Technical Report

UCAM-CL-TR-564
ISSN 1476-2986

Number 564

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Access policies for middleware

Ulrich Lang

May 2003

This technical report is based on a dissertation submitted March 2003 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson College.

Series editor: Markus Kuhn

ISSN 1476-2986

Abstract

This dissertation examines how the architectural layering of middleware constrains the design of a middleware security architecture, and analyses the complications that arise from that. First, we define a precise notion of middleware that includes its architecture and features. Our definition is based on the Common Object Request Broker Architecture (CORBA), which is used throughout this dissertation both as a reference technology and as a basis for a proof of concept implementation. In several steps, we construct a security model that fits to the described middleware architecture. The model facilitates conceptual reasoning about security. The results of our analysis indicate that the cryptographic identities available on the lower layers of the security model are only of limited use for expressing fine-grained security policies, because they are separated from the application layer entities by the middleware layer. To express individual application layer entities in access policies, additional more fine-grained descriptors are required. To solve this problem for the target side (i.e., the receiving side of an invocation), we propose an improved middleware security model that supports individual access policies on a per-target basis. The model is based on so-called "resource descriptors", which are used in addition to cryptographic identities to describe application layer entities in access policies. To be useful, descriptors need to fulfil a number of properties, such as local uniqueness and persistency. Next, we examine the information available at the middleware layer for its usefulness as resource descriptors, in particular the interface name and the instance information inside the object reference. Unfortunately neither fulfils all required properties. However, it is possible to obtain resource descriptors on the target side through a mapping process that links target instance information to an externally provided descriptor. We describe both the mapping configuration when the target is instantiated and the mapping process at invocation time. A proof of concept implementation, which contains a number of technical improvements over earlier attempts to solve this problem, shows that this approach is useable in practice, even for complex architectures, such as CORBA and CORBASec (the security services specified for CORBA). Finally, we examine the security approaches of several related middleware technologies that have emerged since the specification of CORBA and CORBASec, and show the applicability of the resource descriptor mapping.

Contents

Chapter 1

Introduction

Over the last decade, the IT industry was increasingly faced with the task of integrating networked information sources and applications across heterogeneous hardware and software platforms. Initially, the main driving factor for this trend was the demand for integration of different state-of-the-art systems with incompatible legacy IT systems within the intranets of large organisations, such as systems for ordering, parts tracking, and billing. Another reason was the advent of the Internet in the commercial arena during the mid-1990's, which enabled new forms of information access and exchange, both for businesses and consumers. For example, staff on the road should be able to access corporate data resources globally when dealing with customers, and consumers should be able to access applications such as online banking from any home PC. Finally, towards the end of the 1990's until today, the increased use of a wide range of rapidly evolving incompatible wireless pocket computers and mobile phones has spurred attempts in the telecommunications industry to develop a device-independent application platform, which would allow applications (that offer an increasing number of multimedia and interactive features) to be written once and then ported to new devices with little extra effort.

To meet this growing demand for integration, several architectures have been designed and implemented that integrate both legacy and state-of-the-art systems in a cost-effective way, and in a fashion that fits seamlessly into the programmers' usage pattern. The term "middleware" emerged as the generic industry term for any system that "glues together" various networked applications. Examples of such middleware systems are OSF's Distributed Computing Environment (DCE) [127], Common Object Request Broker Architecture (CORBA) [119], Enterprise Java Beans (EJB) [101] and recently Microsoft .NET [106]. In the telecommunications world, the Telecommunications Information Networking Architecture (TINA) [36], and the Parlay APIs for Open Service Access [130] have been designed.

All these middleware technologies have in common that a lot of effort has been put into incorporating a very rich set of features, while at the same time maintaining a high degree of flexibility. On the flipside, the resulting architectures became very complex and highly layered.

It was realised early on in the design phase of most middleware technologies that, due to the complex and unpredictable interactions between applications across insecure networks such as the Internet, security would also play an important role. To fit seamlessly into the application development process, security should be provided automatically as part of the middleware technology (i.e., without active involvement of the application programmer). The security features required by customers included authentication, message protection, access control, event auditing, and in some cases non-repudiation. To avoid cutting down on the wealth of features and flexibility, the architectural position of the security features and their interactions were largely determined by the underlying middleware architecture, and not by any conceptual reasoning about security. The fact that the inclusion of security features alone would not automatically provide a middleware security architecture was either only realised afterwards, or it was accepted that a rich set of features was a better selling factor than an effective security architecture.

1.1 Goal

This dissertation tries to identify how the architectural design of middleware (using CORBA as the reference technology) constrains the design of a middleware security architecture, and tries to analyse the complications that arise from that. To allow reasoning about the intended and effective functionality of such a security architecture, a conceptual middleware security model is constructed in several incremental steps.

The results of our analysis indicate that (cryptographic) identities provided by the network layer are only of limited use for expressing useful security properties, and that additional descriptors are necessary to express individual application layer entities in access control (and audit) policies. In this dissertation, we therefore propose an improved middleware security model that is based on (non-cryptographic) "resource descriptors", which can be used in addition to identities to describe application layer entities in access control (and audit) policies. Such descriptors need to fulfil a number of properties, such as local uniqueness and persistency. Resource descriptors can be obtained on the target side (i.e. the receiving end of an invocation) through a mapping process, which is based on a mapping table that links target instance information to a descriptor. To show the viability of this approach, we also outline a proof-of-concept implementation of a resource descriptor mapper for CORBA.

1.2 Scope

As mentioned above, there are a number of differing middleware technologies, which all have their own specific notion of middleware and a different set of features and properties. However, there are also many architectural parallels between these systems. As a result, the general thesis of this dissertation applies to any middleware technology that fulfils the basic middleware properties defined in section 2.4.1. Although it would be useful to verify this claim by analysing all of the mentioned technologies (see section 8.1), almost all of the research that forms the basis of this dissertation has been done using CORBA for the following reasons:

- CORBA has been widely used in practice for a long time, thus many implementations and case studies are available
- CORBA includes a security system, the CORBA security services (CORBASec), which is a good basis for our analysis

- CORBA and CORBASec specify a rich set of features, whilst trying to maintain flexibility, which makes them an interesting and challenging reference technology for this dissertation

- Both CORBA and CORBASec are based on freely available specifications

As a consequence, we define the term "middleware" in line with CORBA (see definition 1 on page 15), and later on reason about a security model that resembles a simplified form of CORBASec. Our proof-of-concept implementation has also been developed for CORBA and CORBASec.

The security approaches of several other middleware technologies, which have emerged since the specification of CORBA and CORBASec, are examined in chapter 7, where we also show the general applicability of our approach.

1.3 Related Work

One of the key design tasks for most security systems is to decide which security features should be implemented on which system layer. Of particular importance is the architectural position of the access control functionality, which is traditionally described in terms of a reference monitor that checks whether subjects are authorised to access the objects they request (see section 3.2.1).

On one extreme of the spectrum, access control can be implemented as part of the application logic, which allows policy and enforcement to be tailored to the specifics of each application. However, on the downside such custom approaches make application development, reuse, and security administration more difficult, and also complicate conceptual reasoning about security effectiveness and policy consistency. Examples of application layer security include web browsers (e.g., Microsoft Internet Explorer [107]), various media viewers and players (e.g., Adobe Acrobat [2], Microsoft Windows Media Rights Manager [133]), and to some extent database systems (e.g., Oracle database security [31]).

On the other extreme, many systems have been designed that completely separate access control policy and enforcement from the application logic, which allows for consistency, reusability, and easy administration. However, the policies that can be expressed and enforced by such systems often do not match the real security needs of the protected application. This problem normally gets aggravated the lower the layer gets on which access control is implemented because less information about the application logic is available: operating systems (e.g. Unix) reside on a low layer and base their access control on user rights to access processes, files, and resources [49], and not on any information related to the application logic. Access via the network can also be controlled using standard mechanisms (e.g., SSL/TLS [32]) or more elaborate approaches. For example, a multi-level secure system has been designed in which labels correspond to network nodes and illegal information flow is prevented by the communications protocols [139]. On top of that, a virtual machine layer (e.g., Java [54]) may exist that interprets application byte code. On this layer, access control policies can be expressed using the application's object interfaces and operations for targets, and privileges for clients [51]. In addition, some information from lower layers is often available (e.g., network endpoints). Specifics of an invocation are also available in the middleware layer (e.g., CORBA, EJB, DCE). For example, the standard CORBASec model partly bases its access control on the invoked interface and operation. Again, some information from lower layers is available.

Not directly related, access to applications over the network can also be controlled on various layers using firewalls [26]. Often, these work on the lower protocol layers [151], such as TIS `plug_gw` [163] or SOCKS [92]. There have also been some attempts to implement firewalls on higher layers, but often with mixed results due to the fundamental conflict between end-to-end security and firewall security [89].

All these access control systems share the common drawback that it is often not possible to express the fine-grained policies needed for real-world applications, because the information available to express policies is limited by what is available on the layer the access control resides at. In the remainder of this section, we will describe various models and systems related to our work, which try to improve existing access control systems on various layers by adding expressiveness.

The most closely related work is the OMG Resource Access Decision Facility (RAD) specification [123], which puts access control in the application layer, but outside the application logic. It provides a uniform way for CORBA applications to query an authorisation service while invocations are processed, in order to enforce resource-oriented access control policies. By standardizing this service in a technology and policy unspecific way, it allows the definition and administration of a centralised and consistent policy for several applications. A model and implementation of role-based access control $RBAC_0$-$RBAC_3$ [140] for RAD can be found in [13]. Although this approach can express relatively fine-grained policies, it cannot be provided transparently to the application.

Several other interesting models are based on the idea that invocations are redirected to a security object before they are allowed to pass through to the actual target object. One model modifies the semantics of object references in such a way that security meta objects [136, 98] can enforce access control on the invocation before it reaches the target object. Other approaches propose byte code rewriting to implement structural reflection [37, 39], as well as behavioural reflection [173]. Such systems allow the specification of more fine-grained access policies than the ones supported by the underlying platform. However, this approach can only be easily implemented if the source code is available (which is generally not the case for legacy systems) and written in programming languages where the code can be parsed well, such as Java byte code.

Although not directly related to this dissertation, some other work on CORBASec access control will briefly be mentioned: a formal language for describing access control based on the CORBASec credentials model has been presented in [64]. Another formal definition and analysis of CORBASec authorisation in terms of an access control matrix is given in [76], and its use for describing a number of policies, in particular mandatory access control, is discussed. Also, a view-based access model and a declarative specification language for CORBASec have been specified in [21].

The security approaches of several middleware technologies other than CORBA are examined in chapter 7.

Chapter 2

Middleware and CORBA

2.1 Introduction

This chapter introduces the middleware model as a refinement of the object-oriented model and remote procedure call. The middleware model consists of a number of layers, and one of its main purposes is to abstract details about communications and the underlying hardware and software from the application. We describe the main design requirements for a middleware architecture, as well as the main layers and abstraction interfaces.

We then present the Common Object Request Broker Architecture (CORBA) [119], an industry standard that defines interfaces and semantics for object-oriented middleware. It will be used as the reference middleware technology throughout this dissertation. We will show later how CORBA has been extended by a security architecture, the CORBA security services (see chapter 5). CORBA illustrates well that the middleware layering can be complex, and that only parts of the addressing information can be interpreted at each layer. In particular, CORBA introduces the concept of object adapters, which play a role in our proof-of-concept implementation of the resource descriptor mapping (see chapter 6).

2.2 Object-Oriented Model

In the object-oriented programming model [23], "classes" are defined as specifications for software entities that encapsulate both attributes and methods. The class definition specifies the "interface" with all the methods that are available to the client. "Objects" are the dynamically created software instances of classes. An invocation consists of a request from the client to the object and an (optional) reply from the object to the client (see figure 2.1).

The encapsulation concept of object-orientation captures the idea that the variables used inside classes should only be accessed from the method implementations within the same class. From the outside, only the interface and its methods are visible, any data and implementation details are hidden from the caller. The purpose of this concept, which is sometimes also called "information hiding" (because the client can not directly access the object state, only through interfaces), is to allow the software engineering task to be broken down into manageable pieces.

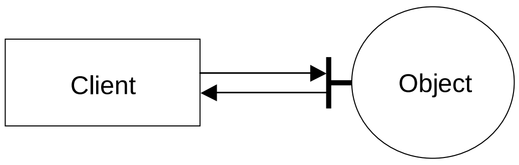

Figure 2.1: Object-Oriented Method Invocation

The object-oriented model also supports inheritance and polymorphism. Inheritance defines a relation between classes that expresses that one class is the specialisation of another class. In other words, a sub-class can inherit attributes and methods from another class, which is called the parent class. Sub-type classes can also overload inherited methods, in which case the polymorphism property (also called late binding) allows the selection of either the method defined in the sub-class or the method defined in the parent class, depending on the arguments passed with the method invocation.

Object-oriented programming languages also often support the use of abstract classes, which act as a template for inheritance, but from which no objects can be instantiated. Abstract classes can contain methods for which no implementation (often called "body") is defined. It is the task of the sub-class to provide this implementation. Other features of object-oriented programming languages include public/private declarations for attributes, and exception handling.

2.3 Remote Procedure Call (RPC)

The described object-oriented programming model assumes that both the client and the target object are part of the same application program. If the client and the target applications reside on different hosts of a network, then an additional software layer needs to be introduced that takes care of all communications across the network. The purpose of Remote Procedure Call (RPC) libraries is to hide the intricacies of the network behind the ordinary invocation mechanism. A client invokes a method on a remote target and suspends itself until it gets back the results. Parameters are passed like in any ordinary procedure call. Under the covers, the RPC run-time software collects values for the parameters, forms a message, and sends it to the remote server. The server receives the request, unpacks the parameters, calls the procedure, and sends the reply back to the client.

While RPCs make life easier for the programmer, they pose a number of challenges to the RPC designer, such as target location and activation, binding, parameter format definition and data representation, failure handling, and security. The best known example for RPC is SunRPC [156], a communications Application Program Interface (API) developed by Sun Microsystems in 1988.

2.4 Middleware

In the IT industry, middleware is a general term for any software that serves to "glue together" or mediate between two separate and often already existing programs.

In Sun's Enterprise Java Beans (EJB) programming model, the term middleware is used to describe software that runs on a server, and acts as either an application processing gateway or a routing bridge between remote clients and data sources or other servers, or any combination of these. Other sources refer to transaction processing monitors, database access systems, or message passing systems as middleware.

In CORBA [119], middleware is defined similarly to RPCs in that it consists of a layer of software between client and target object that delivers extra functionality and hides the complexity of this extra functionality behind a common set of interfaces that clients and targets can invoke. This central architectural component is called Object Request Broker (ORB). It automatically handles all invocations between application entities, regardless whether they reside on the same ORB, different ORBs on the same host, or on different ORBs on different networked hosts. In addition to the RPC-like (blocking) request-reply style described above, ORBs can provide various other styles of communications, such as asynchronous messaging (message-oriented middleware) or publish-and-subscribe via events. Asynchronous messaging allows clients to continue executing after they have invoked a target. At a later stage, the target will call back the client and provide the reply as part of a new (reverse) invocation. Publish-and-subscribe allows targets to publish information in event channels, while clients can subscribe to the relevant event channels to receive the information whenever it is available.

In this dissertation, we define middleware as follows:

Definition 1
Middleware is software that resides between an application and the inner workings of the system hosting the application, and that abstracts the complexities of the underlying technology (see section 2.4.1) from the application layer. In particular, middleware automatically handles all communications related to invocations between client and target applications, and supports application portability, mechanism flexibility, interoperability, and scalability. The software that constitutes the middleware on one host will be called **middleware component**.

2.4.1 Middleware Architecture

We use the following terminology to refer to the different layers that comprise the middleware architecture:

Definition 2
In a layered software architecture, the layer that contains the middleware functionality is called the **middleware layer**. The layer above the middleware layer contains the application and is therefore called **application layer**, while all layers below the middleware are subsumed in the **underlying technology layer**. Underlying technology includes at least the operating system, the hardware, and a communications mechanism (if the system is distributed across a network). In addition, it can contain more specific technologies (e.g., security mechanisms).

Figure 2.2 illustrates the architectural position of these layers in relation to the object-oriented model described above.

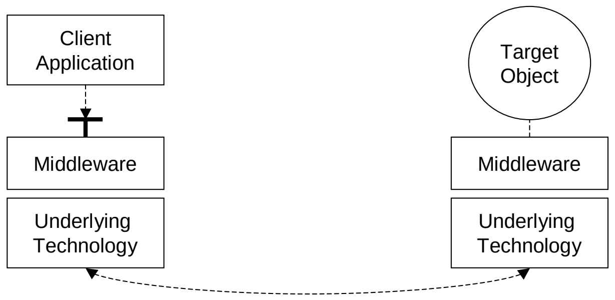

Figure 2.2: Middleware Architecture

2.4.2 Design Requirements

A middleware architecture should ideally meet a number of design requirements, which will be briefly introduced in this section.

Abstraction

The activities of the middleware component should be hidden from the applications, in particular related to communications: whenever a client invokes a target, the underlying middleware should automatically handle all lower-level communications tasks. For example, this often includes that the physical location of the invoked target is hidden from the client application (location transparency).

As far as this abstraction aspect is concerned, middleware can be compared to APIs, which are used in software engineering to present application programmers with a simple interface to the functionality provided by the lower layers. This requirement is related to portability, which is described next.

Portability

Specifics of the middleware component implementation and the underlying technology should be hidden from the application layer to allow applications to be ported from one middleware system to another (platform independence). This feature enables the easy re-use of existing software components for new applications. Portability is related to the interfaces between the application layer and the middleware layer.

Flexibility

Flexibility is the complement of portability and means that middleware should allow the use of different underlying technologies without affecting any of the layers above. In particular, the middleware itself should not be tied to a particular underlying technology. Therefore, flexibility is related to the interfaces between the underlying technology and the middleware layer, and allows the integration of clients and targets across differing hardware and software platforms.

Interoperability

Middleware should also abstract invocations from differences in the middleware implementation and the underlying communications technology (e.g., data representation). To achieve this design objective, middleware generally needs to use its own interoperability protocol and encoding. Interoperability is related to the communications interfaces between different middleware implementations.

Automation

Middleware should automatically intercept the normal invocation path, so that it can automatically add its services whenever an invocation occurs. This allows applications to be largely unaware of the underlying technology when they invoke other applications. From a security perspective, the fact that all invocations have to go through the middleware layer makes it a convenient location for security enforcement. This aspect is also often called “transparency” [119], but throughout this dissertation we will instead use the more descriptive term “automation”.

Scalability

Most middleware today needs to be designed to support large-scale distributed systems. Therefore the middleware should not introduce any additional constraints on naming or addressing of objects, so that the scalability of the addressing information is only limited by the underlying communications technology (e.g., TCP/IP). As a consequence, some middleware architectures (e.g., CORBA) do not introduce middleware specific unique (location independent) names for objects.

2.4.3 Architectural Interfaces

This section illustrates how the introduced design goals can be conceptually achieved by inserting interfaces between the different layers of the middleware architecture to separate it into individual modules. Vertical abstraction interfaces are inserted to insulate architectural components from one another, whereas horizontal interfaces are inserted between several whole architectural stacks.

Vertical interfaces (see figure 2.3) support portability and flexibility. Portability interfaces separate applications from the middleware and thus allow application code to be portable across differing middleware implementations (and underlying technology). The flexibility interfaces provide the middleware layer with mechanism-unspecific access to the underlying technology and thus allow underlying technology to be changed without affecting the middleware (or the application). In this sense, vertical abstraction interfaces are like APIs.

Horizontal interfaces (see figure 2.4) separate the middleware implementation and the underlying technology from the representation on the wire and thus enable interoperability. These interfaces normally involve middleware specific communications protocols that have their own data representation.

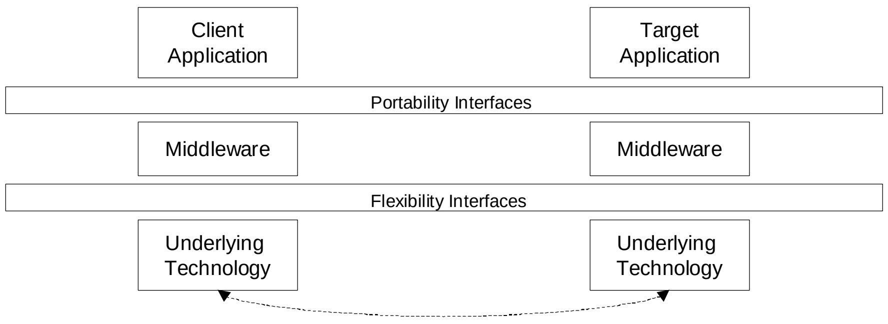

Figure 2.3: Vertical Abstraction Interfaces

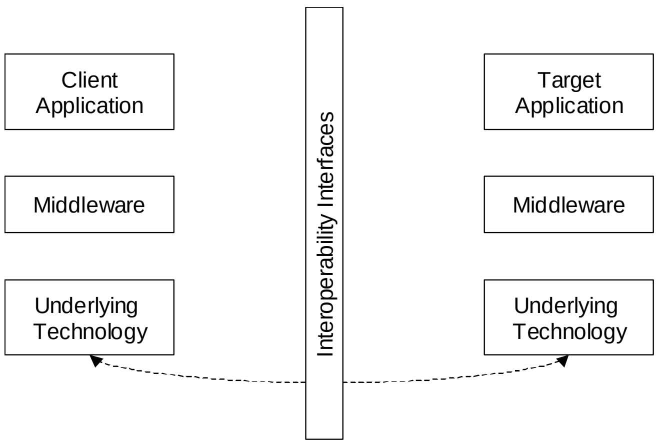

Figure 2.4: Horizontal Abstraction Interfaces

2.5 The Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) [119] was first published in 1990 by the Object Management Group (OMG), a non-profit organisation founded in 1989 to integrate distributed applications based on a variety of existing technologies. CORBA standardises interfaces and semantics for object-oriented middleware. It includes a specification for the Object Request Broker (ORB), a software library with standardised CORBA object interfaces that allows clients and targets to communicate with each other across a network in a well-defined way. In addition, CORBA automatically applies a range of useful services to communications. After the ORB is initialised, all CORBA objects can be invoked by applications just like local software objects.

2.6 Object Management Architecture (OMA)

The first key specification adopted by the OMG is the Object Management Architecture (OMA) [116]. It provides a complete architectural umbrella framework that puts all CORBA standards into context.

The OMA is composed of an object model and a reference model. The object model defines how objects distributed across heterogeneous environments can be described, while the reference model characterises interactions between those objects.

In the OMA object model, an object is an encapsulated entity with a distinct immutable object reference whose services can be accessed only through well-defined interfaces. Clients issue requests to CORBA objects that perform services on their behalf. The implementation and location of each object are hidden from the requesting client. Although the OMA implies that all participating software components are objects, it is only required that applications can support or use OMG-compliant interfaces to participate in the OMA, they need not themselves be constructed using the object-oriented paradigm. Existing non-object-oriented software can be embedded in objects called "object wrappers" that participate in the OMA.

The OMA reference model groups object interfaces in interface categories that are conceptually linked by an Object Request Broker (ORB). The ORB implements the communication infrastructure through which all CORBA compliant objects communicate. It is the middleware and mediates all communications between objects, and transparently activates those objects that are not running when they are invoked. There are also a number of services that can be linked into the ORB: the CORBA services, which define frequently used services, such as naming, trading, and security (see chaper 5); CORBA domains, which are application domain specific (i.e., vertical) facilities for domains such as financial services and healthcare; CORBA facilities, which are standards and services to be used horizontally across application domains (e.g., system management).

2.7 Object Request Broker (ORB)

The ORB, CORBA's middleware component, is defined in the Common Object Request Broker Architecture and Specification [119]. All OMG specifications define objects just

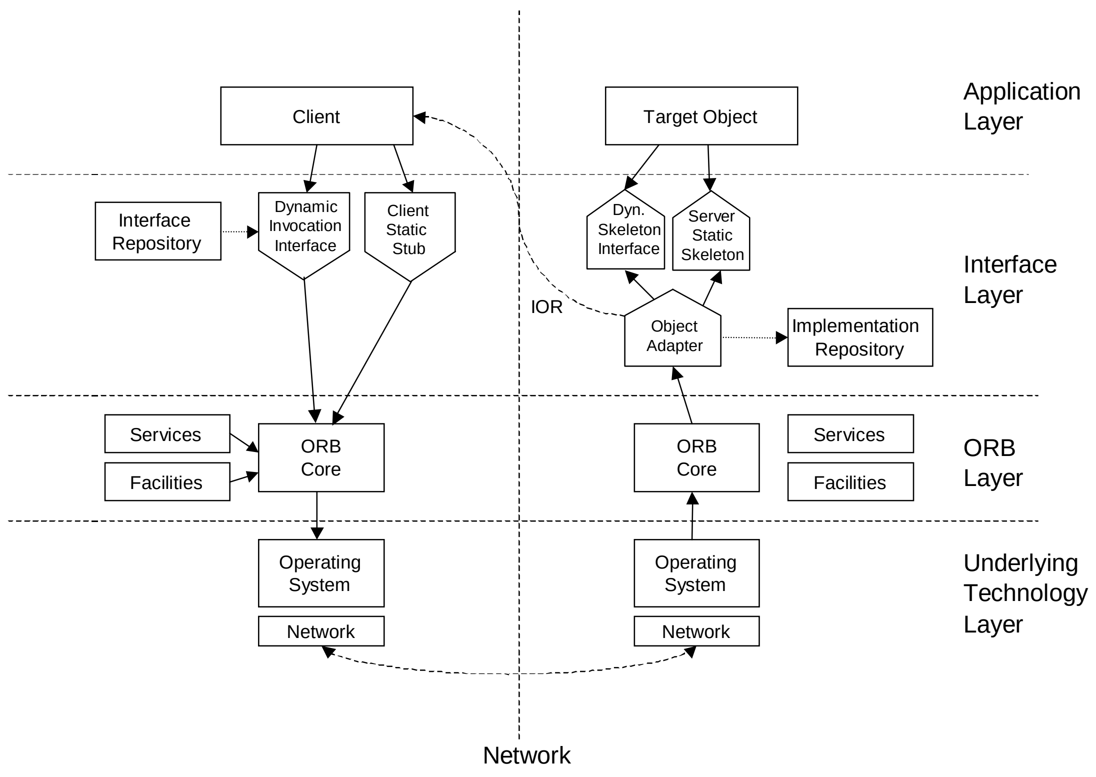

Figure 2.5: Common Object Request Broker Architecture

in terms of interfaces and their semantics, not in terms of their particular implementation. This allows CORBA vendors considerable flexibility in the design of their particular implementation, which is important because different environments often pose specific constraints and requirements on the ORB.

