one account to another. The bank will make him write a cheque and sign it. The bank will archive the cheque and the signature and date on the cheque will be used as evidence that the customer actually demanded the funds transfer. The bank needs to collect similar evidence in a direct banking setting, where the funds transfer is requested via the Internet.

Example 12.11
Non-Repudiation in the Banking Example

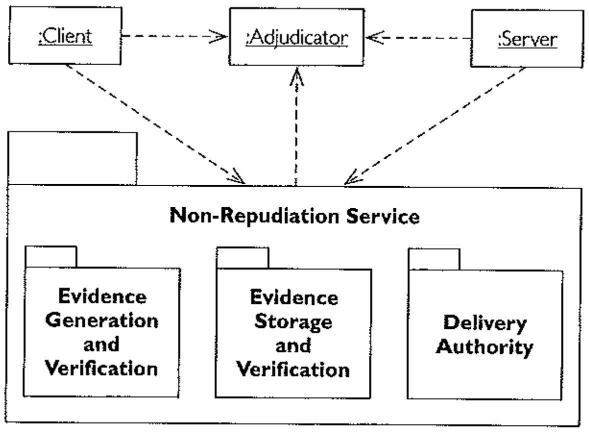
:Client
:Adjudicator
:Server
Non-Repudiation Service
Evidence Generation and Verification
Evidence Storage and Verification
Delivery Authority

Figure 12.8
Overview of Non-Repudiation Service Components

non-repudiation service may store this evidence on behalf of clients and servers so that evidence can be retrieved when a dispute arises. A delivery authority generates proof of origin so that clients and servers cannot repudiate that it was actually them who created or received an object request. Note that these components of a non-repudiation service have to be trusted by both clients and servers so that they both accept the evidence that is provided by the service.

It is worthwhile to note that non-repudiation may involve significant overheads. Therefore, such evidence is only collected for those object requests that have legally binding implications in the domain in which the distributed object system executes. In our direct banking application, for example, evidence will be needed for funds transfer requests, but no evidence will be needed for the request of an operation to return the account balance if we can assume that these requests are performed free of charge. Non-repudiation services are therefore only provided when client or server objects explicitly demand the creation of evidence. Hence the usage of non-repudiation is not transparent to designers of client and server objects.

Evidence is provided in the form of *non-repudiation tokens* that are handed out to clients when servers have received an object request and to servers when clients have created an object request. There are two types of token. The first form of token includes all data that is needed for the evidence. The second form of token is a key that can then be used in a request to the non-repudiation service to produce the data that is needed for the evidence.

Non-repudiation is implemented using digital signatures, which rely on encryption.

The implementation of non-repudiation relies on encryption techniques, just as the implementation of authentication does. In particular, non-repudiation uses the concept of *digital signatures*. To create a digital signature, the client encrypts the principal's identity with the secret key of that principal. Non-repudiation services demand that clients include such a digital signature in the data that is submitted for the generation of a non-repudiation token. The non-repudiation service then stores that data together with the signature as evidence that the principal whose signature was included has requested the service. The same mechanism applies for servers that generate evidence that a certain request has been achieved. If a dispute arises an adjudicator uses a certified public key of the principal to validate that indeed that particular principal has requested generation of the non-repudiation token.

12.3.5 Security Auditing

The services that we have discussed above have in common that they provide proactive measures to prevent attacks. Administrators also need to react to attacks that were attempted or that actually succeeded. Hence, it is necessary that a distributed object system maintains records about security incidents. *Security auditing mechanisms* are concerned with logging security-relevant events on persistent storage so that they can be analyzed by administrators.

Auditing policies determine the analysis of logs of security events.

Potentially, many different events can be logged and administrators want to be able to restrict the amount of information to events that are relevant to their particular distributed object system. Security auditing mechanisms provide the basic primitives for administrators to establish *auditing policies*. These auditing policies determine which events should be logged, how long the logs should be kept and which analyses should be performed on these logs. Two kinds of auditing policy can be distinguished. *System audit policies* determine the system events that the middleware logs in a way that is transparent to both client and server programmers. Examples of system events that may be considered relevant to security are authentication of principals, updates to a principal's credentials or the success or failure of object requests. *Application audit policies* determine the application events that client and server objects log to assist with security auditing; they are obviously not transparent to programmers. Examples of application events in our Internet direct banking application might be the request of a funds transfer including the request parameters.

Example 12.12
Security Auditing Requirements

An audit policy for our direct banking application should, for example, log the credentials of a principal of the bank and whether or not they are successfully authenticated. The policy should also record any failures in access to the credit or debit operation of a bank account.

Events can be logged on files in order to provide *audit trails* for later security analyses. Administrators may, however, also wish to use automated analysis facilities for more serious security incidents. These facilities analyse those events that correspond to serious security breaches and take active measures to alert the administrator if such an event occurs so that the administrator can take measures to locate the attacker and record evidence about the attack.

12.4 Security Services in Object-Oriented Middleware

The previous section presented the principles of higher-level security services and showed how encryption is used in their implementations. We have identified how security services support controlling the access of authorized principals. We have seen how they validate that principals are who they claim to be; we have seen how irrefutable evidence can be generated in order to make principals accountable for what they do; and we have identified how security audits can be achieved.

Java has had a security model from when it was first released. The Java security specification focuses on how to shield the host from (potentially malicious) applet code that may be downloaded from remote servers to execute in a Java Virtual Machine. While this is an important security concern for Java as an Internet programming language, this does not address the security problems that we raised above.

CORBA, COM and Java/RMI security models are similar.

The security models available in CORBA, COM and Java/RMI are quite similar. COM does not support non-repudiation but it does cover the other higher-level services that we discussed in the previous section. RMI does not support non-repudiation or object invocation access policies but enables server designers to specify application access security policies. We now discuss the higher-level security services that are provided by OMG/CORBA in order to show an example of how the security principles that we discussed above are supported in current object-oriented middleware.

The OMG determines security mechanisms for CORBA as part of the CORBAservices. It includes the CORBA Security service specification and defines how to secure distributed CORBA objects against attacks. The CORBA security service defines the CORBA interfaces that are provided to client developers, server developers and administrators for access control, authentication, non-repudiation and auditing purposes. We briefly discuss these interfaces in this section now and indicate by whom they would be used.

12.4.1 Authentication

The CORBA Security service assumes that a client object establishes the credentials of its principal using authentication prior to requesting any operations from a secure distributed object system. The `Credentials` interface shown in Figure 12.9 defines the interfaces that are available for a CORBA credentials object. A `Credentials` object is provided by operation `authenticate` that is part of interface `PrincipalAuthenticator`. To authenticate a principal, a client identifies an authentication method, such as password-based authentication, the security identifier of the principal and the authentication data, which is a password in the case of password-based authentication. Moreover, the client submits a list of attributes that identify the access rights that the principal will need in order to access distributed server objects during a session.

After a successful authentication, the CORBA Security service stores the credentials of a principal in the `Current` object that is associated with the session. We have already seen a