Most of the main features and functional components of CORBA are illustrated in figure 2.5, which also shows how the components relate to one another within the architectural framework outlined in section 2.4. The architecture contains an additional layer, the interface layer, which contains the stubs/skeletons (see sections 2.7.5 and 2.7.7) and the object adapter (see section 2.7.3). Amongst other purposes, this layer implements the abovementioned portability interface and connects the ORB with the application.

In the following subsections, the main parts of the CORBA architecture are described in more detail.

2.7.1 ORB Core

The Object Request Broker (ORB) is the middleware mechanism by which application objects make requests to – and receive responses from – each other, either on the same machine or across a network. The key feature of the ORB is the abstraction of communications between objects. The ORB hides the following information from the application layer:

- Object location: the client does not know where the target object physically resides.

- Object implementation: the client does not know how the target object is implemented, which programming or scripting language(s) it was written in, and the operating system and hardware it executes on.

- Object execution state: the client does not need to know whether the target object is currently activated (i.e., in an executing process) and ready to accept requests. If necessary, the ORB automatically starts the object before delivering the request to it.

- Object communication mechanism: the application does not need to know what communication mechanisms the ORB uses.

2.7.2 Object References

To make a request, the client specifies the target CORBA object by using an object reference that is automatically generated by the object adapter when a CORBA object is activated (i.e., a servant implementation is registered with the object adapter for a CORBA object, see section 2.7.3). Object references have similar semantics to C++ pointers, but can also address objects that reside on different processes or machines. Every object reference describes exactly one object instance, but on the other hand several references can denote the same object. References can also be `nil` (i.e., point nowhere) or dangle (i.e., point at deleted instances). Moreover, references are opaque (i.e., the client does not need to be able to interpret their content), strongly typed, support late binding, and can be either persistent or transient. Object references can also have standardised interoperable formats – such references are called Interoperable Object References (IORs).

Conceptually, an IOR contains three parts of information:

- A standardised *repository id*, a string that describes the "most derived interface" (MDI) type of the object at the time the IOR was created. This makes it possible to locate a detailed description of the corresponding interfaces in the interface repository (see section 2.7.6). The repository id does not always reliably identify the object that implements an operation – it is possible that a parent object actually implements the invoked operation.

- Standardised *endpoint information* that is used by the ORB to establish the connection to the server identified by the IOR. It contains protocol information and physical addressing information (e.g., TCP socket). This endpoint information does not always point to the CORBA server, it can instead point to an activation daemon (i.e., implementation repository) or a firewall.

- The ORB-proprietary *object key* is used at the target side to locate the object. Servers sometimes embed an application-specific object identifier, the so-called "ObjectId", inside the object key (see section 2.7.3) at object creation.

An IOR can contain the endpoint information and the ObjectKey several times. Such IORs are called "multicomponent profiles" and are mainly used for objects that should be accessible over several network protocols.

Object references can be obtained in three different ways:

- At object creation: a creation request returns an object reference for the newly created object to the client. CORBA has no special client operations for object creation – objects are generated by invoking creation requests, which are just ordinary operation invocations on other objects called factory objects.

- Through a directory service: a client can invoke a lookup service (e.g., naming service and trading service in the CORBA services) in order to obtain object references for existing objects.

- By converting references to strings and back: an object reference can be converted into a string and stored into a file or a database. Even after being "stringified" and "de-stringified", it can be used to make requests on the object as long as the object still exists.

2.7.3 Object Adapters

Architecturally, Object Adapters (OAs) serve as the interface between servants and the ORB. The term "servant" [61] refers to the programming language entity that implements one or more CORBA objects. Servants exist within the context of a server application ("a process implementing one or more operations on one or more objects" [121]), which bootstraps the ORB and the servant implementations. The term "CORBA object" refers to an IDL interface and an instance of an implementation.

An object adapter is an interposed piece of software, which provides a mapping that allows a caller to invoke requests on an object even though the caller does not know that object's true interface. Logically, object adapters link CORBA objects, which are described by IDL interfaces and object references, to the actual implementations of the object functionality in a normal programming language (i.e., servants).

Object adapters are responsible for creating object references. They also ensure that each target object is incarnated (i.e., implemented) by a servant, and pass requests from the target-side ORB to the servant. The functionality provided by an object adapter often also includes interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations. CORBA allows for multiple object adapters per ORB, addressing a wide range of object granularities, lifetimes, policies, and implementation styles.

CORBA without object adapters would mean that object implementations would need to connect themselves directly to the ORB to receive requests. Instead of different object adapters, a very complex ORB interface would be required, which would be difficult to standardize and would compromise the ability of CORBA to flexibly support diverse object and ORB implementations.

Portable Object Adapter (POA)

Since version 2.2, CORBA specifies the Portable Object Adapter (POA) that supports several ways of relating CORBA objects with servants implemented in different programming languages. At the same time, the application-facing POA interfaces have been

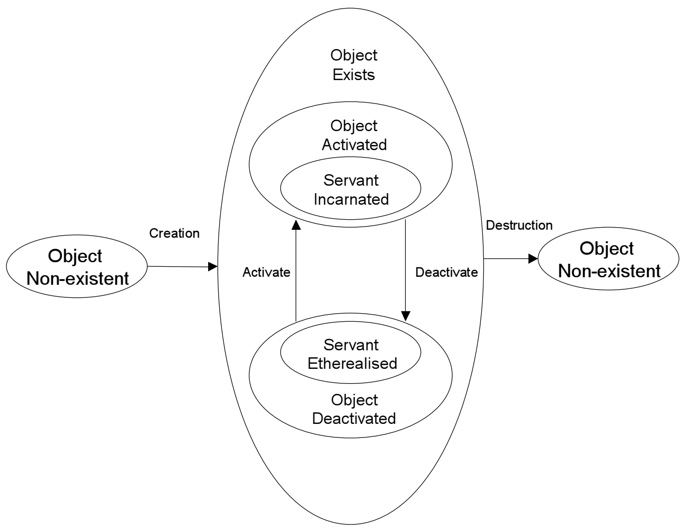

Figure 2.6: Lifecycle of Objects and Servants

designed to allow for application portability. A clear understanding of the POA is central to the proof-of-concept implementation outlined in chapter 6, therefore its specific features will be described in more detail.

First of all, it is important to understand the difference between CORBA objects and servants, and their lifecycles. CORBA objects and their references are created by the POA. They can be either "activated" or "deactivated", and only activated objects can service operations. In addition, only objects that are "incarnated" (i.e., implemented) by an actual servant instance are capable of receiving and carrying out requests (servants can also be "etherealised" to break the bond with its CORBA object). Figure 2.6 illustrates the difference between these concepts [61].

In the POA architecture, the lifecycles of objects and their servants do not have to correspond (i.e., the existence of an activated object and its reference does not mean that its servant is incarnated). It is possible that many objects share a single servant, or that one object is associated with several servants, or that the servant will only be associated with an object at the time the invocation occurs (default servants, see below).

During object creation, the POA is also responsible for:

- generating IORs
- generating ObjectIds, which uniquely identify an object within the scope of that POA. The ObjectId is a sequence of octets which can either be automatically created by the POA, or which can be supplied by the application. ObjectIds are part of the object key (see below) inside the IORs
- linking servants with objects and IORs. Applications can register servants with the POA and associate them with objects.

At invocation time, the POA maps invocations from objects to servants. Only the POA is able to relate IORs, ObjectIds, and servants. When a client invokes the object,

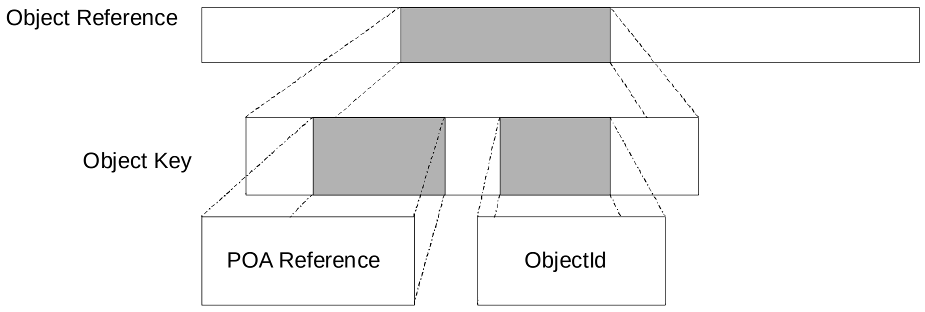

Figure 2.7: Object Key

the POA will pass the invocation (together with its arguments) on to the associated servant. To allow for good scalability, the POA servant manager allows the creation of servants on-the-fly when a request is received. In addition, it is possible to define a default servant, which will be executed for all objects within the scope of the POA that do not have their own servant implementation. For example, such a default servant can be used to implement a database interface, which can use the ObjectId as an index to a database. The default servant is executed for invocations on all objects within the scope of that POA and can use the ObjectId to query the database.

The object key, which is part of the IOR, contains information to locate the object within the scope of the ORB (i.e., after a network connection has been established between the client and the server). The object key is generated by the POA when the object is created, and it is passed as an opaque data item to the client within the IOR. CORBA does not specify the exact format and content of the object key to allow a range of different ORB and POA implementation styles. As a consequence, only the ORB and POA that generated the object key are able to interpret its content. In particular, the client is neither able nor required to understand the contents of the object key, it sends it back to the server unmodified as part of the request header.

To fulfil its purpose, the object key needs to contain two pieces of information (see figure 2.7). Firstly, it needs to include a POA reference that allows the target-side ORB to locate the POA in charge. In addition, it needs to include an ObjectId that allows the POA to locate the correct servant. We will discuss both in turn.

The POA architecture can either support persistent or transient objects (specified for all objects within the scope of the POA in the `PortableServer::LifeSpanPolicy`). Persistent objects should exist beyond the lifetime of the CORBA server application that implements the objects (e.g., static objects that permanently represent data in a database). In this case, the same IOR should always point to the same object. Transient objects on the other hand are tied to the lifetime of the server implementation, so their IOR should become invalid when the server terminates. Transient IORs are useful in cases where client and target share state, because the client can become faulty if the target crashes (i.e., loses its state) and restarts as a new instance without knowing that shared state.

The chosen lifespan policy has to affect the representation of the POA reference inside the object key. As mentioned above, the exact coding is implementation specific. MICO [138], the ORB used for MICOSec (see section 6.2), uses the following information for the POA reference:

- For transient objects, the reference includes the host IP address of the server, the process identifier of the server, the time when the reference was created, and a unique identification created by the ORB. For scalability reasons, CORBA does normally not provide globally unique identifiers for object adapters and objects.
- For persistent objects, it contains a string supplied by the user that associates the implementation repository with a server, and the full name of the server host

CORBA considers the POA reference as an implementation detail that should be hidden from the application programmer, and therefore does not provide any interface to it. However, the target side can query the POA for its "POA name", a string assigned to the POA during POA creation.

Once the ORB has located the POA associated with the request (using the POA reference from the object key), the POA itself needs to find the correct servant. This is done using the ObjectId, an unspecified data item that must be unique within the scope of its POA (but not globally unique for scalability reasons). The ObjectId is accessible from the application layer and can contain application specific information (e.g., a data base index). ObjectIds can be created either by the POA (`SYSTEM_ID`) or by the application (`USER_ID`), which can be specified in the `PortableServer::IdAssignmentPolicy`). Normally, ObjectIds supplied by the POA are used together with transient POA references, and application supplied ObjectIds with persistent POA references. The CORBA specification does not dictate the format for ObjectIds supplied by the POA (MICO uses the format "`_0`", "`_1`" etc.).

2.7.4 OMG Interface Definition Language (IDL)

The OMG Interface Definition Language (IDL) provides a standard way to define the interfaces to CORBA objects in a strongly typed language (similar in syntax to C++ classes and Java interfaces) that is programming language independent. It is a declarative language, not a programming language, so it forces interfaces to be defined separately from object implementations.

IDL language mappings provide the transition from the abstractions and concepts specified in CORBA to the real software implementation in the developer's programming language of choice (e.g., C, C++, COBOL, Java, Smalltalk, and Ada 95). In practice, language mappings are done automatically by an IDL compiler.

2.7.5 Static Invocation – Stubs and Skeletons

In addition to generating programming language specific types, IDL language compilers generate client-side "stubs" and server-side "skeletons" that form the basis for the actual implementation of clients and targets in the respective programming language. A stub is a mechanism that effectively creates and issues requests on behalf of a client, while a skeleton is a mechanism that delivers requests to the CORBA object implementation. A stub essentially represents the target object interface on the client-side. Dispatching through stubs and skeletons is often called static invocation. Stubs and skeletons are built directly into the client application and the object implementation, and therefore need to have complete a priori knowledge of the IDL interfaces of the objects being invoked.

A request sent by the client first has to be converted from the representation in the programming language to one that is suitable for transmission. Once the request arrives at the target object, the skeleton converts it to a (possibly different) representation (depending on the underlying hardware and software platform) and dispatches it to the object. The response is sent back the reverse way. Figure 2.5 shows the positions of the stub and skeleton in relation to the client application, the ORB, and the object implementation.

The general concept of stubs and skeletons is common to many middleware technologies, although the exact details often differ depending on the particular implementation and the used programming language.

2.7.6 Interface and Implementation Repositories

The interface repository provides persistent objects that represent the IDL information in a form available for look-up at run-time. Using the information in the interface repository, applications are able to determine what operations are valid on an object and to make an invocation on it, even if the interface was not known at compile-time. Using operations on the repository interface, applications can traverse an entire hierarchy of IDL information. Since the interface repository allows applications to programmatically discover type information at run-time, its real utility lies in its support of CORBA dynamic invocation (described in section 2.7.7). It can also be used as a source for generating static support code for applications (see section 2.7.5).

The implementation repository contains information that allows the ORB to locate and activate implementations of objects. Ordinarily, the installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the implementation repository. The implementation repository is also a common place to store additional information associated with implementations of ORB objects (e.g., debugging information).

2.7.7 Dynamic Invocation and Dispatch

CORBA supports two interfaces for dynamic invocation: the Dynamic Invocation Interface (DII), which supports dynamic client request invocation, and the Dynamic Skeleton Interface (DSI), which allows servers to be written without having skeletons for the objects compiled statically into the application ("dynamic dispatch"). The DII and DSI can be viewed as a generic stub and generic skeleton, respectively. Each is an interface provided directly by the ORB, and neither is dependent upon the particular IDL interfaces of the objects being invoked.

The DII supports three types of requests: synchronous invocation (after invoking, the client blocks waiting for the response), deferred synchronous invocation (the client invokes the request, continues processing, and later collects the response), and one-way invocation (there is no response).

While the DII offers more flexibility than static stubs, it has potential hidden costs. DII is often slower because the interface repository has to be queried for each object invocation, which often requires an additional request to a remote location. As opposed to that, static invocations do not suffer from the overhead of accessing the interface repository since they rely on type information already compiled into the application.

2.7.8 Inter-ORB-Protocols

The ORB interoperability architecture is based on the General Inter-ORB Protocol (GIOP), which specifies transfer syntax and a standard set of message formats for inter-ORB communication over any connection-oriented transport. The widely-used Internet Inter-ORB Protocol (IIOP) specifies how GIOP is built on top of a TCP/IP transport. The ORB interoperability architecture also provides for other Environment Specific Inter-ORB Protocols (ESIOPs). ESIOPs allow ORBs to be built for special situations in which certain distributed computing infrastructure is already in place. For example, the first ESIOP adopted was the DCE Common Inter-ORB Protocol (DCE-CIOP) that can be used by ORBs in environments where DCE is already installed.

Both the IIOP and DCE-CIOP have built-in mechanisms for implicitly transmitting context data that is associated with object services, such as transaction processing and security.

In addition to standard interoperability protocols, standard IOR transport profiles are necessary for ORB interoperability. For example, an IOR containing IIOP information stores hostname, TCP/IP port number, and other required information.

2.8 CORBA Run-Time Invocation

This section illustrates the circle of object creation and invocation at run-time. In particular, we will focus on the main feature of the middleware architecture, the vertical abstraction of applications from details of the underlying network (and object location). The application does not need to know any network details, it uses an opaque object reference to address the target object. In the following, we will go through the main steps involved in an object invocation (see figure 2.8).

Before an object can be invoked, it has to be registered with its object adapter. Depending on the particular implementation, the object adapter can create the servant instance ① or, to save resources, decide to instantiate the object only once it gets invoked. During the process of object registration, the object adapter creates an IOR, which encapsulates addressing details specific to the underlying technology, the target ORB and the object adapter. The client application can use this opaque reference without understanding any of the content – it is always used in the same way by the client (a bit like a pointer in object-oriented programming languages), no matter where the target object resides, which type of network is used, etc. In other words, applications do not need to be able to interpret the content of object references. Inside, the object reference contains mechanism specific details, which are used by the ORB and the underlying technology, such as addressing information about the target, including the hostname, port number, object adapter reference, and the object identifier (the so-called "ObjectId").

This IOR is then transferred to the client ②, either through a naming service or by other out-of-band means. Before the client can invoke the target, it has to supply the IOR to its ORB ③, which uses the addressing information to establish a network connection ("binding") to the target (if the target is remote) ④. When the client tries to invoke

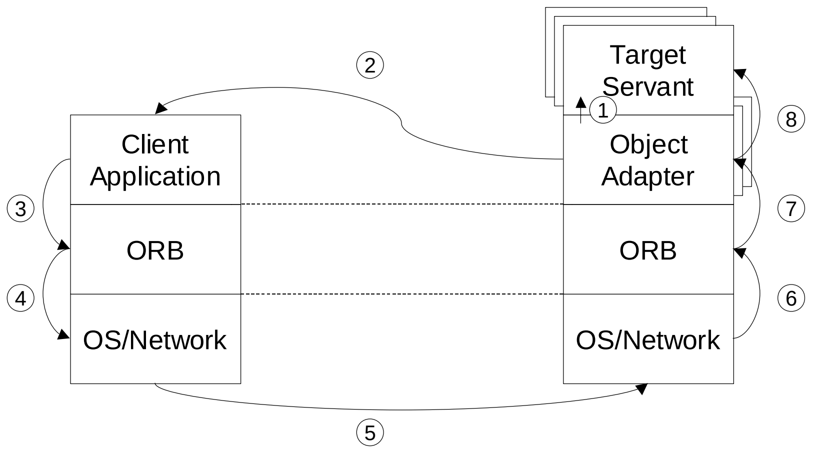

Figure 2.8: CORBA Invocation

the target, it invokes the corresponding stub. If the invocation goes across a network, the ORB packages the invocation parameters into a standardised CORBA request format and sends them across the network ⑤ to the target-side ORB ⑥, which unpackages the request. If the invocation is local, the ORB simply passes the invocation parameters back up to the correct target.

The target-side ORB then routes the invocation parameters up to the object adapter that matches with the addressing information ⑦. The object adapter in turn passes the operation parameters up (through the skeleton) to the object implementation (i.e., the servant) ⑧, which executes the request.

Optionally, a reply is sent back over the existing network connection if the client is remote. CORBA also supports asynchronous messages, which are called “callbacks”, but they are not relevant for our discussion.

2.9 Summary

This chapter introduces a notion of middleware that incorporates the object-oriented programming model and remote procedure call. In this dissertation, the term middleware describes software that resides between an application and the inner workings of the system hosting the application, and that abstracts the complexities of the underlying technology from the application layer. In particular, middleware automatically handles all communications related to invocations between client and target applications, and supports application portability, mechanism flexibility, interoperability, and scalability. The chapter outlines the main design requirements for a middleware architecture to satisfy the given definition. It was also shown how vertical and horizontal interfaces can be inserted into the layered middleware architecture to meet the design requirements.

The discussion of the CORBA architecture illustrated that messages pass through a number of layers on their way from client to target and back. In particular, the CORBA layer on the target side comprises an ORB, one or more object adapters, and the skeleton. At each layer, only parts of the message header can be interpreted (e.g., network socket, POA reference, ObjectId). Due to the cycle of object reference creation and object invocation, all addressing information is opaque at the application layer. In the next chapters, we will discuss the difficulties of achieving a similarly elegant solution when security is added to the middleware architecture.

Chapter 3

Towards a Middleware Security Model

3.1 Introduction

In the previous chapter, we have laid out a layered middleware architecture that meets a number of design requirements (see 2.4.2). Now we construct a simple security model based on principals and their identities that includes authentication and access control. We will first introduce the notion of a reference monitor that restricts access to resources. We then need to define precise terminology for our further discussion. We will introduce a notion of identity that allows the representation of active participants in the access policy. Then we describe the authentication process, and define what is meant by principal, client, and target. This is necessary because there is no consensus in the information security literature about the exact meaning of these terms. We then present a simple communications security model to illustrate how principals, identities, clients, and targets are related. After that, we discuss how a middleware security architecture (that provides authentication, message protection, access control, and audit) should be embedded into the layered middleware architecture such that it preserves the middleware design requirements. Finally, we will construct a first middleware security model that serves as a basis for the next chapter.

3.2 Terminology

3.2.1 Access Control

Most computer security literature describes access control in terms of a reference monitor by Lampson [83], which verifies that subjects are authorised to access the objects they request. In the CORBASec specification [121], the software component that contains the reference monitor (i.e. both access policy and enforcement) is called the access decision function. It provides a yes/no answer when queried either explicitly by the application or automatically as part of the invocation process.

Figure 3.1 shows the elements of the reference monitor model (it has been designed in the context of distributed operating systems [84]):

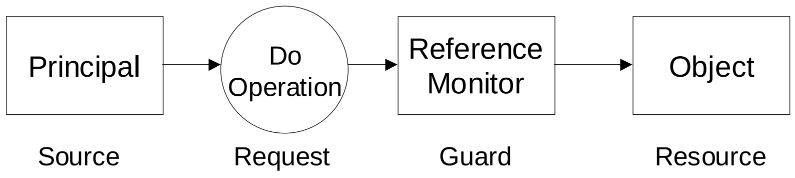

Figure 3.1: Access Control Model

- Principals are defined as the sources for requests
- Requests perform operations on objects
- A reference monitor acts as a guard for each object that examines each request for the object and decides whether to grant it
- Objects are resources such as files, devices, or processes

In the model, the reference monitor bases its access decision on the principal making the request, the operation in the request, and an access rule that controls which principals may perform that operation on the object. To do its work, the reference monitor needs a reliable way of knowing both the source of a request and the access rule. Obtaining the source of the request (i.e., the principal) is defined as authentication ("who said this?" [84]). The authentication process is conducted by the reference monitor. Authorisation is defined as the interpretation of the access rule ("who is trusted to access this?"), which is usually attached to the object. We will show later that it is non-trivial to associate access rules to individual objects in the context of middleware security.

There are also a number of alternative definitions of authorisation in the literature. For example, Ford [47] defines authorisation as the granting of rights, by the owner or controller of a resource, for others to access that resource. Access control is then defined as a means of enforcing authorisation.

Definition 3
Throughout this dissertation, the term **access control** will be used to refer to the evaluation and enforcement of access rules, while the term **access policy** refers to the access rules enforced by the reference monitor.

In line with the CORBASec specification [121], we will later on distinguish between client-side access policies and target-side access policies (see section 3.2.6).

Before we can describe more precisely what is meant by authentication, we need to define exact notions of identity and principal.

3.2.2 Identity

The meaning of identity is controversial throughout the security literature. Anderson [6] uses the term to describe a correspondence between the names of two principals signifying that they refer to the same person or equipment. The CORBASec specification [121] defines identity as a security attribute with the property of uniqueness (i.e., no two principals' identities may be identical). Other security attributes (e.g., groups, roles, etc.) need not be unique in CORBASec, and principals may have several different kinds of identities (each unique), for example for audit and access control. Ford [47] implicitly links the notion of identity to authentication: "authentication relates to a scenario where some party has presented a principal's identity and claims to be that principal". Therefore a principal has an identity that can be authenticated. We will define both principal and authentication below.

In this dissertation, we will define the term identity in relation to access control and authentication:

Definition 4
An **identity** is a name that is used in the access control policy to represent an entity in the system. In addition, it is possible to verify a claimed identity during the authentication process.

3.2.3 Principal

The term principal is defined in numerous differing ways throughout the security literature to describe the participants in a security model. Principals can be either human users or software entities, or both. To motivate our definition, we will first consider a number of alternative notions.

The CORBA Security Services (CORBASec) specification [121] defines principal as "the active entity in the system" and as "a user or programmatic entity with the ability to use the resources of a system". In this definition, it may not be clear why software components should also sometimes be principals, after all they are always started by a human user and therefore inherit the principal identity of the user. The reason is that objects in complex systems often need to be authenticated by their own separate identity, and not the identity of the administrator that started it.

Anderson defines that "a principal is an entity that participates in a security system" [6], but also aptly points out that this definition depends on what is meant by system. At the one end of the spectrum, a system could be a cryptographic protocol, and principals could be the cryptographic keys that define the communications channel. At the other end of the spectrum, the system could include the whole middleware system together with all applications, as well as all human users and administrators of the system. In this dissertation, the system can be defined to include the applications and all layers underneath. As part of the principal authentication process, human users authenticate themselves to the system and provide their identity to the corresponding application.

As mentioned earlier, Ford's definition [47] (in the context of communications security) implicitly links the notion of principal to authentication: "authentication relates to a scenario where some party has presented a principal's identity and claims to be that principal". Therefore a principal has an identity which can be authenticated.

To add to the confusion, the widely-quoted access control model by Lampson (see section 3.2.1) defines principals as sources for requests that perform operations on resources such as files, devices, or processes.

In this dissertation, we define:

Definition 5
A **principal** is a software entity that is verified by the authentication process as the legitimate holder of the corresponding identity (i.e., that resides on one end of the security association – see definition 7).

3.2.4 Authentication

Authentication is an important security function, because all other security functions (e.g., the reference monitor) depend upon it. Authentication can be described as the means of gaining confidence that people or things are who or what they claim to be (i.e., it gives assurance of identity [47]).

This process of verifying a claimed identity is often referred to as entity authentication to distinguish it from data origin authentication. For data origin authentication, an identity is presented along with a data item, and it is claimed that the data item originated from the principal identified.

The CORBASec specification [121] defines authentication as "the verification of a claimant's entitlement to use a claimed identity and/or privilege set". This verification process is based on some authentication information, which is "used to establish a claimant's entitlement to a claimed identity (a common example of authentication information is a password)". Depending on the kind of authentication, a claimant can be a user, an application, or a security system.

In this dissertation, we will distinguish two different types of authentication: principal authentication and peer authentication. Although the CORBASec specification uses the terms "principal authentication" (it even specifies an interface with that name) and "peer authentication", it does not provide explicit definitions.

Principal authentication allows human users (and application components) to authenticate themselves to the security system. As part of the process, the human user or application component supplies its identity together with the associated authentication information to the principal authenticator. The authenticator can then verify that the authentication information corresponds to the identity, which proves that the authentic user or application initiated the principal authentication process. As a result of the process, the security system can enable application components to act on behalf of the human user.

Definition 6
Principal authentication is the process that links the identity of a human user (or application component) to the middleware component that takes part in the peer authentication process. It involves the security system and a human user (or application component).

Peer authentication is the process of establishing an authenticated channel between two (local or networked) communicating entities, so that each communicating entity knows the authenticated identity of the other communicating entity. It involves (cryptographically) verifying whether the entity that should be authenticated knows a secret (e.g., cryptographic key) that is associated with its claimed identity.

In the CORBASec specification, this process is considered part of a "security association establishment" function, which also involves the creation of a (cryptographic) channel that protects invocations in terms of confidentiality and integrity. A security association is defined as the shared security state information which permits secure communication between two entities [121]. In other words, the security association on each side contains the authenticated identities of the repective other entity, as well as cryptographic keys used for message protection.

> **Definition 7**
> **Peer authentication** is the process between two communicating peers of associating the identity of the corresponding principal on the other side with a security association. **Security association** refers to the security state information that is shared between the peers, such as (cryptographic) identities and keys.

3.2.5 Client and Target

In (distributed) middleware systems, some parts of a software application initiate invocations, and other parts of a software application respond to those invocations. The CORBA specification [119] defines client as "the code or process that invokes an operation on an object", but it is sometimes also used to refer to human users or machines. We will reserve this term exclusively for parts of the application by defining:

> **Definition 8**
> **Clients** are the application software entities that initiate invocations.

In terms of access control, Lampson calls the receiving entity that is accessed through the request and protected by the reference monitor the "object", which could be resources such as files, devices, or processes [84]. Depending on scope and purpose, the CORBA specification [119] calls the receiving complement of the client either object, servant, or server (see 2.7.3).

We will try to avoid these terms to prevent overloading of terminology with the object-oriented programming model, and instead use the name "target" (and later also "resource"). The CORBASec specification only defines the term "target" in the context of privilege delegation (i.e., passing on privileges, so that intermediate principals can act on behalf of the originator): a target is defined as the final recipient in a delegation call chain (i.e., the only participant in such a call chain which is not the originator of a call) [121]. This definition is not relevant for our discussion, so our use of the word will not introduce any ambiguity.

In some of the secondary literature on CORBA (e.g., [61]), a "target object", within the context of a CORBA request invocation, is defined as the CORBA object that is the target of that request. We will use a similar definition, but also include Lampson's access control aspect:

> **Definition 9**
> **Targets** are the application software entities that respond to invocations (i.e., object implementations), and that are protected by the reference monitor.

3.2.6 Client-Side and Target-Side Access Policy

In line with the CORBASec specification [121], we will now extend our previous definition of access policy (see section 3.2.1) to distinguish between client-side access policies and target-side access policies (in CORBASec, one or both of these may not be implemented):

> **Definition 10**
> **Client-side access policies** define the conditions that allow the client to invoke the specified operation on the target object. **Target-side access policies** define the conditions that allow the target to accept the invocation.

On the target-side, the access policy states who is allowed to access the services offered by the target. Consider the following example: a company database application that is accessed from a public network (e.g., the internet) contains sensitive information, such as design blueprints. Employees should be able to use client applications to retrieve and update information in two trusted company databases. In this scenario, the target-side access control needs to ensure that only authorised employees can retrieve and update the database to prevent competitors from accessing any sensitive information.

Client-side access control can be useful in cases where clients reveal sensitive information as part of an invocation. For example, if a client application should only be able to invoke a particular target (but no other "rogue" targets that masquerade as the real target), then a client-side access policy could control that. A client-side reference monitor would enforce this policy based on the identities established by the (peer) authentication process. In the previous database example, the client side needs to ensure that it only invokes the correct database targets to prevent sensitive information (e.g., new blueprints) from leaking from the client to any competitors (which could masquerade as the company database). In other words, the policy allows invocations to the target only if the target's identity matches with the identity of one of the trusted company databases. Client-side access policies could be imposed either by the client-side application itself or by the security administrator.

3.3 A Simple Communications Security Model

In this section, we will outline a first communications security model without a middleware layer to illustrate our terminology (see figure 3.2). The left side shows a client that invokes a target, displayed on the right side. Communications can be either local or across a network. At the bottom of the figure, an authentication mechanism (e.g., authentication protocol) authenticates the principals to the respective other side, so that the client side knows the identity of the target side and vice-versa. These identities are used to represent clients and targets in the access policies. In the top half, the client C and the target T are displayed, which contain the application logic. One of the primary properties of this security model is that C and T can be interpreted as the principals P_C and P_T corresponding to the respective identities I_C and I_T.

The client and target application each contain a reference monitor that enforces a client-side access policy and a target-side access policy. On both sides, the access policy

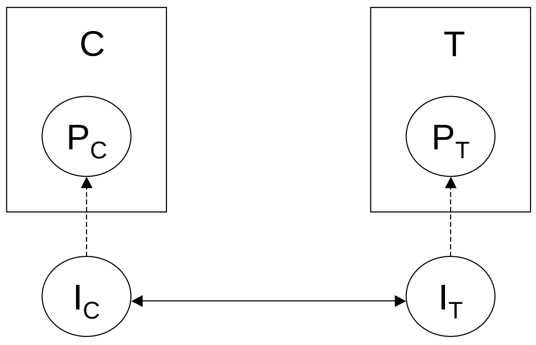

Figure 3.2: A Simple Security Model

Remote Identity	Access Decision
I_{T_1}	Grant
I_{T_2}	Grant
*	Deny

Table 3.1: Client-side Access Policy

is based on the identity of the respective other side. Table 3.1 shows an example of a simple client-side policy, where a client application is only allowed to invoke the target applications P_{T_1} and P_{T_2}, but no other targets. The target-side policy example displayed in table 3.2 only grants access to the target to P_{C_1} and P_{C_2}, while all other clients are rejected.

In line with CORBA's communications abstraction requirement, this generic security model covers both communications across the network and local communications (i.e., client and target reside on the same host, maybe even in the same process).

In the case of local communications, peer authentication could involve the operating system (e.g., transfer of signed tokens between processes). In the case of communications across a network, standard authentication mechanisms can be used to transfer identities (e.g., SSL or Kerberos v5). For the case where communications go across the network, no distinction is necessary in this model between authentication of communications network nodes and authentication of application software entities, because each network node corresponds to exactly one application entity. As a result, each client and target is uniquely identified by the identity established by the network authentication mechanism and can therefore be interpreted as the principal corresponding to that identity. We will discuss later on (in section 3.6) why this changes when a middleware layer is inserted between

Remote Identity	Access Decision
I_{C_1}	Grant
I_{C_2}	Grant
*	Deny

Table 3.2: Target-side Access Policy

the application layer and the communications authentication mechanism.

3.4 Design Requirements

One of the main difficulties of middleware security is to fit various security features into the layered middleware architecture in such a way that the design requirements outlined in section 2.4.2 are not compromised, and at the same time cater to the needs of users, application developers, and security administrators. In this section, we will discuss possible architectural design decisions resulting from each requirement, and the issues and trade-offs that arise. Because various differing design decisions can be deduced from the requirements, we will choose the option that is closest to the CORBA security architecture. The resulting layered security architecture will be presented in 3.5.

It is not the main purpose of this dissertation to analyse which design decisions best reflect the requirements, but rather to declare a sensible set of design decisions as given and analyse the consequences that arise for the middleware security architecture.

3.4.1 Abstraction

The middleware security model should separate the security system from the application logic (i.e., the security functionality should be hidden from the application layer). This is because the security architecture has to preserve the object-oriented programming paradigm at the application layer, but the object-oriented model as such does not make any provisions for the required security features.

Supporting abstraction has a number of advantages. Firstly, it allows "security-unaware" applications to be secured. This means that applications do not need to know anything about the security features, because all policies are enforced below the applications in the middleware layer. In particular, legacy applications can be automatically secured after they have been developed (i.e., without modifications to the code), and application development can be segregated from the security policy, enforcement, and administration.

However, on the downside, abstraction can inhibit the usefulness of the security system, because appropriate policies can often not be written without in-depth knowledge about the application. Also, the supported policies are generally less flexible and expressive, because less relevant security information is available outside the application.

Abstraction includes the further aspect that middleware security should be as invisible as possible to the users of the system. In practice, this can be achieved through a single sign-on feature that allows users to authenticate themselves to the system once to establish credentials, which can then be reused for many invocations.

Middleware security cannot be totally abstracted from security administrators, because security administrators have to understand the security system to be able to specify appropriate security policies. However, abstraction allows for centralised policy administration for all application layer entities that reside on top of each middleware component. To fully achieve this goal, the middleware security system also needs to be architecturally separated from the underlying security technology (see 3.4.3).

3.4.2 Portability

To support porting of applications across differing middleware security implementations and underlying security technologies, all security features should be placed below the application layer. This matches the design decision chosen for the abstraction requirement.

Our definition does not include portability of security policies and enforcement (together with the application), unless the middleware component implementation to which the application is ported supports the same security functionality as the original middleware component implementation. If security features and policies should be ported together with the application, then it might be preferable to tie certain security features (e.g., access control) into the application logic – which would conflict with our portability requirement described above.

3.4.3 Flexibility

All security functionality that is not specific to middleware, such as (cryptographic) communications security mechanisms for authentication and message protection, should reside on the underlying security technology layer. This normally includes general purpose security mechanisms that provide (cryptographic) authentication and message protection (e.g., Kerberos v5 and SSL). General purpose functionality means that it can be used in the same way with middleware as with any other technologies, because it does not rely on many details of the invocation. For example, for SSL there is no difference between a web browser that securely connects to a web server, and a CORBA client that invokes a target object over SSL. Therefore, standard security mechanisms (such as SSL and Kerberos v5), which may already be in place between network nodes, can easily be reused for the middleware security architecture.

In contrast, access control and audit normally rely on additional information that is specific to middleware, such as the invoked target instance, or the particular operation invoked. This information is only available in the middleware layer, because most of the message content cannot be interpreted in the underlying technology layer. For example, in many middleware technologies (e.g., CORBA) it is not easy below the (target-side) middleware layer to find out which specific target instance is invoked, because the instance identifier has been chosen at random in the middleware layer. As a result, this middleware specific part of the security architecture should reside in the middleware layer. These features normally do not need to be replaceable because no standard security mechanisms are available to be reused to provide this middleware specific functionality.

The flexibility interface that separates the (specific) middleware security technology from the (unspecific) underlying security technology hides the exact details of the underlying security technology from the higher layers (e.g., GSS-API [1]). This allows the underlying security technology and policies to be replaced without affecting the higher layers, so that pre-existing security technology can be re-used without the need for new cryptosystems, logons, security attribute repositories, user registries, or policy databases.

Following from this discussion, we define:

Definition 11

In line with the terminology defined in section 2.4.1, we will use the term **underlying security technology** to denote all security functionality that is unspecific to the middleware architecture, in particular for (cryptographic) authentication and message protection. All security functionality that is specific to middleware (but optionally uses security attributes from the underlying security technology), is referred to as **middleware layer security technology**, in particular access control and auditing. The more inclusive term **middleware security** is used to denote security features at both layers.

Ideally, it would be desirable to keep not only evaluation and enforcement, but also the policies independent of any details of the underlying security technology. This would allow for flexible replacement of underlying security technology, as well as centralised policy administration across different security technologies. However, keeping policies completely technology independent is difficult to achieve in practice, because the security functionality that resides in the middleware layer (i.e., access control and audit) normally relies on the security attributes established by the authentication mechanism in the underlying technology layer. Moreover, it is hard to find a semantically correct abstracted representation of the security attributes supplied by the underlying security technology [87].

In general, the flexibility interface should support a variety of different security attribute types and features, depending on the level of protection required. For example, it should be possible to flexibly extend the standard security attributes of the model to reflect additional requirements.

3.4.4 Interoperability

To support interoperability, it should be possible for security administrators to provide a set of consistent security policies across heterogeneous systems where different vendors provide different middleware and security products. This way, organisations can implement distributed systems without vendor restrictions, and choose the most appropriate technology.

Secondly, application layer entities that reside on a security-enhanced middleware component should still be able to interoperate with application layer entities that do not have any security. Of course such communications will not be secured, and it depends on the particular security requirements of the application (on the security-enabled end) if this should be allowed or not. From an architectural design viewpoint, this requirement means that the security protocols have to be layered over the unsecured interoperability protocols, and that the security enforcement has to be integrated into the communications path in such a way that it can be switched on and off depending on the security policy for each invocation.

Finally, it would be useful to support semantically correct interoperability across systems that support different kinds of security policies and underlying security technologies (e.g., different authentication mechanisms). The advantage of this would be that an appropriate set of policies and security mechanisms could be chosen for each application environment without inhibiting interoperability. However, this can only be achieved if a converter is developed that has access to all cryptographic keys, which breaks end-to-end security, in particular peer authentication (see definition 7).

3.4.5 Automation

The middleware security architecture should be integrated into the invocation path in such a way that the security policy is automatically evaluated and enforced whenever an invocation occurs. The layers below the application are a convenient place to intercept all invocations, because all traffic has to go through them on the way from the application layer to the underlying network and vice-versa. Automation is somewhat related to abstraction.

3.4.6 Scalability

The security model should support systems of different size, ranging from small to very large. The described layering can help scalability (and efficiency), because only one single reference monitor is needed to protect a potentially large number of application layer entities.

Actual upper limits to the number of participants or policy entries should be purely implementation-specific (i.e., depend on the underlying security technology and the policy implementation).

In addition, the security architecture should provide the means to make administration of large-scale secure systems easier. To reduce the administrative overhead, individual clients (or human users) should be grouped into roles (or groups) with the same privileges. Analogously, targets which share the same security policy should be grouped into domains.

3.5 Layered Middleware Security Architecture

Following on from the previously described design requirements, this section summarises how the different security features should be integrated into the layered middleware architecture (see figure 3.3). To preserve abstraction, automation, and portability, all security features should reside below the application layer. For flexibility, all (middleware-unspecific) mechanisms for authentication and message protection should reside in the underlying technology layer. To preserve syntactic interoperability, the security architecture needs to enhance the interoperable middleware protocols such that the communications security features are supported.

The security models presented throughout this dissertation are only concerned with access control and authentication. Auditing and message protection are only included in figure 3.3 because they are part of the CORBASec architecture described in chapter 5. Message protection is not part of the model because it is normally provided automatically as part of the security association. The rationale behind not including auditing is that its use of security attributes is very similar to access contol.

3.6 Simple Middleware Security Model

In this section, we present a first middleware security model, which will be used as the basis for our further discussion. The previous sections argued that the middleware layer is a convenient location to place the reference monitor, therefore we insert a middleware

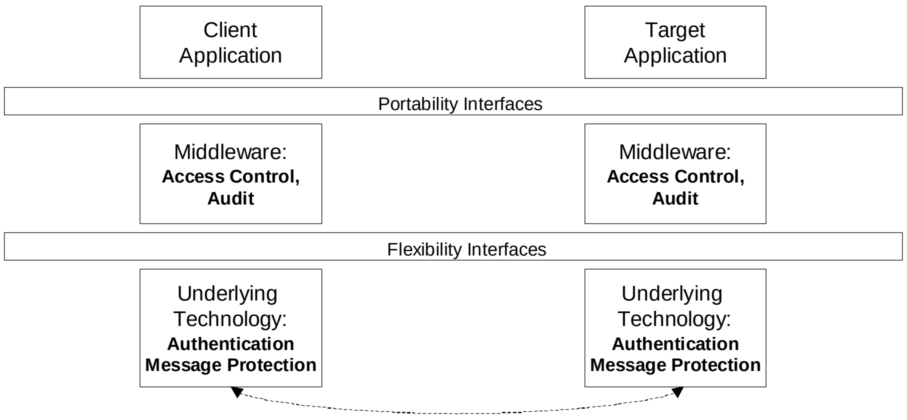
Client Application
Target Application
Portability Interfaces
Middleware: Access Control, Audit
Middleware: Access Control, Audit
Flexibility Interfaces
Underlying Technology: Authentication Message Protection
Underlying Technology: Authentication Message Protection

Figure 3.3: Middleware Security Architecture

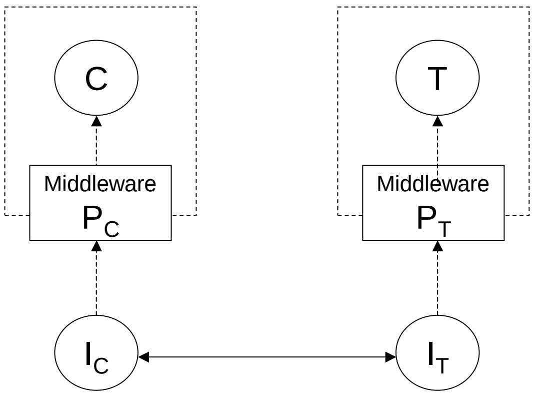

Figure 3.4: Simple Middleware Security Model

layer into the simple communications security model described in section 3.3. Like in the previous model, the reference monitor uses the underlying authentication mechanism to authenticate the identities I_C and I_T of the respective peer. The policy contains the same information as in the previous model. Conceptually, not much changes in this model, as there is only one application per reference monitor. As with the previous model, the client and target could again be viewed as the principals.

However, to set the scene for the next model, it needs to be made clear that the middleware components on both sides are the principals the identities correspond to. This is because the reference monitor in the middleware layer conducts the authentication, and therefore resides on both ends of the security association. Figure 3.4 illustrates this simple middleware security model.

3.7 Summary

In this chapter, we constructed a first layered middleware security model that forms the basis for the discussions in the following chapters. As a precondition, we first defined the terminology used to describe the model (access control, authentication, principal, identity, client, target).

In order to integrate with the middleware architecture, the model had to be designed in such a way that it preserves the architectural requirements outlined in section 2.4.2: for portability, abstraction, and automation (and scalability), all security features should reside below the application layer. Interoperability can be achieved with security enhanced middleware protocols and mechanism converters (e.g., for cryptographic keys).

For flexibility, all security functionality that is middleware unspecific, in particular (cryptographic) authentication and message protection, should reside in the underlying technology layer. The exact details of the used underlying security technology are hidden from the middleware layer by the flexibility interface. All security features that need to know more middleware specific information, in particular access control and auditing, should reside above the flexibility interface (i.e., in the middleware layer). This is because middleware specific information from the message header, such as the invoked target instance and operation, can only be interpreted at this layer. The middleware security technology can obtain security attributes from the underlying security technology through the (standardised) flexibility interface, which hides exact mechanism details. This architecture allows easy replacement and reuse of (middleware unspecific) underlying security technology.

In the next chapter, we will analyse the difficulties introduced by this layering and the fact that access control (and audit) rely on the identities established by the underlying security technology.

Chapter 4

Middleware Security Model and Resource Descriptor Mapping

4.1 Introduction

In the first part of this chapter, the simple middleware security model (see section 3.6) is refined step-by-step to capture a fine-grained access policy inside a reference monitor that resides in the middleware layer. In the model, the target-side reference monitor enforces an access control policy on incoming requests. The discussion in this chapter will show that the identities provided at the underlying technology layer (e.g., network communications authentication) are not fine-grained enough to capture useful access policies. To express individual application layer entities in access control policies, more fine-grained descriptors are necessary in addition to (cryptographic) identities.

The second part of this chapter describes the properties such descriptors should meet in order to be useful, and analyses the availability of descriptors for targets and clients in the middleware architecture. It will be pointed out that only local descriptors are trustworthy, and consequently remote descriptors will be removed from the model. Using the results of this analysis, a middleware security model is presented that is implementable in practice.

The third part of this chapter refines the semantics of descriptors by introducing a specific notion of resource that meets the identified properties, and shows how resource descriptors can be associated with target instances by using a mapping process called resource descriptor mapping (the proof-of-concept implementation of such a mapping process for CORBA will be described in chapter 6). For completeness, this chapter also includes a brief discussion of resource domains.

4.2 The N-to-1 Middleware Security Model

In the previous model (see section 3.6), there was only one client or target per middleware component. The model described in this section comes closer to a real-world middleware scenario in that there are now several clients and targets per middleware component (see figure 4.1). This is a particularly common case on the target side. For example, a bank server could contain a number of bank account objects, which contain the account data for different customers.

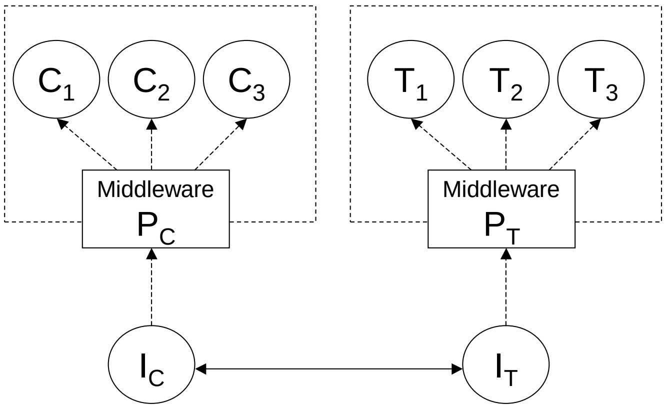

Figure 4.1: N-to-1 Middleware Security Model

In this model, there is still only one identity I_T per target middleware component, and thus all targets T_1, T_2, T_3 are part of the same principal. Again, the access policy resides in the middleware layer (as part of the reference monitor) and contains the same identities as in the previous model. But this time the semantics are different. In the previous models, the identities in the access policy represented the application principals because there was only one application per middleware component. Now the identity on each side represents the middleware component instead, and not individual clients or targets. In other words, the identity specified in the target-side access policy would now represent the client-side middleware component whose clients are all allowed to access (any of the) targets that reside above the target-side middleware component.

In the bank account example outlined above, the principal specified in the target-side access policy would represent the client-side middleware component whose client applications are allowed to access any bank accounts that reside above the middleware component on the bank server. In practice, this is not a problem as far as the client is concerned, as there is normally only one client application (e.g., the home-banking application) per middleware component. The problems surface on the target side, where several bank account objects of different customers reside. Only the authorised account owner should be allowed to access its account target, and it should not be allowed to access accounts of other customers. Such a policy cannot be expressed in this security model.

To conclude, access policies in this N-to-1 middleware security model are not expressive (i.e., fine-grained) enough for many real-world applications, because all clients and targets on top of each middleware component share the same identity as far as the reference monitor (which resides in the middleware layer) is concerned.

4.3 N-to-N Middleware Security Model

This security model attempts to get a more fine-grained notion of principals by assigning separate identities to each client or target, and thus fix the weaknesses of the N-to-1 middleware security model. In the model, the policies inside the reference monitor (which resides in the middleware layer) need to capture additional information about the

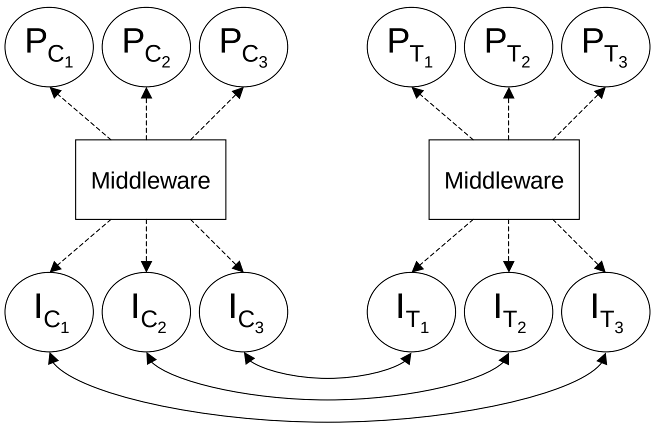

Figure 4.2: Idealistic N-to-N Middleware Security Model

Local Identity	Remote Identity	Access Decision
I_{C_1}	I_{T_1}	Grant
I_{C_2}	I_{T_2}	Grant
*	*	Deny

Table 4.1: Client-side Access Policy

principals that reside within the scope of this reference monitor. This is done by using a separate identity for each client or target application, so that the clients and targets become the principals. In other words, the policy inside the target-side reference monitor states the identity of the clients that are granted access, and the client-side policy states the targets which the clients is allowed to access. The reference monitor on each side would now be able to authenticate on a per-client (or per-target) basis (see example in figure 4.2).

Table 4.1 shows a simple client-side policy example, where the client P_{C_1} should be granted access to some target applications P_{T_1}, but not to any other target application on the same (or any other) middleware component. Similarly, another client P_{C_2} should be granted access to some target applications P_{T_2}, but not to any other target application on the same (or any other) middleware component.

On the target side we can describe an equivalent policy (see table 4.2). Some client P_{C_1} should be granted access to the target applications P_{T_1}, but not to any other target application on the same middleware component. Similarly, another client P_{C_2} should be granted access to some target applications P_{T_2}, but not to any other target application on the same middleware component.

Unfortunately, this model cannot be implemented within the context of the previously described architectural design (see section 2.4.2), where the authentication of identities is conducted by a reference monitor in the middleware layer. This is because the reference monitor in the middleware layer breaks end-to-end authentication between clients and targets: the reference monitor on one side of an invocation can only authenticate the middleware component underneath the target (or client) on the other side, and then it has to trust the remote middleware component to mediate the request or reply to the

Local Identity	Remote Identity	Access Decision
I_{T_1}	I_{C_1}	Grant
I_{T_2}	I_{C_2}	Grant
*	*	Deny

Table 4.2: Target-side Access Policy

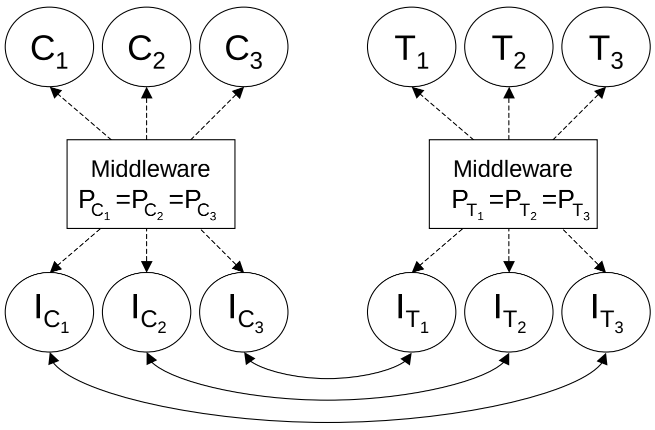

Figure 4.3: Realistic N-to-N Middleware Security Model

correct target (or client). Although separate identities have been assigned to individual targets (and clients), the principal that is authenticated is always only the middleware component below the targets (and clients). This problem could be solved if the reference monitor was shifted up into the application layer, but this approach would break most of the design requirements of the middleware security architecture (as described in 3.4)

Apart from this conceptual problem, such a model cannot be implemented well in practice. This is because a security association has to be established before the actual request is sent, so that the correct policies can be applied to the connection. However, this means that the receiving middleware would need to know which target is invoked before the request has been sent in order to be able to set up a security association with the correct key. One way of solving this "chicken-and-egg" problem involves assigning a separate network connection to each target. However, this is cumbersome to implement, in particular for CORBA, and does not scale well.

This discussion shows that in fact we have described a different model (illustrated in figure 4.3), where each authenticated identity represents the middleware component in that trust domain. So, on the target side, the identities I_{T_1}, I_{T_2}, and I_{T_3} represent the same middleware component (i.e., $P_{T_1} = P_{T_2} = P_{T_3}$), and on the client side, the identities I_{C_1}, I_{C_2}, and I_{C_3} represent the same middleware component (i.e., $P_{C_1} = P_{C_2} = P_{C_3}$).

To summarise, the middleware security architecture breaks end-to-end authentication, so that the use of separate identities for individual clients and targets cannot achieve client or target authentication. The only possible use for such different identities would be to represent the clients and targets locally inside the access policy. This would allow the association of individual access policies with each local target (or local client). However, the same functionality can be achieved with less effort by having one identity per mid-

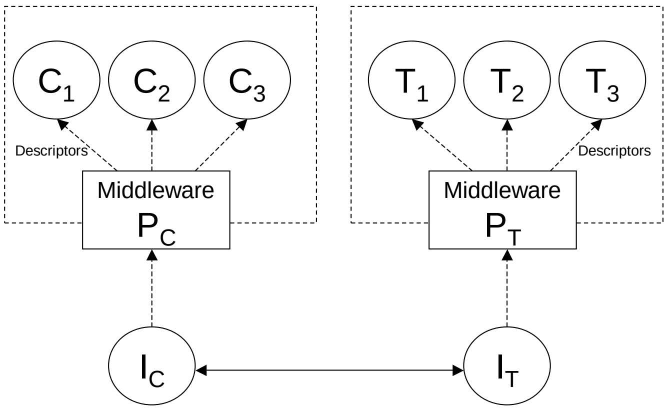

Figure 4.4: N-to-1 Middleware Security Model with Descriptors

dleware and simple descriptors to represent clients and targets inside the access policies. Such a model will be constructed next.

4.4 N-to-1 Middleware Security Model with Descriptors

This middleware security model tries to be as rich as the realistic N-to-N model, but only uses one identity per middleware component. To achieve that, clients and targets need to have names that are unique within the scope of each middleware component. Figure 4.4 illustrates this model.

As we have shown above, such names cannot be authenticated because all clients and targets that reside on top of one middleware component are in the same trust domain (i.e., are part of the same principal). Throughout this dissertation, we will call these names descriptors and represent them with the following notion in the access policy:

```
descriptor (<client/target>)
```

In this model, clients and targets can be uniquely identified (but not authenticated) through a compound that consists of the identity of the middleware component principal and the descriptor of the invoked target.

A client-side policy would look like this (see table 4.3): a client (not principal!) C_1 is allowed to access the target T_1 on the middleware component P_T (principal corresponding to identity I_T). In this model, the client-side reference monitor needs to know the descriptors for both client and target to express this policy.

A target-side policy (illustrated in table 4.4)) could similarly specify that a client (not principal!) C_1 that resides within the middleware component P_C (principal corresponding to identity I_C) is allowed to access the target T_1 on the local middleware component. In this model, the target-side reference monitor needs to know the descriptors for both client and target to express this policy.

Local Descriptor	Remote Identity and Descriptor	Access Decision
descriptor (C_1)	I_T+ descriptor (T_1)	Grant
descriptor (C_2)	I_T+ descriptor (T_2)	Grant
*	*	Deny

Table 4.3: Client-side Access Policy

Local Descriptor	Remote Identity and Descriptor	Access Decision
descriptor (T_1)	I_C+ descriptor (C_1)	Grant
descriptor (T_2)	I_C+ descriptor (C_2)	Grant
*	*	Deny

Table 4.4: Target-side Access Policy

This model has a number of weaknesses. Firstly, remote descriptors cannot be used to authenticate the remote client (or target), thus the underlying remote middleware component needs to be trusted to mediate the invocation to and from the client (or target) that is actually described by the descriptor. In addition, both sides need to obtain and administer the descriptors for the remote clients (or targets), which can become unmanageable if clients and targets are added and removed frequently.

4.5 N-to-1 Middleware Security Model with Local Descriptors

This model tries to take into account the difference between local and remote descriptors. Local descriptors are used to describe application layer entities within the same trust domain as the reference monitor, whereas remote descriptors describe application layer entities that reside within the trust domain of the remote reference monitor. The reference monitor can only trust local descriptors, because it can trust the local middleware component to associate the invocation with the correct client (or target). Remote descriptors cannot be trusted, because it is within the power of the remote middleware component principal to associate the invocation with any client (or target) within its trust domain. Also, it is unclear who would issue and distribute descriptors – if descriptors are generated by the middleware component when the client (or target) is started, then it is within the power of that middleware component to choose any descriptor it wishes, which could cause ambiguity and collisions.

Following from this, it becomes clear that the use of remote descriptors does not reliably add to the expressiveness of the access policy. Therefore we introduce the following model that only uses local descriptors: a client-side policy (see table 4.5) would specify that a client (not principal!) C_1 is allowed to access any target on the middleware component P_T (principal with the identity I_T). In this model, the client-side reference monitor needs to know the descriptor for the client to express this policy. A target-side policy

Local Descriptor	Remote Identity	Access Decision
descriptor (C_1)	I_T	Grant
descriptor (C_2)	I_T	Grant
*	*	Deny

Table 4.5: Client-side Access Policy

Local Descriptor	Remote Identity	Access Decision
descriptor (T_1)	I_C	Grant
descriptor (T_2)	I_C	Grant
*	*	Deny

Table 4.6: Target-side Access Policy

(see table 4.6) could similarly specify that any client that resides within the middleware component P_C (principal with the identity I_C) is allowed to access the target T_1 on the local middleware component. In this model, the target-side reference monitor needs to know the descriptor for the target to express this policy.

4.6 Descriptors

The type of middleware discussed throughout this dissertation tries to preserve the object-oriented programming model, which does not provide explicit unique names for clients and targets. In the object-oriented model, the target object instance is represented by a unique pointer (within the scope of the system), which can be used by the client to invoke operations on the target. Clients do not have any explicit representation in the object-oriented model, their only property is that they invoke operations. As a result of this, clients and targets also do not have explicit unique names within a middleware architecture that preserves the object-oriented programming model (e.g., CORBA).

We have explained in the previous section why descriptors are needed to express policies on a per-target (or per-client) granularity in the N-to-1 middleware security model. In the following, we will describe the properties descriptors should have. Then we will analyse which descriptor options are available within the middleware architecture to describe targets and clients.

Descriptors are necessary in the middleware security model because identities only represent middleware components, but not individual targets or clients. Descriptors describe individual targets and clients within the scope of one reference monitor (i.e., all targets and clients that reside on top of one middleware component). The purpose of descriptors is to express more fine-grained access policies inside the reference monitor.

To be useful, descriptors need to fulfil the following properties:

- Uniqueness:
  Descriptors should be unique within the scope of a middleware component principal and describe the targets or clients at a per-target (or per-client) granularity. In other words, it should be possible to associate a separate descriptor to each target (or

client) that will be unique within the scope of the underlying middleware component (we will show in section 4.10 that we need a more refined definition of target, which we will call "resource")

- Persistency:
  Descriptors should be "persistent" in the sense that the same descriptor should describe the same target (or client) independently from the object instance lifecycle. This property allows static policies to be expressed (in particular before the target object is instantiated). If the descriptor for the same target (or client) were to change frequently, then the policy inside the reference monitor would need to be updated each time.

- No Authentication Mechanism:
  Descriptors should not involve any authentication mechanism. There are a number of reasons for this:

  – in the middleware architecture described in previous chapters, the authentication mechanism and identities reside below the middleware layer (while the reference monitor resides in the middleware layer), so that the middleware component is always the authenticated principal. Therefore trying to authenticate descriptors does not achieve anything, instead it fosters the false assumption that the targets (and clients) could be the principals authenticated by the authentication mechanism.
  – there is no need for authentication of local descriptors because targets (or clients) and the reference monitor are in the same trust domain.
  – for remote descriptors, authentication does not achieve any additional trust on top of the authentication of the middleware component principal, which is already using the corresponding identity. This is because it is within the power of the remote middleware to relay the invocation to any of its targets (or clients)

- Coupling:
  Descriptors should be tied to the invocation in the following sense: if a descriptor is used by the target reference monitor to describe a target in the policy, then the addressing information that is used by the middleware mechanism to forward the request to the target should be the same as the information used to obtain the corresponding descriptor. In other words, it should not be possible for a malicious client to invoke one target, but tamper with the message in such a way that it points to a descriptor for another target. If this were possible, then a malicious client could select any access policy of its choice to be applied to the invocation.

4.7 Target Descriptor Options

In this section, we analyse whether the different options for target descriptors in CORBA-style middleware systems (see chapter 2) meet the descriptor properties. We will examine the interface type, the request header, the target instance identifier, and the target operation name.

Interface	Remote Identity	Access Decision
Bank	I_{C_1}	Grant
Account	I_{C_2}	Grant
*	*	Deny

Table 4.7: Interface-based Target-side Access Policy

4.7.1 Interface Type

The standard CORBASec access control model (see section 5.2.5) uses the target interface to describe the called target in the reference monitor. For example, if two targets reside on top of a middleware, and one is of type `Bank` and the other one of type `Account`, then the (simplified) policy illustrated in table 4.7 could specify who is allowed to access targets of the interface type `Bank` and who is allowed to access targets of the interface type `Account`. (for the sake of simplicity, this example assumes that there is only one client per middleware component).

The advantage of using the interface as a descriptor is that it is intuitive to understand and easy to administer. Also, the interface name is persistent in the sense defined above, and does not involve any authentication.

However, this approach has several severe drawbacks. Most importantly, the interface type does not always meet the uniqueness property because it is not expressive enough to uniquely describe targets if several targets of the same type reside on top of the same middleware component. In this case, a single policy applies to all of them. For example, if several `Account` targets reside on top of the same middleware component, then the target-side policy that each account should only be accessible by its respective account owner cannot be expressed (unless the target is invoked to query its state before an access decision is reached, which is generally not advisable). Another problem is related to the coupling of the descriptor specified in the request and the target that is actually invoked. In CORBA, the interface type is not part of the request header. And even if the interface name was supplied with the object reference, it could not be used as part of the addressing mechanism that forwards (based on instance information) the invocation to the target instance. Therefore it would be possible for malicious clients to supply any interface name with any (unrelated) instance addressing information, which allows it to pick any access rule for the target it wishes to invoke. In our example, it would be possible to replace the `Bank` interface name by the `Account` interface name in the request header that points to a `Bank` instance, so that the principal P_{C_2} (with identity I_{C_2}) could illegally access the `Bank` application.

There are two alternative ways of obtaining the interface name of the invoked target: firstly, an interface repository can be used that stores all mappings from local addressing information (taken from the request header) to the correct interface name. An entry can automatically be added to this mapping table whenever a new object gets instantiated. At invocation time, the reference monitor can then query the table for the interface name of the invoked target, and enforce the corresponding policy. However, communications to such a repository also need be protected (unless it is stored locally inside the target middleware component), which results in a "chicken-and-egg" situation. In the second solution, the reference monitor queries the target directly for its interface name. To make this work, all targets would need to export an operation that supplies their interface. However, this can fail if interface inheritance is used, because it depends not just on the invoked object, but on the individual invoked operation if the implementation is part of the target interface or if it is actually inherited from a parent class. In other words, it is not always possible to determine the exact "most derived interface" (MDI) of an invoked target operation [63].

This discussion illustrates that the interface name is not a suitable target descriptor because it does not always uniquely describe the target, and because it is not always coupled to the invocation mechanism.

4.7.2 Request Header

The request header is used by the middleware component to mediate the invocation to the correct target, and thus contains all the information to uniquely describe the target object instance. The client-side middleware component uses the object reference provided by the target-side middleware component to construct this request header. The addressing information includes the address used by the underlying transport mechanism, a target instance identifier (the ObjectId in CORBA) that is used inside the middleware component to locate the correct target object, and the name of the invoked operation. The target instance identifier is unique within the scope of the middleware component to make sure that the correct target is invoked.

Target Instance Identifier

The information that describes the target instance fulfils the uniqueness property, because it uniquely describes the target within the scope of the underlying middleware component. It does not involve any authentication mechanism. It is also coupled to the target addressing mechanism.

However, the target instance identifier does not fulfil the persistency property because object references are normally transient (i.e., limited to the lifetime of the target instance). CORBA also caters for persistent object references, but these should only be used in specific circumstances where applications are stateless or can handle lost state. There are various software engineering reasons for using transient references instead of persistent references, in particular to cater for the case where a target instance is shut down and re-instantiated. In this case, the state of the old instance is lost (unless it is made persistent). Clients that were still running sessions with the old instance should not continue their sessions with the new instance, because this can lead to undesired effects, such as errors or corrupted data in the client or the target. From a more theoretical perspective, the middleware model should also preserve the object-oriented programming model where pointers are transient, thus persistent ObjectIds should be avoided.

If the transient target instance identifier is used to describe the target in the reference monitor, then the lifetime of the corresponding access rule is also limited to the lifetime of the target instance, which can cause severe administrational problems if targets are shut down and re-instantiated frequently. Also, target instance identifiers are often not human-readable, which further complicates the administration task.

Consider the example where two printer targets reside on one middleware component, one colour printer and one black and white printer. The corresponding access rules (see

Local Instance	Remote Identity	Access Decision
1	I_{C_1}	Grant
2	I_{C_2}	Grant
*	*	Deny

Table 4.8: Instance-based Target-side Access Policy

Local Instance	Local Operation	Remote Identity	Access Decision
1	print	I_{C_1}	Grant
1	reset	I_{C_2}	Grant
*	*	*	Deny

Table 4.9: Instance-based Target-side Access Policy with Operation Name

table 4.8) should specify which client-side principals are allowed to access the colour printer (instance id=1), and which are allowed to access the black and white printer (instance id=2).

Now assume the colour printer jams and needs to be reset (and the corresponding target needs to be re-instantiated). To illustrate the need for transient object references, none of the clients that are using the printer at the time of the paper jam should continue with their print jobs. The new instance of the colour printer should therefore have the new instance id=3. However, there is no access rule for this new instance, thus the administrator needs to reconfigure the access policy to reflect the changes.

Target Operation Name

The request header also contains the name of the operation that should be invoked. In line with the object-oriented programming model where no two operations can have the same name within the same interface (unless polymorphism is supported), this name is unique within the scope of the target.

The use of the operation name in the policy allows the specification of access rules on a per-operation granularity. Table 4.9 shows an example printer policy where one client-side principal is allowed to print on the printer (instance id=1), while another client-side principal is allowed to reset the same printer.

Operation names are persistent in the sense that a target will always export the same operations, regardless if the instance changes. The establishment of operation names does not involve any authentication mechanism, and they are coupled to the invocation mechanism because the middleware addressing mechanism uses the same operation name to find out which operation is being invoked.

As mentioned above, operation names are not unique if polymorphism is supported. This means that several operations can have the same name, and the correct one is selected based on the number and type of the supplied parameters. In this case, the same policy will apply to all instances of the specified operation.

Remote identity	Access Decision
I_T	Grant
*	Deny

Table 4.10: Client-side Access Policy with Identities

4.8 Client Descriptor Options

The target can be explicitly described by its interface name or by its instance identifier, and optionally by the operation name for finer granularity. In contrast, clients do not have any explicit representation in the object-oriented programming model, in particular they are not defined by interfaces or pointers. Clients are only characterised by the fact that they use an object reference to initialise the underlying middleware component, and then invoke operations on a target.

Most information in the request header is used to describe the target, not the client. The only client-side information in the request header is a randomly chosen number (in CORBA this number is called RequestId) that is copied into the reply header. This number allows the client-side middleware component to match replies with requests. However this number only characterises an invocation and not a client.

In summary, there are no descriptors for clients available in the middleware architecture.

4.9 N-to-1 Middleware Security Model with Local Target Descriptors

The discussions in the previous sections have shown that there are some options for target descriptors, even though none of them fulfils all of the descriptor properties outlined above. We have also shown that there are no descriptors available for clients.

As a consequence, we have to remove the client descriptors from the N-to-1 middleware security model with local descriptors (see section 4.5). Also, target descriptors cannot be used on the client side because they are not local (i.e., cannot be trusted). Therefore the client-side policy (see table 4.10) can only express that any client within the scope of the client-side middleware component is allowed to access any of the targets within the scope of the target-side middleware component P_T (principal with the identity I_T).

The target-side policy (see table 4.11) can use the target descriptors because they are local: any client that resides on top of the middleware component P_C (principal with the identity I_C) is allowed to access the targets T_1 or T_2 on the local middleware component. In this model, the target-side reference monitor needs to know the descriptor for the target to express this policy.

Local descriptor	Remote identity	Access Decision
descriptor (T_1)	I_C	Grant
descriptor (T_2)	I_C	Grant
*	*	Deny

Table 4.11: Target-side Access Policy with Local Target Descriptors

Target Name	Remote Identity	Access Decision
Colour	I_{C_1}	Grant
B&W	I_{C_2}	Grant
*	*	Deny

Table 4.12: Target-side Access Policy with Resource Descriptors

4.10 Resources

The previous discussion about target descriptors has covered a number of options, in particular the interface type and the instance identifier. However, both do not fulfil all the descriptor properties identified above. The interface type does not uniquely identify targets and is not always coupled to the invocation mechanism, and the instance identifier is normally not persistent.

In the following sections, we will try to construct a descriptor that is based on the transient instance identifier, but that is also persistent (i.e., not bound to the lifetime of the target instance). Such a descriptor would fulfil all the required properties.

As a first step, we have to define what we would exactly like to protect. We do not want to protect a particular target instance as such, instead we want to protect the "service" offered to the caller. This service is of course provided by an instance, but it is irrelevant whether the instance gets terminated and a new instance takes over to provide the service (which implies that the object reference will change). In line with the previously described access control model [83], we will call such a service "resource" and the corresponding descriptor "resource descriptor":

Definition 12
A **resource** is a service offered to a client by a target instance. The instance that provides a resource can change over time, but it will always be protected by the same access rule. A **resource descriptor** describes (inside the access policy) a resource that is provided by a target instance.

In the previously described printer example, we would like a persistent (i.e., unchanging) name that is associated with each printer resource. For example, the colour printer should be called "Colour", while the black and white printer should be called "B&W". Both printers are of the same interface type. These names should describe the respective printer, regardless of the target instance identifier and the interface type of the target. The exemplary policy is illustrated in table 4.12.

Resource descriptors fulfil all the properties required for descriptors. They do not involve any authentication mechanism. They are coupled with the invocation addressing mechanism, because they are based on the instance identifier. They can uniquely identify the target, because each target instance can be associated with a separate resource descriptor (we will show later that resource descriptors can also be shared). In particular, resource descriptors fulfil the persistency property, because the administrator will provide the same resource descriptor each time the same resource is instantiated. Therefore, the policy can remain unchanged even if the instance that provides the resource gets shut down and re-instantiated (and the object reference changes).

An additional practical advantage of using such resource descriptors over using target instance identifiers or interface names is that they are intuitive to understand and easy to administer.

4.11 Resource Descriptor Mapping

Unfortunately neither the object-oriented programming model nor the middleware model directly provide any descriptors for resources. As a consequence, we have to design a mapping process that establishes resource descriptors that meet all the properties outlined in the previous chapter. In this section, we describe the mapping configuration and the mapping process at invocation time.

4.11.1 Mapping Configuration

Figure 4.5 illustrates the main steps involved in the configuration of the resource descriptor mapping. Firstly, a resource descriptor needs to be provided to the mapping process (e.g., manually as an additional parameter when the target that provides the resource is instantiated) ①. Only the human user or software component that initiates the instantiation knows for certain which resource it is starting (e.g., a colour printer or a black and white printer) and can therefore select the correct resource descriptor.

When the target is instantiated, the middleware chooses a random target instance identifier, which is unique within the scope of the target ②, and instantiates (or registers) the object. This identifier will be placed into the object reference ③, which in turn will be used by the client-side middleware component to construct the request header. To match the target instance identifier and the resource descriptor, the middleware component also has to store the link between the target instance identifier and the resource descriptor ④ in a mapping table that is kept within the middleware component.

4.11.2 Invocation Mapping

Figure 4.6 illustrates the basic steps involved in the mapping process at the time the invocation occurs. Before the client can invoke the target, it first needs to obtain the object reference (which includes the target instance identifier) for the target. This can be done through a naming service or by other means (e.g., manual copying). The client then provides this object reference to the underlying middleware component ①. At invocation time, the client-side middleware component constructs a request header from the information provided in the object reference and sends the request to the target side ②.

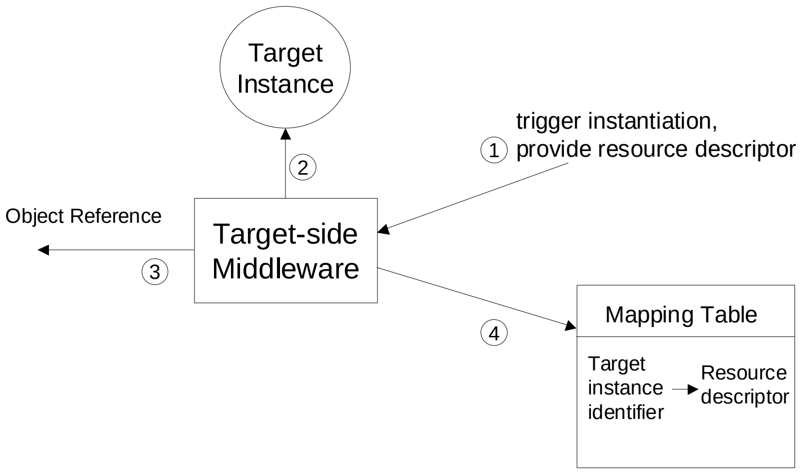

Figure 4.5: Configuration of Resource Descriptors

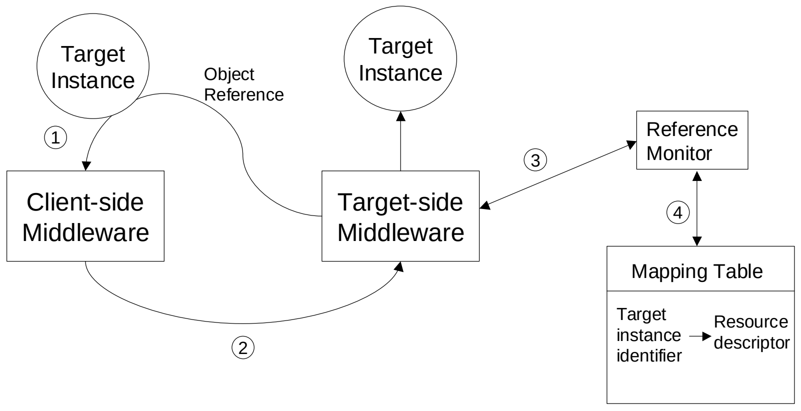

Figure 4.6: Invocation Mapping

The request contains the target instance identifier, which will be used by the target-side middleware component to locate the correct target.

When the request arrives at the target-side middleware component, the target instance identifier is extracted from the request header and passed to the reference monitor (optionally together with the name of the invoked operation) ③. Next, the target instance identifier needs to be mapped back to the resource descriptor, because the policy inside the reference monitor is based on resource descriptors ④. Finally, the resource descriptor is passed back to the reference monitor, which then evaluates and enforces the access rule for the corresponding resource. Depending on the outcome, the result is either rejected or passed up to the target instance.

4.12 Resource Domains

The mapping described above associates a separate resource descriptor with each resource. This works well as long as the number of resources remains small, but once the number gets larger, it would be preferable to group several resources into domains. We will call such domains “resource domains”, and the resources in the resource domain are called “resource domain members”. The main property of resource domains is that the same access rule applies to invocations to all of its members.

In the following, we will describe two simple ways of adding domain support to the mapping described above.

4.12.1 Simple Resource Domains

Although resource descriptors were chosen such that they fulfil the uniqueness property, it may be preferable for scalability reasons to use the same resource descriptor for several resources. This way, the resources are automatically grouped into resource domains. In this sense, the previously described resource descriptors can be viewed as domains with only a single domain member. The access rule associated with the descriptor applies to all invocations to the resource domain members.

Consider the previously described printer example, but now with four printer resources, two colour printers, and two black and white printers. The colour printers and the black and white printers should each be clustered into one domain. The resource descriptor “Colour” could be given to both colour printers, while the resource descriptor “B&W” could be given to both black and white printers. The access rules in the reference monitor can remain unchanged, the resource descriptors automatically become resource domain names.

This simple domain model has several advantages: firstly, the previously described mapping configuration and invocation-time mapping can remain unchanged. Secondly, it is easy to use and understand. All the administrator has to do is to supply a domain name whenever a resource is instantiated.

The disadvantage of the model is that resources can only be in a single domain (i.e., the described simple resource domain model does not support the case where resources should be members of several domains at the same time).

4.12.2 Overlapping and Hierarchical Resource Domains

Domains which have one or more members in common are often called overlapping domains. For example, assume that one high quality colour printer should be accessible only by one subject with the identity I_{C_1}(e.g., the marketing department), while a black and white printer should only be accessible by another user with the identity I_{C_2} (e.g., the accounting department). To support this, the first printing resource will be associated with the domain name “Colour”, and the second printing resource will be associated with the domain name “B&W”. Now a third printer resource should also be available to both these user groups (i.e., that printer resource should be in both domains).

The only way of supporting this example within the simple domain model described above is by manually adding an access rule that is associated with a specific descriptor for the third resource name, and that contains the combination of the domains “Colour” and “B&W” (see table 4.13). However, this approach impedes scalability, in particular if resources are frequently added or removed, and if the domain topology is complex.

Instead, it would be preferable to simply supply both domain names “Colour” and

Local descriptor	Remote identity	Access Decision
Colour	I_{C_1}	Grant
B&W	I_{C_2}	Grant
Colour_B&W	I_{C_1} I_{C_2}	Grant
*	*	Deny

Table 4.13: Exemplary Target-side Access Policy with Resource Domains

"B&W" when the third printer is instantiated, so that the combination of both corresponding access rules applies. Alternatively, another descriptor could be assigned to each resource, and a separate domain table could then be configured to group resources into different domains. Our proof-of-concept implementation (see chapter 6) uses hierarchical domain names to group resources into domains, which could be elegantly implemented by reusing CORBA's object adapter hierarchy.

Combinations and topologies of resource domains are considered outside the scope of this dissertation and will therefore not be discussed any further.

4.13 Summary

This chapter shows that the identities supplied by the underlying technology layer are not expressive enough to describe useful access control policies. Instead, additional more fine-grained descriptors are needed, which have to meet a number of properties in order to be useful. However, no suitable descriptors are available in the middleware layer to describe the client, and although some options are available to describe the target, none fulfils all properties.

Using the results of this analysis, an improved middleware security model is presented that is implementable in practice, and that is based on "resource descriptors" that fulfil all required properties. Resource descriptors can be associated with target instances by using a mapping process called resource descriptor mapping. The proof-of-concept implementation of such a mapping process for CORBA and CORBASec (see chapter 5) will be described in chapter 6.

Chapter 5

CORBA Security Services

5.1 Introduction

This chapter tries to bridge the gap between the largely descriptive concepts described in the previous chapters and the real-world implementation described in chapter 6. Its first part will introduce the features of the CORBA Security Services (CORBASec) [121] that are relevant for our discussion, while the second part will examine whether or not its architecture and design meet the middleware security design requirements outlined in section 2.4.2.

5.2 CORBA Security Services (CORBASec)

The CORBA Security Services (CORBASec) [121] specification was first published in 1995 and subsequently went through several updates to mitigate a number of discovered architectural issues, in particular regarding interoperability and portability. In version 1.5, the SSL-Inter-ORB-Protocol (SSLIOP) was added to the specification to meet industry demand for SSL integration. The OMG is currently working on a number of additional security-related documents, most notably the Security Domain Management Membership (SDMM) [124] service revised submission (see section 6.3), and a final submission for Common Secure Interoperability v2 (CSIv2) [120].

In addition to the four principal goals of secure systems (confidentiality, integrity, accountability, and availability [66]), the CORBASec specification mentions the following further fuzzy requirements: simplicity, consistency, scalability, transparency, easy administration, easy implementation of applications, certification, assurance, mechanism independence, reusability of the existing security infrastructure, flexibility, and interoperability.

The CORBASec architecture provides applications with the following security features (also see [67]):

- Authentication:
  Principal authentication allows clients and targets to authenticate themselves to CORBASec to set their credentials. Peer authentication is concerned with the verification of the identity of the respective other party. In CORBASec, the peer

authentication feature relies on the underlying security technology, which causes a number of problems (see section 5.2.3).

- Delegation:
  Clients can authorise intermediate targets to use their identity or privileges, optionally with restrictions. This feature relies on the underlying security technology (SSL cannot support delegation, but the CSIv2 [120] protocol introduces an additional protocol layer that supports the use of delegation tokens). Delegation is not related to our discussion and will therefore not be described further.

- Message protection:
  Data in transit can be protected from integrity and confidentiality attacks. This feature also relies on the underlying security technology (e.g., SSL).

- Access control:
  Access to target application entities can be controlled. This feature is dependent on the security attributes established during authentication (and possibly also delegation). We will show later in chapter 6 how the resource descriptor mapping introduced in chapter 4 enhances the access control model to allow for more expressive policies.

- Audit:
  Security relevant events can be recorded into different audit stores. Audit policies state the conditions that have to be met to trigger the logging of events. Audit policies benefit from the increased expressiveness provided by the resource descriptor mapping in exactly the same way as access control policies, therefore it is not necessary for our discussion to describe auditing in addition to access control.

- Non-Repudiation (optional):
  The generation and verification of irrefutable evidence of actions [68] are not relevant for our discussion and will therefore not be described.

In line with the architecture described in section 3.5, CORBASec relies on underlying security technology (in particular for communications security), such as Kerberos v5 [80], SESAME [74], and SPKM, through an interface modelled after GSS-API [94]. SSL is also widely used as a basic security mechanism for CORBASec. Unfortunately it does not integrate well into the CORBA security architecture because it establishes a TCP/IP network connection as part of the SSL security context establishment. As a consequence, SSL has to be integrated into the ORB as an alternative transport mechanism, so that a security context is set up automatically whenever the ORB opens a new network connection.

The CORBA architecture interacts with CORBASec at various point in the invocation path through so-called "interceptor interfaces": request level interceptors are called before the request is marshalled by the ORB and are therefore a convenient location for access control and audit, while message level interceptors are called after marshalling (i.e., just before the message is sent over the network), and normally host the message protection functionality.

In addition to providing security features automatically below the application layer, CORBASec also comprises application-facing interfaces that allow applications to evaluate and enforce their own security policies and to configure or query the underlying middleware security features. However, these interfaces are used for application layer security and not for middleware layer security, and are therefore not of interest to our discussion.

In the following sections, only the security features that are relevant for our discussion are presented in more detail. Because some of the terminology used in the CORBASec specification is ambiguous, imprecise, and sometimes overloaded, we use the more precise definitions presented in chapter 3 (e.g., principal, identity, client, target, authentication) instead. This makes overloaded CORBASec terminology such as “subject” obsolete.

5.2.1 Security Policies and Domains

Security policies in CORBASec define the rules for authentication, secure invocation, privilege delegation, access control, audit, and non-repudiation [121] in the middleware layer (i.e., outside the application logic), which helps segregating security administration from application development.

The attribute contents of security policies are mostly specific to the used security technology, which often complicates cross-domain communication. To a certain extent, the specification tries to mitigate this problem by defining standard attributes (e.g., of access rights – see 5.2.4 for more details), but the fundamental clash with interoperability [88] remains if policies should be flexible enough to fit to a variety of different application scenarios.

To facilitate security management for large-scale distributed systems, CORBASec groups targets to which the same security policy applies into security policy domains. When an target is created, it automatically becomes a member of one or more domains, and therefore is subject to the security policies of those domains. A domain can have subdomains that reflect organisational subdivisions (e.g., departments), and security policy domains can be different for different security policy types (e.g., access control domain, audit domain). The ORB makes sure that the policies associated with a domain are automatically enforced on all invocations to targets in that particular domain. CORBASec’s `DomainManager` interface allows policy objects to be created, deleted and updated, and interfaces are provided to locate `DomainManager`.

CORBASec does not address the problems that arise when domain membership should be persistent (i.e., independent from the instance lifecycle). In particular, it is unclear how domain membership could be specified for a target before it has been instantiated (or registered). Furthermore, CORBASec does not explain how the security policy domain of an invoked target can be identified when the request arrives at the target-side ORB (before the target has been invoked). The resource descriptor mapping described in this dissertation addresses both these issues, and thus makes domain based access control and auditing possible.

5.2.2 Principal Authentication

Principal authentication bootstraps the entire security system, because most other features rely on the verified credentials that are created during the principal authentication process. For example, identities are transferred to the other party during peer authentication, access control is often based on the peer identity, and audit event logging often needs to record the initiator of the event to provide accountability.

Credentials and Privileges

The purpose of principal authentication is the generation of credentials, which contain information that describes the security attributes of a principal [121]. Attributes can be identities and privileges of the principal (or both). A privilege in CORBASec is a security attribute that, as opposed to an identity, does not need to be uniquely associated with a principal (e.g., groups, roles, clearances). The security attributes contained in the credentials express the characteristics of a principal, and thus form the basis of the system's policies governing that principal. Both clients and targets can have credentials.

Whenever the CORBA security system encounters a new and unknown principal, it automatically assigns a default credential to it, which contains no identity and only one privilege attribute "public". For such a principal, a default policy will be enforced. If more (non-public) privileges are required, then the principal has to authenticate itself to the security system. After verifying the claimed identity, the security system will assign additional attributes to it (based on the security policy).

Principal Authenticator

The central object of the authentication model is the `PrincipalAuthenticator`, which provides a method `authenticate` for users and application layer entities to authenticate themselves and to create their credentials (generally through a user login program). In addition to supplying its claimed identity and the associated authentication information (e.g., password or certificate), the caller can specify the authentication method to use (e.g., password validation), and the security mechanism (e.g., X.509 certificate).

Depending on the information provided, the `authenticate` method returns caller specific credentials (together with mechanism-specific data, and optional continuation data if authentication proceeds in several steps). The created `Credentials` object is placed into the `Current` or `SecurityManager` object so that it can be used during security session establishment (see 5.2.3).

Policy

The principal authentication policy specifies which identities and privileges are to be given to a principal based on the presented authentication information. This policy is not explicitly represented within CORBA security, it is rather implicitly enforced by the underlying authentication mechanism: based on the provided security information, the authentication mechanism behind the principal authenticator will decide which privileges to put into a principal's credentials object. The generated credentials often contain some authentication mechanism specific data (e.g., an X.509 certificate that describes the principal), which complicates central administration.

Enforcement

Principal authentication policies are enforced by the principal authenticator, and ultimately by the underlying authentication mechanism. The model only describes how the underlying authentication mechanisms are integrated into the security architecture. For example, if Kerberos is used, there is an authentication server which contains all authentication information and is trusted to authenticate principals correctly. If SSL is used, then there would be a trusted certification authority which signs cryptographic identity certificates.

5.2.3 Security Association Establishment

Before a client can securely invoke a target, the CORBA security system needs to associate its credentials with the communications context, and, if communications across the networks are involved, exchange credentials securely with the remote CORBA security system (defined as "peer authentication" in definition 7). This is done as part of security association establishment. The transferred credentials provide both sides with the information necessary for security enforcement (e.g., access control on the target side often depends on the identity of the caller). Establishing the security association can take several exchanges of messages containing security information (e.g., to handle mutual authentication or negotiation of security mechanisms). Figure 5.1 illustrates how principal authentication and security context establishment are related.

Most implementations of the CORBA security services use standard network authentication methods for peer authentication (e.g., SSL, Kerberos v5, SESAME). While this has advantages, such as a high degree of technology maturity and off-the-shelf availability, they introduce the major flaw in CORBASec that peer authentication can only be done if client and target communicate across a network. If client and target reside on the same ORB, the network does not get involved in the communications, which consequently renders the network authentication mechanism ineffective. Although rather dissatisfying workarounds are possible, such as using SSL for inter-process communications within one host, the basic problem remains: most CORBA security services implementations make an implicit jump from the peer authentication between software entities to the peer authentication between network entities.

Security Contexts Objects

For each security association, a pair of `SecurityContext` objects (one for the client, and one for the target) contain the shared state information that represents a security association, such as the credentials used, the target security name, and the session key.

Once an association exists, it can be used for many subsequent interactions. Security associations may be shared (e.g., a client invokes several target objects that have the same identity), and there can be more than one `SecurityContext` object between the same client and target (e.g., if a client uses different privileges for different invocations on the same object). During the lifetime of a security association, applications can check its validity (with the operation `is_valid`), and may be able to refresh it (with the operation `refresh`) if permitted.

The `SecurityContext` stores several different types of credentials:

- Own credentials: contain the identities and privileges of the local principal associated with the active context. Own credentials are generated on both sides during the principal authentication process (see section 5.2.2). Separating principal authentication from peer authentication supports single sign-on.

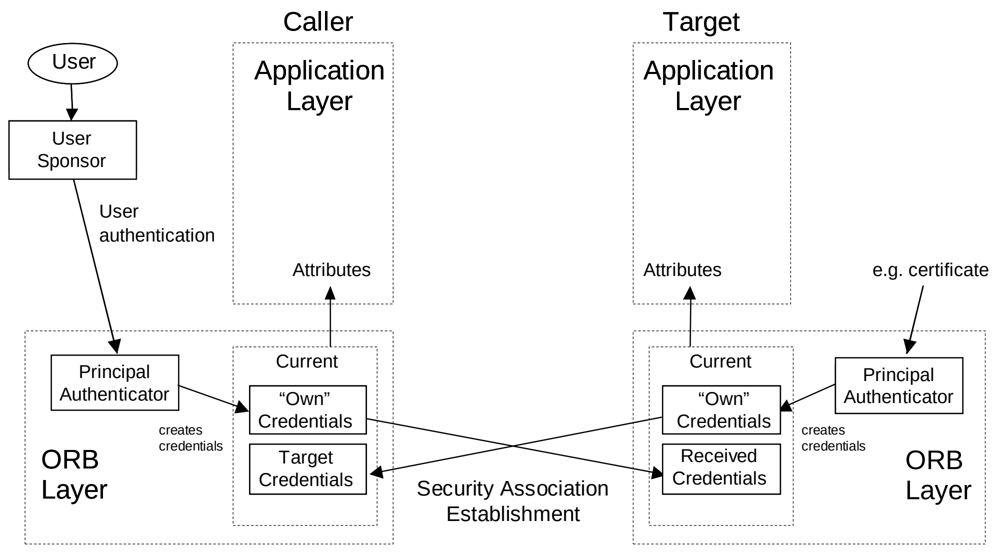

Figure 5.1: CORBASec Authentication and Security Assocation

- Received credentials: reside on the target side and contain the identities and privileges of the remote entity from which the execution context has most recently received a message (if it has received any).

- Invocation credentials: contain the identities and privileges the execution context will use the next time it sends a message. This is normally the same as the own credentials, but if the execution context has become a delegate, then the invocation credentials may be the same as the received credentials.

- Target credentials: reside on the client side and contain a remote principal's authentication information for the client's security association with the target.

Security Association Creation

Most of the actual association establishment work is done by the so-called `Vault` object. It is responsible for creating `SecurityContext` objects and for establishing the security association between client and target. The ORB interceptor calls `Vault::init_security_context` to request the security token that is to be sent to the target. The `Vault` then generates the client-side `SecurityContext` object and the token, which essentially contains the own credentials and some additional context information. Once the token is securely transferred to the target side, the ORB security service there calls `Vault::accept_security_context`, which generates the corresponding target-side `SecurityContext` object. The transferred credentials from the token are stored as the received credentials on the target side. The target's credentials on the client side are stored in the target credentials.

If the establishment of the security association involves mutual peer authentication or negotiation of security mechanisms, then several token exchanges may be necessary. These exchanges, like the `Vault` object itself and the `SecurityContext` objects it creates, are invisible to all applications. After such a handshake, both parties have access to a

`SecurityContext` object with all related credentials.

Context Access

The ORB architecture provides a standard way to access information associated with the active execution context. An application entity can find out what execution context it is in and what that context's credentials are by calling the ORB to get its so-called `Current` and `SecurityManager` objects and then querying them to obtain any of the credentials associated with the current execution context.

The CORBA security model associates security state information, including the credentials of the active principal, with the `Current` object. So, in essence the `Current` and `SecurityManager` objects provide access to security information, such as the credentials for the principals involved, from the active security association.

Interoperability

To support the described protocol exchanges necessary for security association establishment (and for message protection, see 5.2.4), the CORBA IIOP protocol requires a number of enhancements. To allow for replaceability, these enhancements are added as a separate Secure-Inter-ORB-Protocol (SECIOP), which is inserted on top of the IIOP protocol and transmits security information and GIOP messages securely across the network. Where possible, SECIOP messages are sent together with IIOP messages rather than as separate exchanges. However, this is not always possible, for example when a client wishes to authenticate the target before it is prepared to send an IIOP message.

SECIOP uses standardised security tokens to support the establishment of security associations. Although the types of security tokens are standardised (e.g., establish context, message context), the number of tokens used and the content of the tokens are mechanism specific. Token details for Kerberos, SESAME, and SPKM are specified for use with SECIOP, so that implementations which use the same mechanism (and consistent policies) can interoperate. If SSL should be used as a communications security mechanism, then the SSL-Inter-ORB-Protocol (SSLIOP) replaces SECIOP in the protocol stack.

5.2.4 Message Protection

Requests and replies need to be protected against unauthorised disclosure or modification while in transit between client and target.

Policy

The message protection policy specifies what Quality of Protection (QoP) needs to be applied to each message. The security model supports three different kinds of message protection (origin authentication, confidentiality, integrity), and the QoP policy defines which of them should be applied to a message, and at what strength.

The security model allows the definition of message protection policies in three places: system owners can specify client secure invocation policies and target secure invocation policies (on the ORB layer), which are attached to the `Current` object; target object owners can define application layer policies in the target's object reference, so that data flowing into and out of their targets is adequately protected; principals can specify in the principal's `Credential` object the minimum level of protection that has to be applied to all messages they send and receive.

Negotiation

To implement an appropriate level of message protection, the CORBA security system needs to combine the individual policies to arrive at the QoP that has to be applied to a message. The model supports a weak form of QoP negotiation between the communication parties: first, the client-side required QoP is collected from the principal's credentials, the object reference, and the `Current` object. Next, the target-side accepted QoP is established from the `Current` and `Credentials` object, as well as the sender's required QoP. In case of a match, the security service creates a `SecurityContext` object that implements the QoP.

Enforcement

Message protection is enforced automatically for all messages by the security system, which calls the operations `protect_message` and `reclaim_message` on the `SecurityContext` object. CORBA allows a choice of cryptographic algorithms for message protection. Furthermore, request and response may be protected differently, and both integrity and confidentiality protection can be applied to the same part of the message. In the CORBA security architecture, the underlying cryptographic mechanisms are not visible outside the security service – confidentiality and integrity protection are abstracted from the application layer.

Interoperability

The SECIOP and SSLIOP (see section 5.2.3) add cryptographic encapsulation of GIOP messages to the IIOP protocol. In addition, a security enhanced IOR format includes the authenticated identity of the target, relevant target-side security policy attributes, and the list of security mechanisms supported and required by the target (for QoP negotiation). Despite these standard protocols, secure interoperability can only be supported if ORBs share consistent security policies, and if the same security mechanisms are used on both sides.

5.2.5 Access Control

Access control (see section 3.2.1) in CORBASec is normally needed at the target side. Whenever the request arrives at the target side, the security system there needs to decide if the caller is authorised to invoke the target method. Client-side access control (see definition 10) is also supported and controls which requests client objects can dispatch.

Policy

In many environments, access control policies on a per-object granularity are not enough, as targets will frequently expose a number of operations with differing security needs. For example, an electronic banking object `Bank` could have a method `get_admin_contact`

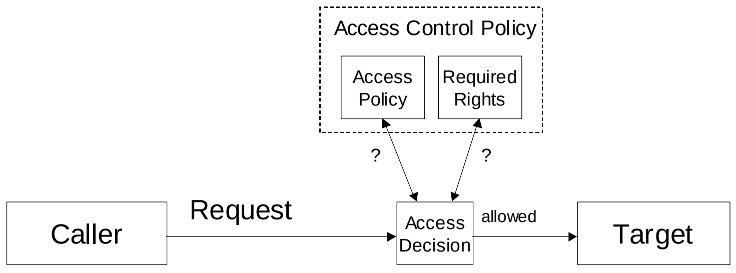

Figure 5.2: Access Control Evaluation

which is accessible to any user of the system, whereas a `withdraw` operation should be restricted to authorised customers.

Consequently, CORBASec supports access policies on a per-operation granularity. To solve the scalability problems of per-operation granularity, the model associates standard sensitivity levels to each operation (`g` (get), `s` (set), `u` (use), and `m` (manage)). This way, the policy can compare the level required to access the operation with the level granted to the client, and only allow access if the client's granted level is sufficient. Several access rights can be combined as a union (`any`) or intersection (`all`).

Implementations may define additional customised rights families to fit the model to their particular access control requirements. This way, the access control model can support a number of different policies, such as Access Control Lists (ACLs), Capability Lists, and Role Based Access Control (RBAC). However, the use of additional rights families impedes interoperability since the target side may not be able to interpret the caller's privileges correctly.

Evaluation

The access control evaluation functionality, which is encapsulated within the `AccessDecision` object, works as follows (see figure 5.2):

- Whenever a message arrives, the ORB security service intercepts it and passes it to the `AccessDecision` object to find out if the access is allowed.

- The `AccessDecision` object then forwards the caller's credentials (from the `Current` object) to `DomainAccessPolicy` (from the target's `DomainManager`), which returns the granted rights for the calling principal.

- `AccessDecision` then calls the `RequiredRights` object to find out the required rights for invoking the target method on the target object.

- `AccessDecision` is now able to compare the granted rights with the required rights. It will only allow the invocation if the granted rights match (at least) the required rights, otherwise the request will be blocked. Additional checks can be put in place at this point, such as controls that are applied to the privileges (e.g., lifetime).

The following IDL fragment shows the central operation on the `AccessDecision` interface, which bases its access decision on the following attributes:

- The list of credentials associated with the invocation, in particular the remote peer identity. This argument is of critical importance to the access decision and can be obtained easily from the security context.
- The object reference used to invoke the target. We have already shown in the previous chapters that it is not advisable to use the information from the object reference directly, because it will change frequently, and the policy will need to be updated every time the object reference changes.
- The name of the operation being invoked on the target. This attribute is useful for access decisions at a finer granularity, and it can be obtained easily from the request header.
- The name of the interface to which the operation being invoked belongs. This interface is called "most derived interface" (MDI) and will have to be supplied in cases in which the operation being invoked does not belong to the interface of which the target is a direct instance. As previously explained, there is a problem related to the fact that the MDI cannot be obtained reliably in all cases, which impedes its usefulness for access decisions.

```
boolean access_allowed(
    in SecurityLevel2::CredentialsList cred_list,
    in Object target,
    in CORBA::Identifier operation_name,
    in CORBA::Identifier target_interface_name
);
```

The `AccessDecision` object does not directly rely on any underlying security mechanisms for its evaluation, it deals with the standard (required and granted) access rights instead. The implementations of `DomainAccessPolicy` and `RequiredRights`, on the other hand, select access rights based on (mechanism-specific) credentials provided by the previously described security features.

This operation will be modified in chapter 6 to take into account the names of the domains of which the target is a member (i.e., the resource descriptor).

Enforcement

Enforcement is automatic because the security service intercepts all messages in the message path and mediates them through `AccessDecision`.

5.3 CORBASec and Middleware Security Design Requirements

In this section, we will briefly examine whether CORBASec meets the design requirements for a middleware security architecture as outlined in chapter 3.

The CORBASec architecture is complex for a number of reasons. Firstly, security enforcement in distributed object systems is inherently complex because systems are generally large, dynamic, and heterogeneous. Secondly, the architecture tries to cater for many application requirements with a one-fits-all architecture. Thirdly, abstraction in such systems is not easy to achieve and thus the architecture gains further complexity. Finally, administration of the CORBA security architecture is complicated because of its support for many different policy and mechanism types, and because there is often mutual distrust between the participants.

One of the main difficulties is that some goals conflict with others. In particular, there is a clash between interoperability and flexibility [88], because flexibility involves the customisation of functionality, whereas interoperability can only be accomplished through standardised functionality and protocols. For example, the flexible use of custom security attributes will inhibit interoperability, because they can only be understood by security implementations that support the same custom attributes. Another conflict along these lines is between flexibility and assurance – real assurance can only be certified if the system as a whole is looked at, so each time a component is changed the whole system needs to be re-evaluated. But flexibility in CORBA security means that components can be changed without affecting any parts on the layers above. It is clear that the security architecture can only try to find the best trade-off between such conflicting goals.

Discussions about the CORBASec specification also get complicated because some design goals depend on the actual implementation of the ORB and the CORBA security services product, in particular the assurance, performance, and scalability requirements. OMG specifications only specify object interfaces, but do not dictate how objects should be implemented. This way, OMG specifications allow for a wide range of possible implementations but, while being able to prevent major obstacles, cannot ultimately ensure performance and scalability for concrete implementations.

How secure the resulting system is also depends heavily on the effectiveness of the underlying security technology. CORBA security is only as strong as the weakest mechanism (i.e., the resulting system is vulnerable if there are any bugs in the underlying security technology, even if the CORBA security architecture has been correctly implemented).

We will now briefly examine potential shortcomings of the CORBA security model for each of the main middleware security design goals introduced in section 3.4.

5.3.1 Abstraction

From an application programmer's perspective, ORB layer security features can be (almost) transparently enforced. Security-unaware applications do not need to know the details about the underlying security technology, because all policies are handled outside the application on the middleware layer. From the user's perspective, the security architecture is almost completely hidden if single sign-on is implemented.

As expected, CORBA security is not transparent for security administrators. But this is not a problem because security administrators have to deeply understand the system to be able to specify appropriate policies. However, the fact that CORBA security cannot effectively abstract the content of security attributes from the underlying security technology [87] creates administrative problems. Central policy administration tools would need to be able to handle all kinds of (mechanism-specific) attribute content, and attribute content would need to be updated whenever security mechanisms at a node in the system are changed.

5.3.2 Portability

CORBASec can enforce security policies without any active involvement of the protected applications. Such security-unaware applications are totally segregated from the security architecture and are therefore portable across different security technologies. In particular, the user sponsor is often not integrated with the application, which separates the technology specific authentication information from the application.

Security-aware applications use standardised interfaces to interact with the security system (e.g., to authenticate themselves, or to set their access control policies). If such applications handle mechanism-specific data, such as identity certificates, then they are not portable across different security technology without reconfiguration.

CORBASec also tries to make security policies portable and consistent across different ORB and security services products by specifying standard attribute types, access rights families, and audit selector types. However, these policies often contain technology-specific security attributes, which prevents portability across differing security mechanisms.

5.3.3 Flexibility

The security architecture is specified to be independent of the underlying security technology, so that a variety of different underlying security technologies can be used. However, this abstraction from the underlying security technology only works from the application layer perspective – underlying security technology can be changed without affecting the application.

Within CORBASec, the replacement of underlying security technology requires the modification of the implementation of some technology-specific objects in the architecture. In particular, this applies to the principal authenticator object, which needs to process information that is specific to the used authentication method, and all objects that process identities (and privileges) inside credentials objects. This includes the delegation policy, the access policy, the required rights objects, and the audit policy object. In other words, the use of security technology specific security attributes ties the implementation of all other functional components to the particular authentication technology.

CORBASec's flexibility interface is based on the Generic Security Service API (GSS-API) [1], which allows a particular class of security mechanisms to be easily integrated (e.g., Kerberos or SESAME). However, the integration of SSL shows that not all security mechanisms can be easily integrated into the architecture. SSL has to be integrated into the ORB as an alternative transport layer and thus does not integrate with various core components of the model (e.g., the `Vault` object).

To allow for a variety of different security policy types and security features, CORBASec allows the flexible definition of additional security attributes, such as additional access rights. However, the use of such extended attributes will inhibit interoperability as they can only be understood by security implementations that can interpret the same extended attributes.

5.3.4 Interoperability

Interoperability clashes with flexibility [88]. By specifying common attribute types, access rights families, and audit selector types, the model tries to support consistent security policies across different ORB and security services products. To support interoperability, the Common Secure Interoperability (CSI) part of CORBASec specifies a common set of security mechanisms, as well as standard security-enhanced protocols and object references. On the other hand, these mechanisms (and on-the-wire tokens) have to be supported on all participating nodes and cannot be modified without changing them everywhere, and without also modifying the attribute content in policies. This inhibits flexibility and portability.

Part of the reasoning behind not abstracting the technology-specific attributes is that their exact nature often cannot be abstracted without semantic mismatches or granularity problems [86]. Interoperability between nodes with differing security attributes or differing underlying security mechanisms would require mechanism specific bridges. These implementation specific bridges, which are not part of the specification, need to be secured and trusted to map correctly. They are difficult to design and implement, and introduce the semantic (and granularity) problems mentioned above.

5.3.5 Automation

The CORBA architecture allows the integration with CORBASec at various points in the invocation path through interceptor interfaces. This architectural feature allows CORBASec to automatically enforce a security policy on every invocation.

However, proper non-repudiation requires the explicit consent of the user and therefore cannot be provided automatically. Principal authentication also often requires the interaction with the user or the application and cannot be done automatically, unless the implementation accepts the authentication information from the command line.

Automatic security enforcement by intercepting all traffic on the ORB message path (through interceptors) is a sound architectural feature, but its reliability depends to a large extent on the style and the quality of the actual implementation. Real assurance is difficult to achieve and certify for applications which are built on top of the CORBA security model, because a lot of components on different layers of the architecture play together to enforce security, and because there is no small trustworthy security kernel in the CORBA security model.

Also, the flexibility and portability goals, which encourage frequent replacement of different components, jeopardise the reliability of the system as a whole. This is because combining components that each fulfil security properties does not automatically imply that the resulting system fulfils the same security properties [179]. To solve this problem, it would be necessary to re-evaluate the whole system after each modification.

5.3.6 Scalability

The upper limit of users, objects, policy entries etc. depends on the actual implementation. From an administration perspective, CORBASec supports large-scale systems by using domains and roles (or groups, clearances, etc.). However, no tools are currently specified to manage them. As a result, it is currently not possible to manage CORBA systems in an interoperable way, which in itself limits scalability. In addition, management of cryptographic keys needs to be scalable, which depends largely on the particular PKI implementation and on the way it is integrated with CORBASec.

5.4 Summary

In this chapter, the main components of the CORBA Security Services (CORBASec) were introduced, and it was examined to which extent its design meets the middleware security design requirements. It turns out that CORBASec, although catering for security policy domains, does not state how domain membership can be defined persistently (e.g., before the target is instantiated), and how the domain of an invoked target can be identified by the target-side CORBA security system when a request arrives. The resource descriptor mapping described in chapter 4 solves both problems. A proof-of-concept implementation will be described in the next chapter.

Chapter 6

Proof-of-Concept Implementation

6.1 Introduction

This chapter elaborates on our proof-of-concept implementation of the resource descriptor mapping process (see chapter 4) for CORBA. The chapter will first give an overview of the OMG's upcoming Security Domain Membership Management (SDMM) service to illustrate the current effort within the OMG to standardise a similar mapping process, the so-called object-domain-mapper (ODM). After that, the main design decisions for MICOSec's ODM are presented, in particular the instance addressing information used for the mapping. Finally, both the (persistent and dynamic) mapping configuration and the run-time mapping for different granularities are illustrated using several short C++ code fragments, to show that the concepts described in this dissertation are usable in practice.

6.2 MICOSec

The proof of concept implementation was implemented as part of MICOSec[1] [104], a level 2 conformant Open Source implementation of the CORBA security services v1.7 [121]. MICOSec runs with MICO [103], a freely available and "CORBA compliant" (OpenGroup Open Brand for CORBA [128]) implementation of the CORBA 2.3 standard [118]. MICO only uses the standard Unix API (i.e., does not rely on proprietary or specialised libraries), and has a modular design even for implementation internals to ensure easy extensibility.

MICOSec uses the Secure Sockets Layer/Transport Layer Security (SSL/TLS) [32] protocol as its underlying security technology, which provides principal authentication, security association establishment (including peer authentication), and message protection. It supplies host granularity identities (X.509) to the middleware security technology. SSL/TLS was selected for the sole reason that it was already available as an alternative transport (SSL-Inter-ORB-Protocol, SSLIOP) for MICO. The implementation is based on the OpenSSL Open Source implementation.

According to the specification, only the standard CORBASec security attributes need to be mapped to the X.509 identity of the principal. However, to give clients and tar-

[1]The MICOSec project is not the sole product of the author's work, it also involved staff from ObjectSecurity Ltd, in particular Rudolf Schreiner.

gets more convenient access to the rest of the certificate content, it was decided to map other (largely non-conformant) security attributes to individual certificate items (e.g., organisational unit, certification authority).

SSL has several shortcomings that affect most of MICOSec's security features. We have previously illustrated the problems related to the insufficient granularity of identities established by the underlying security technology layer. In addition, SSL only supports target authentication or mutual authentication, but not client authentication. It also has the particular property that it establishes a TCP/IP session as part of the security association establishment process. As a result, SSL cannot support privilege delegation across intermediate nodes, because it can only establish a security association between two directly connected TCP sockets [89]. However, the OMG Common Secure Interoperability v2 (CSIv2) [120] protocol, which has also been implemented for MICOSec, adds support for client authentication and transfer of (privilege delegation) tokens.

Both access control and audit are implemented using request level interceptors, which are initialised when the application is launched. This way, the security features are automatically activated by the ORB whenever a message arrives, which allows access control and auditing for security-unaware applications. In addition, it is possible for security-aware applications to define their own application layer access control and audit policy. Audit events can be recorded into a flat file, UNIX `syslog`, or a relational database (e.g., the Open Source database `PostgreSQL` [161]). Both access and audit policies use the domain names provided by the object domain mapping described in this chapter to represent targets.

The current version of MICO runs the ORB together with all its servants in a single process (i.e., no thread libraries are used to segregate individual servants), which renders MICOSec's security features ineffective for local invocations. This is because servants can access the memory space of other servants or of MICOSec (e.g., to obtain security information of other servants, or to tweak MICOSec to bypass security enforcement, or to allow them to masquerade as someone else). However, the practical implications of this are unclear since CORBASec relies on the identities established by the underlying authentication mechanism (e.g., SSL in MICOSec), which cannot be used for invocations inside a process anyway. One possible solution would be to execute each servant together with its MICOSec instance inside a separate process (run under a separate operating system user id). In this case, SSL could be used for local communications between separate processes (i.e., MICOSec is effective), and servants would be segregated from each other by the operating system. And if MICOSec is linked as a shared library, the ORB would also be protected from the servants.

6.3 OMG Security Domain Membership Management Service

Since 1999, efforts have been made within the OMG to standardise a service that maps targets into domains. The current work is part of the upcoming Security Domain Membership Management (SDMM) [124] service, which is going through the final submission phase at the moment.

The submission has two main functional objectives:

- The first is to define interfaces for querying which security policy domain(s) an invoked target belongs to. The mechanism to provide this functionality is referred to as object-to-domain mapping (ODM). It is conceptually related to the resource descriptor mapping described in chapter 4 and will be discussed in more detail below.

- The second objective is to define interfaces for retrieving information about the state of invoked target ("object security attributes", OSAs), which can be used in the evaluation of various security policies (in particular access control).

The current submission also mentions the need for administration of both these functional components, but does not specify any administrative interfaces.

6.3.1 Design Requirements

The design of the described functionality should meet a number of architectural requirements, which will be briefly summarised. As with the rest of the CORBA security services, the ODM and OSA functionality should be supported transparently for security-unaware applications, but should also allow application-specific use within security-aware applications.

Also, the specified architecture should have minimal impact on the object adapter specifications (in particular the POA), so that it can be implemented independently of existing ORB products (e.g., as part of CORBA security products). Although the current submission only requires object adapters to receive and return standard policy objects, its ODM and OSA functionality is also used by security interceptors, which require support of the ORB and the object adapter.

In line with the previously defined coupling requirement (see chapter 4) for resource descriptors, the ODM should tie into the existing POA architecture and its internal addressing mechanism. In particular, the mapping should not be based on ODM specific information (such as a domain name) encapsulated into the IOR, because the request header is not integrity protected, so ODM information in the IOR cannot be trusted. Also, if ODM information is encapsulated into the IOR, then the IOR becomes obsolete as soon as the target's domain membership changes.

Although the POA is the most widely-used object adapter, and the SDMM architecture is designed to integrate well with POA based products (specific interfaces are specified for use with the POA), it should also be possible to use the SDMM architecture with other object adapters.

Also, the SDMM interfaces should allow for portability (i.e., it should be possible to replace one SDMM implementation with another without affecting the application, the security interceptors, or the security services implementation).

6.3.2 Object Security Attribute Retrieval (OSAR)

The first part of the submission describes a standard mechanism that allows ODM implementations (or other parts of the server) to retrieve security-related information, so-called object security attributes (OSAs), pertaining to a target. It specifies the syntax of the data structures representing OSAs, as well as the interfaces necessary for retrieving OSAs.

Although this part of the submission is not central to our discussion, the main elements of the architecture will be briefly summarised for completeness.

OSAs consist of two strings, an attribute type and a value, which can contain any information suitable for describing the security state of a target.

The central element of the OSAR architecture is the `OSAManager`, which stores OSA information for all targets under its authority. The specified `OSAManager` interface is object adapter unspecific and serves as a common ancestor for derived interfaces that are specific to particular object adapters. The object adapter specific derived interface contains an operation (`get_attribute_retriever`) that returns the (object adapter unspecific) `AttributeRetriever`, which allows the caller to retrieve OSAs associated with the target in question.

An `OSAManagerPolicy` (inside the object adapter) allows the security interceptor to obtain a reference to the instance of the `OSAManager` that contains the OSA information for the target within the scope of that object adapter.

Finally, a `Current` interface provides thread-specific operations for the following purposes: firstly, it allows applications to create instances of `OSAManagerPolicy`; secondly, it allows setting and obtaining a reference to an `AttributeRetriever` object (and other non thread-specific operations necessary for using OSAR mechanisms by security-aware applications).

6.3.3 Object Domain Mapping (ODM)

The main purpose of the ODM is to supply the target-side security services with information, which allows the selection of the policies (e.g., access control and audit) that should be applied to the invocation in order to protect the target. More specifically, the ODM should return the name of the policy domain(s) the target belongs to, based on the information extracted from the incoming invocation.

In line with the domains described in the CORBASec specification [121], domains are denoted and uniquely identified by their domain names. A domain name is defined as a structured name (i.e., a sequence of strings). The ODM submission allows a target to be a member of any number of such domains.

The ODM submission only covers the discovery of domain information for an invoked target, but not the administration of such information. Also, it is only concerned with domain membership of targets, but not with the relationship among domains. Domains are often arranged hierarchically (e.g., in MICOSec), but implementers are free to organise domains in any other relation.

Configuration

The main components of the ODM (i.e., the `ODMManager`, `ODMManagerPolicy`, and `ODMFactory`) are configured as follows: for each object adapter, there is an `ODMManager`, which holds the domain information for all targets within the scope of that object adapter. Within each object adapter, an `ODMManagerPolicy` interface holds a reference to that `ODMManager`.

For security-unaware applications, a default `ODMManagerPolicy` is assumed. Alternatively, security-aware applications can use an `ODMFactory` to create instances of the `ODMManagerPolicy`, which can be registered with the object adapter as part of the object adapter creation process.

Run-Time Mapping

At run-time, the security interceptor first has to obtain a reference to the `ODMManager-Policy` from the ORB. If the reference value is "nil", then the interceptor instead obtains a reference to the default `ODMManager`. If the reference is not "nil", then the interceptor narrows it to the object adapter specific interface. Once this is done, the interceptor can obtain a list of domains for the target in question (by invoking an operation with object adapter specific arguments) and feed them into the access decision function.

POA Mapping Information

The interfaces described so far are defined in an adapter unspecific way and therefore cannot be used with real-world applications. This is because the information used in the mapping depends on the design of the particular object adapter used. Instead, the purpose of these unspecific interfaces is to serve as a basis for deriving IDL interfaces that take into account the specifics of a particular object adapter implementation. These specific interfaces need to define the information that is to be used for the mapping.

The specification describes such object adapter specific interfaces for the POA. It defines that the mapping should be based on:

- A reference to the POA that serves the invoked target
- The ObjectId of the target for which the list of domains is required
- A pointer to the servant implementation behind the target for which the list of domains is required

The MICOSec implementation bases its mapping on different information (see section 6.5.1). In particular, the POA name is used instead of the POA reference.

ODM Access Decision

The CORBASec [121] specification defines a basic access decision operation that is not based on domain names (see section 5.2.5). The SDMM specifies an additional access decision operation that uses domain names instead of the object references. The new parameter includes a list of domains the target is a direct member of:

```
boolean get_authorization(
    in SecurityLevel2::CredentialsList cred_list,
    in ObjectDomainMapping::DomainNameList domains,
    in CORBA::Identifier operation_name,
    in CORBA::Identifier target_interface_name
);
```

An early version of MICOSec implemented domain based access control according to the CORBA security specification (see section 3.2.1), but during the practical evaluation a number of problems surfaced: firstly, it was unclear how the access control decision object could find the required rights associated with a target, as neither the IOR nor the interface type are suitable in practice to represent the target (see section 4.7). As a result, the domain name (obtained from the ODM, see section 6.5) had to be used for this purpose. In addition, it was deemed not very intuitive to map target instances to domains, and then use these domains to map (client-side) principals to their granted rights. Instead it makes more sense to use the domain name to obtain (target-side) security policies, because targets, their domains, and their associated security policies form one logical unit. Also, from a more technical perspective, the domain based mapping of principals to granted rights unnecessarily complicates the integration of directory services that store user rights.

In MICOSec, these issues were solved by changing the access control in such a way (non-conformant to the current specification) that the mapping of the principal's attributes to its granted rights is not domain based. Instead, a principal simply has a set of granted rights to invoke operations on all targets. On the target side, targets are mapped to domains, which are associated with the security policy to apply and with the required rights.

6.4 Client-side ODM

Both the SDMM final submission and MICOSec ODM only provide a mapping mechanism for the target side that restricts access to target operations. In terms of information flow, it can control information flow out of (and into) target-side resources. In such a model, CORBASec (i.e., the reference monitor) needs to decide which policy should be applied to protect a resource. To do that, the ODM maps the CORBA object reference that represents the resource to a domain name, and then determines the policy associated with that domain name.

In some cases, it may be useful to also restrict on the client side which operations a client is allowed to invoke on which target. In terms of information flow, such a policy could control information flowing out of the client (e.g., credit card details in an online shopping application). To express such a policy, a client-side ODM mechanism would be required.

For a number of reasons, client-side ODM cannot be implemented analogously to the target side. Firstly, the object key is target-side ORB implementation specific (i.e., can only be interpreted by the ORB on the target-side that generated it). Similarly, only the POA that generated the ObjectId is able to interpret it. Secondly, a client-side policy at a finer granularity than at the server's authenticated identity cannot be trusted by the client, because once the request is within the trust domain of the server, its further mediation cannot be controlled by the client. An early draft of the SDMM suggested that the client should ask the target for the policy associated with an object key. However, this way the client would effectively enforce the target's policy instead of the client's policy, which is not intended.

Because of these issues and its relatively low importance in practice, client-side ODM support was omitted from the current SDMM submission and has not been implemented for MICOSec.

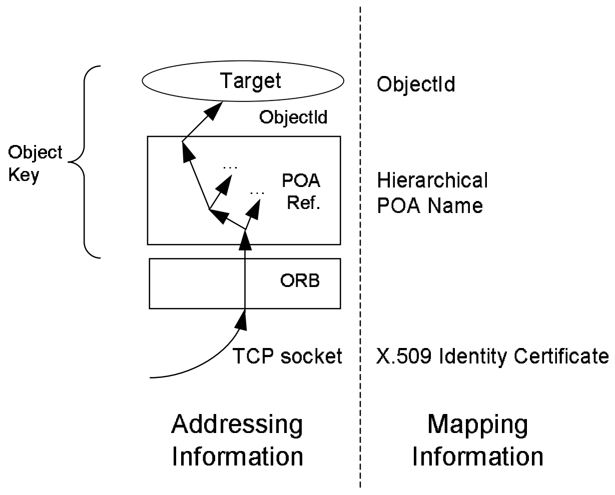

Figure 6.1: MICOSec ODM: Adressing Information and Mapping Information

6.5 MICOSec ODM

The SDMM draft was at an early stage when the design and implementation of MICOSec's ODM was undertaken[2], and so it seemed sensible to redesign and optimise the mapping mechanism. In particular, it is more logical to match the ODM mapping with the mapping of target CORBA objects to servants as far as possible. As a result, it bases its mapping on different information than the previously described SDMM.

6.5.1 Mapping Information

In MICOSec ODM, different mapping information is used to describe the target at several levels of granularity (see figure 6.1): the server identity, the POA name, and the ObjectId. We will discuss each in turn.

Server Identity (X.509 certificate)

As a first coarse level of granularity, it is useful to described the server (i.e., host). While this is not strictly the purpose of the target resource descriptor as defined above, it allows the ODM (in a future implementation) to be centrally administered for a number of hosts. In this case, communications to such a centralised ODM repository would itself also need to be protected.

Instead of the TCP endpoint (specified in the IOR), the identity of the underlying security association was chosen. Since MICOSec is based on SSL, this identity is the X.509 certificate of the server. It describes the server logically, which has a number of potential benefits:

- Servers can be migrated to a different physical location without affecting the mapping (i.e., the description of the server is not tied to the physical location of the

[2]Although the conceptual model and design are part of this dissertation, the actual coding and subsequent integration with MICOSec was done in collaboration with ObjectSecurity Ltd.

host)

- Firewalls in the communications path can be supported, because the TCP endpoint can point to the firewall, while the SSL identity can point to the actual server behind the firewall
- Similarly, indirect binding to an implementation repository can be supported
- Load balancing can be easily implemented by assigning the same certificate to a group of servers. It does not matter on which of the servers a target runs, at this level of granularity all servers will be mapped onto the same domain name.

Because SSL works at the transport layer, all targets running at the same time behind the server's TCP socket will share the same X.509 identity, and therefore will be members of the same domain. However, it is possible to assign different X.509 identities to several CORBA applications on the same host (although they cannot execute at the same time), because the certificate to be used can be specified either by the application itself, or at the command line when the application is executed.

The mapping can be easily configured. For persistent configuration, the certificate is available before the application is launched and can therefore easily be specified in a configuration file. For dynamic configuration at run-time, the application can supply the certificate content as part of the credentials.

POA Name

Next, the target needs to be described more precisely within the scope of the server (described by its X.509 identity). The IOR uses the object key for that purpose, which contains a reference to the POA that serves the target, and an ObjectId, which specifies the target within the scope of the POA.

Although the POA reference uniquely describes the POA, it cannot be used for persistent mapping. This is because its content is ORB implementation specific and unpredictable (i.e., it is not available before the POA is created).

To solve this problem, MICOSec uses a different descriptor, the so-called "POA name". POA names can be assigned to a POA as part of the POA creation process. This can be done dynamically by an application when it creates a POA, or persistently by the administrator as a command line argument when the application is launched. As POAs are arranged in a hierarchical structure, the full POA name contains the names of all POAs from the root POA to the POA that serves the target, and thus uniquely describes the correct POA. Using this POA name, it is possible to define a separate domain name for each POA.

Object Identifier (ObjectId)

In some cases, the ODM should map on a per-target granularity. Individual targets are described in the request header by the ObjectId, which is part of the object key. Within the scope of the underlying POA, this ObjectId uniquely describes the target in question.

It depends on the `IdAssignmentPolicy` of the POA whether persistent configuration is possible, or if the mapping can only be configured dynamically at run-time:

- If the SYSTEM_ID policy is selected, then the ObjectId contains unpredictable information chosen by the POA when the target is created. In this case, only dynamic configuration is possible.
- If the USER_ID policy is used, then the ObjectId that is stored in the IOR can be specified by the application before the servant is activated. At run-time, the ObjectId has to be supplied as an additional element to the mapping function. If the target specified by [X.509 Id, POA name, ObjectId] cannot be found in the mapping table, then the whole search operation is repeated at the coarser level of granularity [X.509 Id, POA name]. Unfortunately, in some cases, additional transient information about the application is stored in the ObjectId, in which case persistent mapping cannot be configured.

This discussion shows that persistent mapping at per-target granularity is more difficult to achieve than at per-POA granularity, because targets are not named persistently. Also, a number of modifications to the application code are required to support per-target granularity, which renders it unusable for security-unaware applications. As a result, the remainder of this chapter will not further explore the use of ObjectIds.

6.5.2 Mapping Configuration

This section illustrates how the ODM mapping is configured. The configuration can be done in two ways. Firstly, the administrator can configure a persistent mapping independently of the application lifecycle. This way, the mapping does not need to be changed when applications get shut down and restarted, and the configuration can be done without modifying the application source code (i.e., for security-unaware applications). Secondly, it is possible for security-aware applications to dynamically create or modify the ODM configuration at run time.

Persistent POA Configuration

This section illustrates how POA names are configured persistently within MICOSec. The POA creation process starts with the root of the POA hierarchy, which is available through an initial reference and has a default policy:

```
CORBA::Object_var poaobj =
     orb->resolve_initial_references ("RootPOA");
PortableServer::POA_var poa =
     PortableServer::POA::_narrow (poaobj);
```

The RootPOA object (accessible through the pointer poa in this example) automatically has the persistent name RootPOA. Within the scope of this RootPOA, child POAs can be created. The following command creates a new child POA object mypoa with the name MyPOA (the other arguments are not relevant for our discussion):

```
PortableServer::POA_var mypoa =
     poa->create_POA ("MyPOA", mgr, pl);
```

The full hierarchical name of this POA is represented in a notation similar to file paths, so that the full name of `MyPOA` is: `/RootPOA/MyPOA`.

Using this command, a hierarchy of POAs can be created that groups targets into domains at a per-POA granularity, and all POAs can be described by a unique POA name. The advantage of using POA names instead of POA references is that it is possible to describe POAs independently from the life cycle of the application in a domain name mapping table. The following MICOSec ODM configuration file illustrates an exemplary persistent mapping configuration (for simplicity, the X.509 identity is not displayed):

```
# POA Mapping
# Keys: AccessId, POA                     Domain

# Default, always applies
# if no mapping was defined
[<X.509>]/                                /CompLab

# The different POAs and their domains
[<X.509>]/RootPOA/                        /CompLab/Domain
[<X.509>]/RootPOA/MyPOA/                  /CompLab/Domain1
[<X.509>]/RootPOA/MyPOA2/                 /CompLab/Domain2
[<X.509>]/RootPOA/AccountPOA/             /CompLab/Accounts
```

This configuration file has to be created manually by the administrator and is read by MICOSec when the application is launched. In the next section, we will show that security-aware applications can also modify the ODM configuration at run-time and afterwards store it back into a file.

Dynamic POA Configuration

The per-POA granularity example above defined a persistent mapping between targets and security domains in a configuration file. In some cases, it may be necessary during the application lifetime to dynamically create or modify the mapping, or to add application-specific ObjectIds into the mapping configuration file at run-time. This section illustrates how this is done inside the servant implementation.

First, an ODM factory for the ODM needs to be created by invoking `create` on the narrowed initial reference to `ODM`:

```
CORBA::Object_var objodm =
     orb-> resolve_initial_references ("ODM");
ObjectDomainMapping::ODM_var odm =
     ObjectDomainMapping::ODM::_narrow(objodm);
```

```
ObjectDomainMapping::Factory_var factory =
     odm-> create();
```

Next, a child POA called `MyPOA` is created, which registers the generated ODM factory:

```
PortableServer::POA_var mypoa =
     poa-> create_POA ("MyPOA", mgr, pl);
mypoa->registerODMFactory(factory);
```

Now the newly created domain manager is associated with the POA. It can be accessed by calling `get_ODM`:

```
ObjectDomainMapping::Manager_ptr dmanager =
     mypoa-> get_ODM();
```

In this example we would like to query if a domain exists in the configuration file for a defined key. Later we will also show how new keys can be inserted into the configuration. If required, a pre-existing configuration file could be read into the ODM by calling the `loadConfigFile` operation on the ODM factory before this is done.

To ask the ODM if a domain name exists for a key, we first have to define the key `okey` that has to be searched for:

```
string key = "[/C=UK/ST=Server State/L=Cambridge
               /O=CompLab/OU=RD
               /CN=ServerTest/Email=server@test] MyPOA";
Security::Opaque okey;
int len = key.length();
okey.length(len);
for (int i = 0; i < len; i++)
    okey[i] = key[i];
```

Once this is done, the list of domain name objects can be retrieved and the domain names can be extracted as follows:

```
SecurityDomain::NameList * list =
     dmanager-> get_domain_names(okey);

  for (int i = 0; i < (*list).length(); i++) {
     SecurityDomain::Name nm = (*list)[i];
     for (int j = 0; j < nm.length(); j++) {
        SecurityDomain::NameComponent nc = nm[j];
        cout << j << "  " << nc.id << endl;
     }
  }
```

It is also possible to define a new mapping in the configuration, which involves the following three steps. Firstly, the new key has to be defined and stored in a suitable variable:

```
Security::Opaque okey2;
string key2 = "[/C=UK/ST=Server State/L=Cambridge
                /O=CompLab/OU=RD
                /CN=ServerTest/Email=server@test] MyPOATR";
len = key2.length();
okey2.length(len);
for (int i = 0; i < len; i++)
      okey2[i] = key2[i];
```

Next, a hierarchical domain name for the key has to be created. In this example, the hierarchy contains only a single layer:

```
SecurityDomain::NameList dl;
dl.length(1);
SecurityDomain::NameComponent nc;
nc.id = CORBA::string_dup("New Domain");
nc.kind = CORBA::string_dup("");
SecurityDomain::Name nm;
nm.length(1);
nm[0] = nc;
dl[0] = nm;
```

Finally, the entry has to be inserted into the mapping table by calling the operation `set_domain_name_key`. This operation allows the modification or complete redefinition of the mapping table at run-time:

```
dmanager2->set_domain_name_key((CORBA::UShort)2, okey2, dl);
```

If needed, the new ODM configuration can also be stored in a file:

```
factory->saveConfigFile("newconfig.cnf");
```

Garbage Collection

To avoid that the ODM mapping table fills up with unnecessary entries, each entry that was dynamically generated by the application should be removed when that application reaches the end of its lifecycle. In MICOSec, applications can do this by invoking an operation `remove_domain_names` on the `ODMManager`, which takes the mapping information as an input parameter.

Potential problems are caused by the fact that the application is still active when it removes its ODM entry. In the time window between the removal of its ODM entry and the destruction of the target instance, the target will be a member of the default domain of the server. Similarly, it will be a member of a default domain of the server during the time window between the time when a target instance is created and when the ODM entry is inserted. This needs to be considered when the default policy is defined in order to avoid possible security holes.

6.5.3 Mapping Process

This section will illustrate how the mapping proceeds at run-time. The ODM mechanism can be used by the interceptors (for security-unaware applications) or by security-aware applications. The mechanism is the same in both cases.

X.509 Identity

As a first step, the target-side X.509 identity needs to be obtained. The following code fragment illustrates how this is done by reading out the `AccessId` attribute from the own `Credentials` of the `SecurityManager` object:

```
securitymanager = orb->resolve_initial_references ("SecurityManager");
assert (!CORBA::is_nil (securitymanager));
secman = SecurityLevel2::SecurityManager::_narrow(securitymanager);
assert (!CORBA::is_nil (secman));

SecurityLevel2::Credentials_ptr own_cred;
own_cred = (*(secman -> own_credentials()))[0];
Security::ExtensibleFamily fam1;
fam1.family_definer = 0;
fam1.family = 1;
Security::AttributeType at1;
at1.attribute_family = fam1;
at1.attribute_type = Security::AccessId;
Security::AttributeTypeList atl1;
Security::AttributeTypeList atl1;
atl1.length(1);
atl1[0]=at1;

Security::AttributeList_out al1 = own_cred->get_attributes(atl1);
```

For further processing, MICOSec's ODM requires the identity to be stored in square brackets:

```
string key1 = "[";
for ( int ctr = 0; ctr < (*al1).length(); ctr++)
    {
        key1 += (char *)(&(*al1)[ctr].value[0]);
    }
key1 += "] ";
```

POA Name

At run-time, the ODM mechanism needs to obtain the POA name associated with the object key specified in the incoming request. In theory, it would be possible to generate another table that maps POA references to POA names. A new entry would need to be added to the table whenever a POA gets created, and the table would need to be searched for every incoming request. Such a complex approach has several drawbacks: firstly it would lower the performance of the mapping process, and secondly it would require major changes of the ORB and POA interfaces. Thirdly, it would not be possible to separate the ODM implementation (which should be developed by security specialists) from the ORB and POA (which should be developed by ORB vendors).

Fortunately, a much more elegant solution was found. The ORB stores information about the current execution context associated with an invocation. In particular, the target-side ORB stores the POA name in the `POACurrent` object, which can simply be read out by the ODM mechanism by invoking the operation `the_name` on the POA. The following two code fragments illustrates how this is done:

```
CORBA::Object_var poao =
     orb->resolve_initial_references ("POACurrent");
PortableServer::Current_var cpoa =
     PortableServer::Current::_narrow(poao);
PortableServer::POA_ptr poa = cpoa->get_POA();
```

To obtain the full POA name, the POA hierarchy has to be followed up to the `RootPOA`, and individual POA names have to be appended to a string (separated by slashes):

```
PortableServer::POA_ptr np = poa;
string key2;
string tstr;
while (np != NULL) {
    tstr = np->the_name();
    if (key2.length() > 0)
        tstr += '/';
    tstr += key2;
    key2 = tstr;
    np = np->the_parent();
}
cout << "POA=" << key2 << endl;
```

Mapping Process

Now the actual mapping can be carried out. The `ODMManager` that stores the mapping table for the POA can be obtained through the `get_ODM` operation on the POA:

```
ObjectDomainMapping::Manager_ptr dmanager = poa->get_ODM();
```

Finally, the domain names can be obtained by supplying the two generated strings to the mapper as an opaque parameter:

```
key1 += key2;

  if (dmanager) {
    Security::Opaque okey;
    int len = key1.length();
    okey.length(len);

    for (int i = 0; i < len; i++)
      okey[i] = key1[i];

    SecurityDomain::NameList * list =
        dmanager->get_domain_names(okey);
```

Finally, the domain names can be obtained from the domain name list like this:

```
    cout << "Domain=";
    for (int i = 0; i < (*list).length(); i++) {
      SecurityDomain::Name nm = (*list)[i];
      for (int j = 0; j < nm.length(); j++) {
        SecurityDomain::NameComponent nc = nm[j];
        cout << "/" << nc.id;
      }
      cout << endl;
    }
  }
```

6.5.4 Modifications to the CORBA Specification

One of the goals of the ODM design was to keep modifications to existing CORBA specification to a minimum to separate the ORB and POA implementations from the ODM, and thus to fulfil CORBA's portability and flexibility requirements.

Only a few modifications to the MICO ORB were necessary, in particular two functions to the POA that allow the registration of an ODM factory with a POA and to query the ODM:

```
void registerODMFactory
     (in ::ObjectDomainMapping::Factory fry);

ObjectDomainMapping::Manager_ptr
    MICOPOA::POA_impl::get_ODM()
```

Also, an initial reference "`ODM`" had to be added, so that a reference to the ODM can be obtained. This reference is necessary to allow access to the ODM from within interceptors and servants. Without this, features like dynamic configuration could not be implemented.

6.6 Summary

In this chapter, we have presented our proof-of-concept implementation of the resource descriptor mapping for the MICOSec CORBA security services implementation. It maps (target-side) X.509 identities, hierarchical POA names, and ObjectIds to hierarchical domain names. The chapter illustrated how the mapping can be configured persistently and dynamically, and how the mapping is carried out at invocation time inside interceptors and servants.

The X.509 identity is only useful if the mapping is not stored locally. If the mapping information is stored and possibly carried out centrally for several nodes (e.g., in a centralised security administration console), it is a useful and trustworthy mapping attribute to precisely describe the middleware component that resides below the invoked target.

The most important mapping attribute is the hierarchical POA name. MICOSec ODM reuses the POA hierarchy as the domain hierarchy, which is simple, elegant, and ensures that the domain names and the hierarchy are coupled to the target addressing mechanism. Each POA constitutes a domain, therefore the POA hierarchy has to be created to reflect the desired domain hierarchy. The use of the user-defined POA name instead of the dynamically assigned POA reference has various further advantages. In particular, it facilitates the persistent specification of the mapping (i.e., before the POA hierarchy has been created), and it allows the use of the same human readable text string to be used for the POA name and for the domain name.

Mapping on a per-target granularity is also supported, although persistent configuration can sometimes be difficult because of the unpredictable and implementation specific ObjectId content.

This chapter demonstrates that the resource descriptor mapping presented in this dissertation works in a real-world environment, and that it can be integrated well with the complex CORBA (and POA) middleware architecture. Moreover, it shows that modifications to the existing specifications can be kept to a minimum, so that flexibility and portability between ORB (and POA) implementations are not inhibited.

Chapter 7

Middleware Security

7.1 Introduction

The earlier parts of this dissertation explored the challenges inherent in implementing the CORBA security architecture. Since the specification of OMG CORBA and CORBASec there have been significant developments in both middleware research (e.g., component-based middleware, reflective and adaptive middleware, Web services) and security research (e.g. various security models). In this chapter we review the extent to which these developments make middleware security more or less challenging to achieve, and examine the relevance of the resource descriptor mapping described in chapter 4. The chapter concludes with some general observations as to how middleware should be designed to accommodate security.

7.2 Enterprise Java Beans (EJB)

7.2.1 Java Security

Java differs from many other programming languages (such as C++) in that its source code is compiled into Java bytecode, which is interpreted by a Java virtual machine (JVM) [54]. The JVM is an abstract computer that runs on top of a host machine's native operating system and insulates applications from the differences between underlying operating system and hardware. The JVM includes a number of security features [51] (originally called "sandbox model" in JDK1.0.x [71]). Firstly, the class loader prevents untrusted classes (normally downloaded "applets" that are assumed unsafe to execute) from impersonating trusted classes (normally the local Java runtime environment classes) by providing name-spaces for classes loaded by different class loaders. Secondly, the bytecode verifier checks that the bytecode is (to a large extent) type-safe and conforms to the language specification (e.g., no forged pointers, no circumvented access restrictions, no overflowed stacks). Thirdly, the security manager performs run-time verification of access to crucial system resources (e.g., I/O, network access). Since JDK1.1.x [73], applets can be digitally signed to be treated as if they were trusted local code. JDK1.2 [72] introduced a number of additional security features [52]: First, all code, regardless of whether it is local or remote, can be subject to a security policy. The security policy defines the set of permissions available for code from various signers or locations and can be configured by a user or a system administrator (e.g., with an external configurable policy file). Each permission specifies a permitted access to a particular resource (e.g., read and write access to a specified file or directory). The runtime system organizes code into individual domains, each of which encloses a set of classes whose instances are granted the same set of permissions. Moreover, an access controller class allows code to query whether a permission would succeed if performed at the time of the query, and a secure class loader loads classes and records the protection domains they belong to. More recent versions of the JDK include some additional features (e.g., the Java Certification Path API and the Java GSS-API) and integrate several previously optional packages (e.g., Java Authentication and Autorization Service, Java Cryptographic Extension, Java Secure Socket Extension).

7.2.2 The EJB Container

Sun Microsystems' Enterprise Java Beans (EJB) [158] is an open standard for Java-based server component architectures. The core middleware component of EJB is called the container, or application server. It hosts the application components, so-called "enterprise beans". Enterprise beans are built from Java classes and have an interface that exposes the business logic. Enterprise beans are often designed to be deployed as part of different applications in different containers.

A container provides and manages many of the low-level details for the enterprise beans, such as life cycle management, transactions and security. In contrast, a CORBA ORB only provides CORBA services to the application, but does not transparently manage them for the application.

The container takes care of all interactions between enterprise beans in the same container or across the network. An enterprise bean can also act as an initiator or a recipient of an interaction. The EJB standard facilitates interoperability between different EJB container products. In addition to its native RMI protocol, EJB supports CORBA IIOP, which allows CORBA clients to access server EJB beans.

7.2.3 Provider, Assembler, Deployer

In the past, distributed applications have normally been developed and deployed within a single organization. It was not very common that an organization bought parts of the application from a vendor and integrated them into its internal applications. This was mainly a consequence of the missing support for life cycle management, the assembly of objects and their deployment. Therefore the applications as a whole and their environments were often under the control of the organization developing the application, and appropriate security measures could be applied during the whole application life cycle and in all its individual parts. For example it was possible to implement security functionality directly into the application because at the stage of application development the security requirements and the environment the application had to run in were already known.

In component-based systems such as EJB, the situation is different: Applications are not anymore developed in a monolithic style by a single organization, but using a collection of different beans provided by several vendors. This has far reaching consequences for the specification and enforcement of security policies, since now the development process of the application is distributed between several parties [158]: The "bean provider" offers beans with well defined interfaces that implement business logic. The “application assembler” assembles these beans to complete applications, but has no detailed knowledge of their inner working or structure. The “deployer” installs and runs the application in a container, which provides a runtime environment including additional services such as security. The duties of the three parties are clearly separated: The bean provider is a specialist for the implementation of the business logic of a single bean, but does not know in which applications and in which environments the bean will later be used. The application assembler knows the interfaces of the beans and the business logic of the application as a whole, but does not know the inner working of the beans or the environment where the application will be deployed. Finally the deployer knows the specific requirements of the environment, but for him the application itself and the beans it is composed of are black boxes.

7.2.4 Declarative Security

The described separation of duties and of knowledge about requirements and internals of beans and application, has far-reaching consequences for the implementation of security specification and enforcement. The bean provider, who has access to the source code of the bean, cannot implement the security policy and enforcement directly in the bean itself, since he normally does not know what the bean finally will be used for and what the concrete security requirements are. The deployer knows the security requirements, but cannot modify the code of the beans anymore. As a consequence, a “declarative” approach is used, where the policy is specified in an XML deployment descriptor outside the application. The security enforcement, in particular access control, is implemented in the container.

EJB supports role-based [140] access control policies, which has several advantages. Firstly, roles group users in the access control policy for scalability and manageability. Secondly, roles provide the required level of indirection that allows the separation of duties between the bean provider, the application assembler, and the deployer: the bean provider and application assembler specify (general) roles, while the deployer associates individual user identities with these roles.

The security-related tasks of the bean provider, application assembler, and deployer are illustrated in figure 7.1 and described in the following:

The bean provider is normally not concerned with security. If application specific policies need to be coded into the application, security-aware beans can call `getCallerPrincipal` and `isCallerInRole` on the `javax.ejb.EJBContext` interface to obtain the current calling principal (it can be used to obtain the principal's name using the `getName` method), and to check whether the caller is in a particular role [158]. If this is done, then the bean provider must specify the used roles in the deployment descriptor. However, coding security features into the application breaks the concept of declarative security.

The application assembler, who at least needs to know the interfaces of the different beans and who knows the business aspects of his application as a whole, defines the security aspects of the application at a high level. In particular, he describes in the deployment descriptor the security roles that can be used by the deployer, and the access control rules for beans and their methods. The deployment descriptor contains the names of the roles used by the bean and also describes them in a human readable manner.

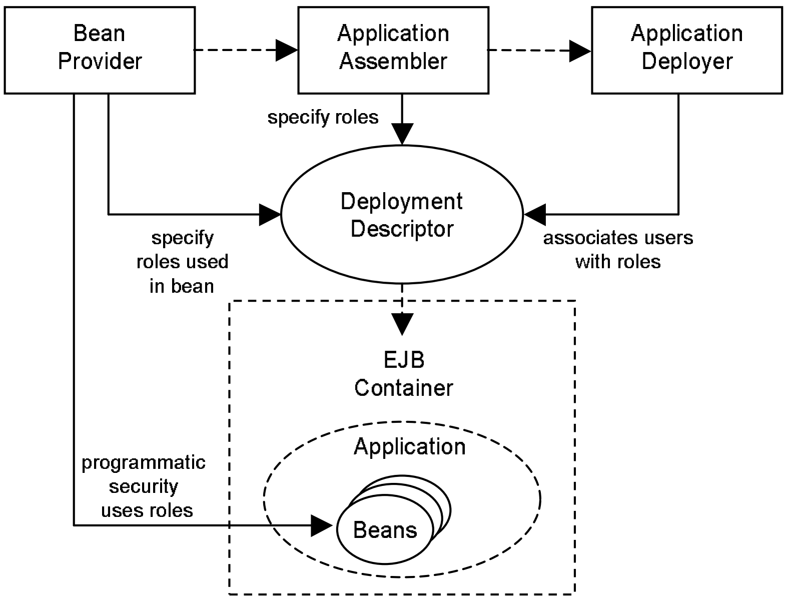

Figure 7.1: Provider, Assembler, Deployer

This role description is purely abstract and independent from real world users and their security properties. Access control rules for the methods of the bean are specified using simple (role, method)-pairs that specify the role needed to invoke a certain method on the bean. This can be done individually for each method or for all methods in a bean, possibly using wildcards. Roles can also be linked together in the deployment descriptor.

In addition to the definition of the access control rules the assembler is also responsible for defining how credentials (e.g., identity) can be used by beans to call other beans. EJB allows the initiator to give an intermediate bean the right to forward all his credentials to any other component (sometimes called "impersonation"). The rules that specify which identity an intermediate object has to use for further invocations are also defined in the bean's deployment descriptor. The default mode is to use the bean's identity itself. With `user-caller-identity` the initiator's identity is forwarded to the server, and `runAs-specified-identity` allows flexibly setting an arbitrary identity for the calls. Unfortunately, a user has no way to restrict the privileges passed on to others.

Finally, the deployer is responsible for assigning principals (i.e., users) described by their mechanism specific security attributes with the abstract roles used in the declaration of the access control and impersonation policies.

To assist the described players with their duties, the container vendor should provide a security tool chain that should include user and role management tools.

While the described declarative approach has many advantages, it also has considerable limitations: It is only possible to declare security policies supported by the container implementation, while the flexible definition and implementation of security policies is not possible. This restricts the security models that can be enforced, and the security policy can only be based on a limited set of arguments (subject identity or role, target identifier, and invoked operation). Other criteria, such as the operation arguments or other environmental information like time, cannot be used in the definition of the security policy.

7.2.5 EJB Security Model

The basic EJB security model is simple [57]: when a client program invokes a method on a target enterprise bean, the identity of the user associated with the calling client is transmitted to the target-side container. The container checks whether the caller's identity (or role) is in the access control policy associated with the bean's invoked method. If the caller's identity (or role) matches, the container permits the invocation or otherwise rejects it. The authentication method applied, as well as message protection, depend on the underlying security technology (e.g., SSL/TLS).

Most of the security functionality resides inside the EJB container, which allows application programmers to write beans without having any knowledge of security. The container and the underlying security mechanism (e.g., SSL or Kerberos) provide authentication, message protection, and access control invisible to the application (EJB currently does not provide any auditing). The sole place where the security mechanism is visible (to the deployer) is the vendor specific assignment of the identities used by the security mechanism to the abstract roles of the EJB access control. As described above, security-aware applications can optionally obtain the principal and the role of the current user.

While the specifics of the security mechanisms are not relevant in the abstract security model or as far as a single container is used, they are most relevant for interoperability of applications using different containers. Only if the same communications protocol with the same data format and the same encryption is used, interoperability between components in containers from different vendors can be achieved. Since this was a major objective during the development of the EJB standard, a security protocol was defined jointly by the CORBA and the EJB communities, the Common Secure Interoperability version 2 (CSIv2) [120] protocol (level 0/stateless with username/password authentication over SSL with several required ciphersuites). This protocol enables security interoperability between EJB components, CORBA clients, and CORBA components.

7.2.6 EJB and the Resource Descriptor Mapping

In EJB, all beans of the same type in a container share the same security policy (i.e., EJB does not support instance-based authorisation outside the application [57]). There is no way to define policies for single instances, or to express the notion of "ownership" (or any other attribute than role). Moreover, there is no support for grouping bean instances into logical domains. The J2EE 1.4 draft specification [34] mentions the need for instance based access control in its "future work" section.

EJB can be enhanced to support instance-based access control using the resource descriptor mapping described in chapter 4. However, this would require modifications to the container implementation. In fact, several EJB container vendors provide some form of instance-based access control, such as IBM's Websphere and Borland's J2EE Application Server Security Service.

It has also been demonstrated in [42] that the Java Authentication and Authorization Service (JAAS), which is part of the Java 2 SDK Standard Edition v1.4 [157], can be extended to support a more restrictive form of class instance-based authorisation. This approach introduces the notion of a class instance owner. The general policy that a class instance can only be accessed by its owner can be specified in an extended JAAS permission statement. Classes that require class instance-based authorisation must implement an interface with a `getOwner()` method that returns the owner of the class instance. Note that this approach implies that each class instance needs to know its owner (the described framework also includes "special relationships" in addition to instance ownership).

7.2.7 EJB Summary

EJB security is similar to CORBA security in that it provides security on the middleware layer, but also offers application-layer security for security-aware applications. EJB's access control uses roles, while CORBA's access control is based on a required/granted rights model. Both security models only support type-based access policies. EJB's security model fits to the bean development, assembling, and deployment process by allowing each involved party to define the security policy in the deployment descriptor.

7.3 CORBA Component Model

The CORBA Components Model (CCM) [126, 53] (which is part of CORBA 3.0) is a programming language independent specification for creating server-side, scalable distributed applications. CCM extends the traditional CORBA object model by defining features and services that allow application programmers to implement, manage, configure, and deploy components that integrate commonly used CORBA services, such as persistence, security, transaction, and event services, in a standard environment [167]. A component reference is comparable to a normal CORBA IOR to support compatibility with previous CORBA versions [148].

The CCM architecture is a programming language independent extension of the (Java-specific) EJB architecture. As with EJB, the main constituents of the model are components, container, infrastructure, as well as standardised development and distribution processes. The "basic CCM" is equivalent to the EJB 1.1 Component Architecture, while the "extended CCM" adds multiple ports, more advanced persistence, an event model, and participates in the construction of component object references [126]. The basic CCM supports a standard mapping between CCM and EJB. Therefore, a CCM component can appear as an EJB "bean" to EJB clients, and an EJB bean can appear as a CCM component by using appropriate bridging techniques (EJB also supports CORBA IIOP) [167].

7.3.1 CCM Security

The security of CCM is modelled after EJB security and uses CORBASec 1.2 as an underlying security service. The CCM container relies on CORBASec to obtain the security policy declarations from the deployment descriptor and to check the active credentials for invoking operations. Access permissions (i.e., CORBASec required rights, rights families, and rights combinators) are defined by the deployment descriptor for any of the component's ports (as well as the component's "home" interface and event ports). The granularity of permissions must match the set of rights recognised by the underlying CORBASec implementation.

Credentials can be established either using SSL (using a proprietary API) or SECIOP (using the CORBASec API), which take care of secure transportation of credentials between systems.

The CCM container programming model defines a set of APIs that simplify the task of developing and/or configuring CORBA applications [93]. In particular, the internal interfaces of the container API, which make container services available to the component, offer security-related methods: `get_caller_principal()` obtains the CORBASec credentials in effect for the caller, while `is_caller_in_role(in string role)` serves to compare the current credentials to the credentials defined by the role parameter. These operations match EJB's `getCallerPrincipal` and `isCallerInRole` methods.

There is some mismatch between different security models: On the one hand, the container API supports role-based access control (RBAC) in line with EJB's security architecture. On the other hand, the deployment descriptor specifies access control in terms of the CORBASec rights model, where rights are granted to principals that have to match the rights required to invoke an operation on a target.

If the CORBASec rights model should be used, then the container API should return the calling principal identity, not the role.

If RBAC should be used (it is possible to simulate RBAC in CORBASec – see section 7.7), it would be more elegant to specify required roles instead of required rights in the deployment descriptor, as done in EJB security. Unfortunately SSL, the most widely used security mechanism for CORBASec, does not support the notion of roles.

As a result of this mismatch, identity credentials have to be mapped to roles on the target side, which then need to be mapped to granted rights, and compared to the required rights of the called target. Having these two levels of indirection is neither elegant nor efficient, as each model effectively makes the other obsolete.

In general, CORBASec 1.2 does not meet the requirements of complex component-based applications well. For example, controlled delegation of credentials (e.g., authorisation information) is a critical requirement that is not supported by CORBASec (over SSL). The OMG Common Secure Interoperability (CSIv2) [120] protocol supports controlled delegation, even over SSL, but does not integrate well into CORBASec 1.2.

7.3.2 CCM Security and Resource Descriptor Mapping

Just like standard CORBASec, CCM security does not support component instance-based access control outside the application. As a result, it would benefit from the resource descriptor mapping described in chapter 4.

The COACH project [29] plans to develop a new OMG security services specification for CCM that will also include the resource descriptor mapping.

7.3.3 CCM Summary

The security of CCM is modelled after EJB security and uses CORBASec 1.2 for security. There is some mismatch between different security models in that the container API supports RBAC, while CORBASec (and the policy in the deployment descriptor) are based on a required/granted rights model.

7.4 .NET

In June 2000, Microsoft announced its .NET initiative [109], a "vision" that embraces the Internet and the Web in the development, engineering and use of software. It was declared as being simultaneously a firm strategy, a technological infrastructure, a development platform built around XML, and "a large set of possibilities for users". Microsoft's .NET currently only works on Microsoft platforms, but there is also Ximian's Mono Project [110] that implements an open source Unix version of .NET. Furthermore, the iNET project by Halcyon Software aims to re-implement .NET in Java [56], and Intel Labs' Open CLI Library (OCL) project [155] plans to develop an Open Source implementation of portions of the CLI (Common Language Infrastructure) runtime library as defined in a draft ECMA CLI specification.

Although Microsoft has recently decided to drop .NET in its product branding [169], we will describe the range of technologies that were developed as part of the .NET initiative.

7.4.1 .NET Framework

The .NET Framework is the set of programming interfaces at the heart of the Microsoft .NET platform. Visual Studio.NET is a multi-language suite of programming tools. .NET supports a number of different programming languages (currently 27), such as Visual Basic, Visual C++, VBScript, Visual J# (Java), and Jscript (JavaScript), as well as a new type–safe programming language called Visual C# that is derived from C and C++.

NET application code is compiled into so-called "assemblies" that comprise code in the format of an intermediate language (MSIL) and some metadata that describes each class, in addition to security data, version data, and references to other assemblies. Assemblies are processed by a programming-language unspecific Common Language Runtime (CLR). The CLR examines code (so-called "managed code") to enforce code safety (e.g., memory and system resource management) and code access security. Although conceptually similar, the CLR differs from the Java JVM in that the code is just-in-time compiled and not interpreted. The CLR includes the Common Type System (CTS) that defines the roles the CLR follows when declaring, using, and managing types. All .NET languages use these primitive types. .NET assemblies can be deployed by simply copying the assembly to the required directory. Assemblies can be shared using a so-called "strong name" that includes the name and version, as well as a public key and a digital signature.

.NET provides a collection of language-neutral Base Class Libraries (BCLs) that provide support for networking, security, and other "base" services. It supports the creation of self-describing software components, and facilitates multi-language integration, cross-language component reuse, and cross-language inheritance.

7.4.2 Code Access Security

A core part of the CLR is code access security, which is conceptually related to the Java sandbox model. Code access security determines the access permissions assigned to a piece of code based on its origins, publisher and other factors, without any active involvement of the application. .NET uses the concept of "evidence-based security" [108] to describe the process by which application code is examined at run-time. Access permissions to resources are granted depending on the "code group", which is defined by the origin (e.g., URL, Web site, Internet Explorer zone, hash of the assembly, application directory, "strong name" signature of the assembly) and the creator of the code (e.g., publisher certificate). A code access permission is the right of a piece of code to access a particular resource or perform a particular operation (e.g., access files, registry, network, user interface, or the execution environment). When an application is built, the permissions that it requires to run can be included as part of its description. Further evidence can be added by the hosting environment.

In addition to the validation of metadata when the assembly is loaded into the cache, the actual MSIL code is verified to be type-safe, to not include stack underflow/overflow, and to correctly use of the exception handling facilities and object initialisation [170].

There are a number of configurable security policies (e.g., enterprise policy level, machine policy level, user policy level) that determine the mapping between the assembly evidence that a host provides for an assembly, and the set of permissions (e.g., FullTrust, SkipVerification) granted to an assembly. Assemblies can also directly state the minimum required set of permissions or refuse certain permissions. During code access security evaluation, other assemblies ("policy assemblies") might need to be loaded to be used in the policy evaluation process.

The intersection of the permissions established at the different policy levels is compared with the required set of permissions to determine whether the code should be allowed to execute or not. If there is a call chain involving several assemblies, then so-called "stack walking" is carried out to prevent the "luring attack" [108] where malicious code tricks more trusted code into something it cannot do alone.

The CLR also enables so-called "unmanaged code" (i.e., code that is not examined by the CLR) to run, but unmanaged code does not benefit from these security measures, and permissions to call into unmanaged code should be conservatively granted.

The .NET Framework allows developers to express security constraints in two styles. Expressing security constraints in the "declarative" style means using custom attribute syntax in the metadata of the code. Expressing security requirements in the "imperative" style means creating instances of `Permission` objects at run-time and invoking methods on them. Not all security actions can be expressed using the declarative style, and only the imperative style allows constraints to be expressed at runtime. [170].

7.4.3 Role-Based Access Control

.NET security also supports a role-based access control model for expressing security settings based on user identities [170]. The model is based on the notion of a principal on whose behalf the code is executed. The authentication process involves examining credentials (e.g. username/password) and establishing the identity of the principal. In addition to identity, a principal may have zero or more roles to which it belongs, representing authorisations. Applications can learn the identity of the current principal, or query it for a particular role as necessary to perform some privileged operation using the `User.IsInRole` method. Alternatively, roles can be checked declaratively in the meta-data, freeing the business logic from having to deal with access checks [96]. In .NET, the Windows login username can become the principal identity, and the groups the user belongs to are the names of the roles assigned. Alternatively, a generic principal object can be used for application-specific authentication and authorisation [55].

The .NET framework also includes various cryptographic functions [55] for encryption, digital signatures, and random number generation for use from within security-aware applications and for internal use of .NET.

ASP.NET (Active Server Pages) uses Microsoft's Internet Information Server (IIS) to provide authentication (basic, digest, NTLM, Kerberos 5.0, SSL/TLS, Passport, Cookie) and authorisation (URL, Windows users/groups, application defined roles) [48]. Custom authentication and authorisation providers can also be integrated. ASP.NET allows per-folder configuration of role-based security outside the application [112].

7.4.4 .NET Remoting Framework

To allow distributed objects to interact with one another across application domains, .NET provides the ".NET Remoting Framework" [115], which to some extent is comparable to DCOM, Java RMI, and CORBA. It hides some of the complexities of calling remote methods, and provides a number of services, including several activation options, lifetime support, callbacks, passing objects by reference and by value, additional context information specific to .NET, events, as well as (multithreaded) communication channels for transporting messages to and from remote applications. The Remoting Framework can be configured from the application or using configuration files. When a remote object is called, clients normally create proxy objects that serve as representatives that transparently forward local calls to the remote server. Formatters are used for encoding and decoding the messages before they are transported by the channel. Applications can use configurable protocols, such as binary encoding or XML encoding (with SOAP) over HTTP or TCP.

It has been shown in [8] that authentication, signing, and encryption services can be integrated into distributed applications that use the Remoting Framework. In this work, the Security Support Provider Interface (SSPI) [105], which is based on the Generic Security Service API (GSSAPI) [94, 176], is used. GSSAPI abstracts the implementation details of different authentication protocols and provides a single, common programmatic interface. It has further been illustrated in [9] that security features can be added to the .NET Remoting Framework transparently to the application logic through interceptors (so-called "message sinks" and "channel sinks") and .NET configuration files (programmatic configuration is optional). The described solution allows a choice of any Windows authentication protocol (in particular Kerberos, NTLM, and Negotiate), different impersonation levels (client's trust to the server; allowable settings are identify, impersonate, and delegate), different authentication levels (determines when authentication should take place), how messages should be protected, and the ability to set a custom principal on the server side, especially to support role based security.

.NET (and especially Remoting) differs from traditional middleware (such as COR-

BA) in that it does not provide a comprehensive and transparently integrated security architecture for invocations between distributed objects. Instead, it includes a diverse range of security tools and technologies that can applied outside the application (through configuration or the declarative specification of security metadata), or be applied imperatively within the application. As a result, .NET security is currently still somewhat of a patchwork that, if applied correctly, can fulfil its task with a high degree of flexibility, but that can also easily be applied inappropriately.

7.4.5 .NET and Resource Descriptor Mapping

.NET supports per-class (or per-method) access control outside the application by allowing the specification of required roles in the metadata of a class or method. It does not include class instance-based access control outside the application, and as a result would benefit from the resource descriptor mapping described in chapter 4.

7.4.6 .NET Summary

.NET source code is compiled into an Intermediate Language (MSIL) (and some metadata) that is processed by the Common Language Runtime (CLR). Similar to the Java virtual machine, the CLR enforces code safety and code access security based on code properties and evidence. Like EJB and CCM, .NET also supports role-based access control. Remote method invocations can be implemented using the .NET Remoting Framework, which is comparable to DCOM, Java RMI, and CORBA. .NET Remoting does not include a security architecture, but can be security-enhanced manually.

7.5 XML Web Services

A more recent development related to message-oriented middleware [7] revolves around so-called Web services, which are sometimes described as "middleware based on XML" [43], or as "programmable application logic accessible using standard internet protocols" [79]. Several vendors provide their own definitions of Web services. For example, Sun Microsystems defines a Web service as "specific business functionality exposed by a company, usually through an Internet connection, for the purpose of providing a way for another company or software program to use the service" [97]. IBM's definition states that "Web services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web" [162]. For Microsoft, a Web service is "a unit of application logic providing data and services to other applications. Applications access Web services via ubiquitous Web protocols and data formats such as HTTP, XML, and SOAP, with no need to worry about how each Web service is implemented..." [171]. Throughout the literature, one primary goal of Web services is to develop a "machine readable web" [166].

7.5.1 Web Services Protocols

From a technical perspective, Web services combine aspects of component-based development and the Web. Like components, Web services can be used without knowledge about how the service is implemented. However, unlike component technologies, Web services are not accessed via technology-specific protocols such as Java RMI or CORBA IIOP. Instead, they are accessed using widely-used Web protocols and data formats, in particular the Hypertext Transfer Protocol (HTTP) [41], which is mainly used to transfer Web pages, and the Extensible Markup Language (XML) [18], which is a simple and flexible text format derived from SGML (ISO 8879 [69]). XML is called extensible because it is not a single, predefined markup language like HTML [134]. Instead, XML is a "meta-language" – a language for describing other languages – with which customized markup languages for different types of documents can be created. A Web service is defined only in terms of the XML messages the Web service accepts and generates, and clients can be implemented on any platform in any programming language, as long as they can create and process the messages defined for the Web service interface. Web services are located by a unique Uniform Resource Locator (URL) [160], which can be persistent or dynamic.

Web services are built using three core standards. First, the Simple Object Access Protocol (SOAP) [17] defines a (supposedly lightweight) protocol for information exchange. The specification defines how XML should be used to represent data (inside an "envelope"), coding rules for an extensible message format, conventions for representing RPCs, and bindings to HTTP. In theory, Web services can also communicate over other protocols, but it is anticipated that mainly HTTP will be used due to its widespread use on the Internet. One of the advantages of SOAP is that it is (supposedly) significantly less complex than earlier approaches [175], such as IIOP or RMI – for example, it does not specify object activation or a naming service.

Second, the Web Services Description Language (WSDL) [27] is an XML-based contract language that documents what messages a Web Service accepts or generates. A WSDL file is an XML document that describes a set of SOAP messages and how the messages are exchanged. To some extent, WSDL is to SOAP what IDL is to CORBA. The WSDL notation to describe message formats is based on the XML Schema standard [22], which makes it both programming-language neutral and standards-based. In addition to describing message contents, WSDL defines where the service is available and what protocols should be used. WSDL defines the interfaces required to write a program to work with a Web service.

Third, another industry effort revolves around the Universal Description, Discovery, and Integration (UDDI) [11] protocol specification, which allows Web service providers to advertise the existence of their Web services and for Web service clients to locate Web services of interest. A XML directory entry consists of three parts. The "white pages" describe the company offering the service; the "yellow pages" specify industrial categories based on standard taxonomies; the "green pages" describe the service interface so that users can write applications to use the Web Service. The UDDI directory allows a search for Web services to take place in several ways, such as type of service or geographic location.

In summary, an XML-based Web service can be defined (from a technical perspective) as a software service exposed on the Web through SOAP, described with a WSDL file and registered in UDDI [175].

From an n-tier application architecture perspective, a Web service allows programmatic access to a service which is then implemented by other means. The front-end that deals with XML request and reply handling is often called the "listener", while the part that exposes the operations supported by the business logic is called the "facade" [165]. The business logic behind the facade is then implemented by other means (e.g., using a traditional middleware platform such as CORBA).

Current Web services products include Sun's Java 2 Platform Enterprise Edition (J2EE) framework [159], or Microsoft's .NET framework [109]. Other more well-known Web services development platforms include Sun ONE, Microsoft .NET, IBM WebSphere/dynamic e-Business, and BEA Web Logic.

The World Wide Web Consortium (W3C) is currently the main standardisation body that releases standards related to Web services and XML. Another global consortium that is active in the area of Web services standards is the Organization for the Advancement of Structured Information Standards (OASIS). Moreover, the Internet Engineering Task Force (IETF) standardises several Web service related technologies, such as the SSL/TLS [32] protocol, and the Web Services Interoperability Organisation (WS-I) publishes interoperability profiles. Individual companies (e.g., IBM, Microsoft) have also released their own specifications independently of W3C or OASIS.

7.5.2 Web Services Security

Security is important for Web services, as more and more comprehensive services are made available across the Internet, and as Web services become increasingly interconnected "ad hoc" across organisations over the Internet. This environment is more hostile than the environments covered by more traditional middleware such as CORBA. In fact, the lack of security features is considered the most significant inhibitor for the large-scale adoption of Web services at the moment [45, 44].

Apart from the typical security challenges of large-scale distributed applications across heterogeneous systems and administrational domains, Web services have to deal with the fact that XML documents contain links to other documents. It is currently unclear how are these links can be secured.

At the moment there is no complete security model for Web services: instead there is a mishmash of emerging protocols for individual security features [45] (the current SOAP standard assumes security is a transport issue and is silent on security issues [175]). At the time of writing it is unclear which of these specifications will ultimately be used in practice (some authors expect the specifications to settle by mid-2003 [147]). Some of the more promising emerging specifications are described in the following.

The W3C consortium standardised several specifications related to cryptography: The XML Digital Signature (XML-DSIG) provides integrity, signature assurance and non-repudiation for whole or partial documents (it needs the Canonical XML specification, which normalises an XML document down to a form that always uses a standard syntax, so that signed documents can be accurately compared). The XML Encryption (W3C) standard encrypts and decrypts digital content for whole or partial documents, and the XML Key Management Specification (XKMS) provides a method to access public key infrastructure services for distributing, registering, and validating public keys.

The OASIS consortium published several specifications that allow the exchange of security information. In particular, the Security Assertion Markup Language (SAML) allows the exchange of information about the end-user's authentication, authorization, and attributes to the receiver; allows users to carry entitlement ("security assertions") with them across multiple sites, thus facilitating single sign-on. The Extensible Access Control Markup Language (XACML) is used to express access control policies for whole documents or individual elements, and how access control is transferred. The WS-Security [35] specification extends SOAP with XML security protocols ("security headers") for integrity, confidentiality, and authentication; allows security tokens to be associated with messages and describes how binary security tokens are encoded. It provides a framework for different security assertions and certificates (e.g., SAML, Kerberos, X.509 certificates and PKI) to exchange data in a standards-based way, and defines the sequence of processing of encryption, signature, and authentication. Furthermore, it separates encryption, signature, and authentication to cater for multi-hop scenarios where only some information should be understood by a particular node. Microsoft, IBM, and Verisign (and others) released several additional specifications that extend WS-Security and form the "Global XML Web Services Architecture" (GXA) [16]: WS-Trust, WS-SecureConversation, and WS-SecurityPolicy. They are planning to release several additional security specifications in the near future, in particular WS-Policy, WS-Privacy, WS-Federation, WS-PolicyAttachment, WS-PolicyAssertion, and WS-Authorisation [38].

In the meantime, due to the currently unclear future of these emerging specifications, many Web services resort to the ubiquitous HTTPS (i.e. HTTP over SSL) protocol to secure communications. This "channel security" [85] approach may be sufficient for point-to-point applications, but as applications start becoming loosely coupled in multi-tier environments, a "package security" [85] approach (as taken by the emerging specifications mentioned above) will be required to provide end-to-end security across multiple nodes. Package security allows parts of the message to be encrypted and signed for particular nodes in the communication path.

7.5.3 Web Services and Resource Descriptor Mapping

Web services are "services offered computer to computer" [38]. On the last node on the communications path, the Web service listener is the target-side endpoint of the security association, and the corresponding reference monitor resides in the listener (or the facade). Therefore the target side resembles the "N-to-1 Middleware Security Model" described in section 4.2. As a result, clients can only authenticate the listener of the Web service (i.e., at a per-listener granularity), and (normally) all services that reside behind the listener share the same identity (e.g., the SSL socket). If the node is an intermediate on the communications path, then some authentication information can be passed on to the next hop in the path (using SAML or WS-Security).

Web services are located by their URL (or URI [12]). If static URLs are used, then Web services have unique persistent names which are part of the XML message, and which are coupled to the invocation mechanism. In this case, a resource descriptor mapping as described for CORBA in section 4.11 is not required. Individual security policies can be applied at a per-URL granularity to the incoming XML message inside the listener. On the downside, if Web services are shut down and restarted (i.e., the state is lost), then clients that were communicating with the old instance need to be able to handle lost state when communicating with the new instance – traditional middleware systems such as CORBA use transient object reference to avoid this problem.

If dynamic URLs are used to publish Web services, and if security policies independent of the Web service life-cycle are required, then some form of the resource descriptor mapping would be useful. The administrator or the application that launches new Web services would need to provide an appropriate resource descriptor for the service, which is stored in a mapping table. At invocation time, the mapping from URL to resource descriptor would be carried out by the listener or the facade.

7.6 Reflective and Adaptive Middleware

By definition, traditional middleware platforms such as CORBA hide the implementation details from the application programmer. Some sources [5] argue that this "black box philosophy" is increasingly becoming untenable, and that the middleware should provide openness, configurability, and reconfigurability, in a principled way through the concept of "reflection". The goal is to provide transparency to the applications that want it and fine-grained control to the applications that need it.

7.6.1 Reflection Terminology

In general, the term "reflection" is used for systems that have the ability to reason about themselves, using some kind of self-representation. This reasoning is done at a "meta level" where certain aspects of the system are "reified" (i.e. represented) as "meta objects". The code dealing with meta objects is called a "meta-program", and the interface to the meta objects is called the "meta-object protocol" [77].

The advantages of separating meta-level code from base-level code are twofold. Firstly, it can help separating concerns, for example, system aspects such as security can be addressed in the meta objects. Secondly, it can provide for more flexibility and better adaptability to changing environments. Reflection can range from inspecting certain system properties to adapting or extending a system or programming model.

Reflection was first introduced by Smith [154] and initially triggered a large body of work to the field of programming language design [77, 168, 3]. Later reflection has been applied to operating systems [178] and distributed systems [102].

Throughout the literature, there are a number of categorisations of reflection, which will be briefly introduced in the following. "Computational reflection" [98] allows systems to reflect about the computational process by representing concepts that are implicit in the programming model, e.g. by representing interfaces or method invocations. "Implementational reflection" [135] allows a program to access aspects of the system implementation, e.g. modify thread scheduling. Computational reflection can be further categorised in a "meta-class model", where a meta-object is attached to the class of an object, so that all instances of a class share the same meta object, and a "specific meta-object model", where individual objects can have their own meta-objects. The "meta-communication model" reifies method invocations [40]. Furthermore, [40] distinguishes "structural reflection", which allows the inspection of the system structure such as classes, inheritance and instantiation relationships, from "behavioural reflection", which is concerned with observing or changing the behaviour of a system, e.g. by redirecting method invocations, dynamically adding classes or types. In addition, reflection can be categorised in "explicit reflection", where the switching to meta-level computations is explicitly done, and "implicit reflection", which means that the system will reflect at certain predefined stages during processing (e.g., at method invocation) [100]. Reflection can occur at compile time and at run-time. "Compile-time reflection" is concerned with the static program structure, while "run-time reflection" allows access to individual object instances. "Introspection" is the ability of the program to observe and reason about its own structure and state, while "intercession" is its ability to change its own state or to alter its own processing.

An important aspect of reflection is that the meta-level (i.e., the representation of its own behaviour which is open to inspection and adaptation) is causally connected to base-level (i.e., the underlying behaviour it describes). "Causally connected" means that changes made to the self-representation are immediately mirrored in the underlying system's actual state and behaviour, and vice-versa. It can therefore be said that a reflective system is one that supports an associated "causally connected self-representation" [99]. Reflection enables both "inspection" and "adaptation" of systems at run-time [30]. Inspection allows the current state of the system to be observed, while adaptation allows the system's behaviour to be altered at run-time to better match the system's current operating environment.

7.6.2 Reflective Middleware

Following from the above, "reflective middleware" can be defined as a middleware system that provides inspection and adaptation of its behaviour through an appropriate causally connected self-representation [30]. Middleware that supports adaptation is also often described by the more fuzzy term "adaptive middleware"

Strictly speaking, standard CORBA is not a reflective architecture because it does not allow the extension of its object model or to modify the entire system behaviour in arbitrary ways. However, CORBA does heavily rely on run-time type information, and thus exhibits aspects of the meta-class model. Every CORBA object supports the operation `get_interface()`, which returns a meta object describing the object's type. This object allows the inspection and modification of the entire type information dynamically. This kind of structural reflection is useful for the Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI), which need to access run-time type information. Runtime type information in CORBA is managed by the Interface Repository (IR), which allows request, inspection, and modifcation of type information.

In addition, CORBA's interceptors constitute a limited form of an implicit meta-communication model with computational behavioural reflection. Programmers can write interceptors and register them with the ORB so that they will be called somewhere on the invocation path. The interceptor is given access to the request or reply meta-object and can inspect or (to some extent) adapt it. This is particularly useful for security, as cryptographic message protection and access control can be carried out inside interceptors.

Many CORBA implementations also provide interfaces to access system level aspects (e.g., threading policy), which constitutes implementational reflection [20]. Furthermore, the Portable Object Adapter (POA) policies offer a limited form of reflection [172], and object-by-value facilities provide object serialisation [78].

In the following, we describe two examples of reflective middleware:

DynamicTAO [81] was developed at the University of Illinois from 1998 to 2000. It extends the TAO system to provide a CORBA compliant reflective ORB that allows inspection and reconfiguration of its internal engine. Reification is achieved through a collection of entities known as component configurators. Component implementations are shipped as dynamically loadable libraries that can be linked to the ORB process at runtime. Component implementations are organized in categories representing different aspects of the ORB's internal engine. The DynamicTAO approach is very low-level and quite powerful in allowing change to the strategies the system adopts, but it is not so simple and powerful when programmers want to extend communication behaviour with features not directly related to strategies already involved in communication.

OpenORB [15] has a component-based model of computation. It adopts structural reflection (i.e. the meta-level exposes the actual implementation). Meta-spaces provide a flexible mechanism to reify and reflect into any system aspect, but it is component-oriented and it behaves similarly to object-oriented models, neglecting to handle communications as a whole. Reflection works on a per-component granularity. Explicit bindings reify communication channels, but not the involved components. They allow the handling of messages as a stream and not in terms of what they represent, which makes them easy to filter or to handle as a whole, but difficult to manipulate them at a higher level. Related to that, the Open-ORB Python Prototype (OOPP) [5] allows programmers to inspect and change the implementation at run-time through a well-defined "open binding" model.

Other reflective middleware includes LegORB [137], OpenCORBA [90], FlexiNet [59], Quarterware [152], mChaRM (multi-Channel Reification Model) [25], and AspectIX [58].

7.6.3 Security and Reflective/Adaptive Middleware

Consider a security-related reflection example that involves a system on a laptop computer that should adapt the used security mechanisms depending on the security requirements of the particular environment. A monitoring component could detect whether the laptop is plugged in a corporate intranet or uses a wireless connection, and apply the appropriate security features. For example, simple access control and light encryption could be sufficient for the intranet, whereas strong access control and encryption should be applied if the laptop were using a wireless link.

In the following, we describe some examples of middleware that support adaptive security.

The dynamicTAO ORB [81] supports on-the-fly adaptation of the security mechanisms. It provides hooks to which "security strategies" can be attached. Security strategies can add message-level interceptors (for encryption and message protection) and request-level interceptors (for access control). One project [95] applied the Cherubim CORBASec security framework [24] in the context of dynamicTAO to implement a reference monitor, audit logging, and caching of security decisions. The Cherubim security framework supports access control by using "active capabilities": Java objects that are protected by digital signatures and encryption and contain the credentials of clients. Before a client can invoke a target, its active capability must be installed into the reference monitor. This can be done explicitly by the client application or transparently to the application by a third party. In dynamicTAO, the reference monitor contacts an "active capability evaluator", which in turn contacts a "policy server" to fetch the active capability from a secure store. After the active capability is evaluated, the access decision is cached in the authorisation cache (it can also be revoked). Moreover, security decisions can be stored in an audit log.

Adaptive middleware can also be used to react to attacks that try to steal network and CPU resources, or which directly attack applications. This was demonstrated in [146] as part of the DARPA Quorum programme. In this project, the adaptive middleware platform was based on the Quality-of-Service (QoS) middleware "Quality Objects" (QuO) [164] and the CORBA ORB TAO [149]. QuO facilitates the creation and integration of distributed applications that can specify their QoS requirement, the system elements that must be monitored and controlled to measure and provide QoS, and the behaviour for adapting to QoS variations at run-time. TAO is a QoS-enabled CORBA ORB that also supports the real-time CORBA specification [117], with enhancements to ensure predictable QoS behaviour for real-time applications. In this project, the adaptive middleware coordinates a number of defence mechanisms, such as access control, packet filtering, intrusion detection, replication management and bandwidth management. For example, applications under attack may be replicated or may migrate to a different host, or network flooding may be prevented by packet filtering. Practical evaluations of this project have shown that, although "defence-enabled" adaptive middleware [146] increases the survival time of distributed applications, it is yet unclear whether the survival time is sufficient in practice.

The need for adaptive security has also been identified for ubiquitous computing environments due to the open interactions [177]; security services should be adaptive in response to the changes in the environments and application requirements.

A number of observations can be made regarding adaptive middleware and adaptive security. Firstly, a security system is generally worthless without assurance, and it is difficult to effectively evaluate systems where security features may be automatically reconfigured on-the-fly. A potential, although expensive workaround would be to evaluate set configurations of the security features and ensure that the middleware can only configure the system into one of these set configurations.

Also, if the security features can be reconfigured automatically by the middleware depending on the environment, the communications partner, and the application, then the adaptation process itself needs to be adequately protected from unauthorised use. This can lead to a "chicken-and-egg" problem. Also, if adaptations that can be triggered over the network (e.g., using special protocols or agents like in dynamicTAO), they need to be adequately protected against misuse. For example, attackers on the network need to be prevented from causing a node to adapt to a lower encryption mechanism so that the traffic can be decrypted more easily.

Another question is who is allowed to trigger the adaptation and under what circumstances. In general, a policy is required that describes when particular security features have to be applied (depending on the security environment), and the trigger conditions in the environment need to be monitored. For example, a monitoring component in a mobile device could monitor the network connection and select the appropriate encryption and access control depending on whether the device is on a corporate LAN or a wireless link. This kind of monitoring could be done by the application or the middleware. Alternatively, the user could manually select the appropriate security features through the application. Again, the adaptation mechanism itself needs to be protected against misuse.

From a more technical perspective, the used security mechanisms (e.g., authentication mechanism) can generally not be changed easily, because often mechanism-specific security attributes (e.g., X.509 identity certificates) are stored in the security policy. In this case, adapting the security mechanism would also require reconfiguration of the security policy. Alternatively, the policy needs to contain abstracted identities, which has several drawbacks described in [87].

Also, adapting security features on-the-fly raises the question what should be done about existing security associations. One radical approach would be to abort all existing security associations and notify the remote side that the security features have changed. A more graceful approach (as implemented in dynamicTAO) would be to allow old connections to use the old security features, while new connections use the new security features. However, this approach has some drawbacks: it requires several security features to run at the same time, and a garbage collection mechanism may be required for old connections.

7.6.4 Middleware and Quality-of-Service

Quality-of-Service (QoS) refers to a technique that uses several different technologies to provide consistent delivery of traffic across a network. The network actively monitors the usage of its available bandwidth and watches for signs of congestion. It proactively generates usage patterns and bandwidth statistics. It also enforces policies relating to the provisioning, use, and distribution of available bandwidth. QoS provides the ability to distinguish between different traffic types for the purpose of resource allocation using bandwidth, latency, jitter, and packet loss as its metrics [4].

Soft QoS is when Class-of-Service (CoS) tags are used without signalling, and the available bandwidth is managed by policies established independently at each intermediate device in the network. This hop-by-hop approach does not give absolute end-to-end guarantees, but attempts to manage congestion based on priority assignment for each CoS. Those CoS classes have meaning only locally to that node.

Hard QoS is when QoS (bandwidth, delay, etc.) can be negotiated (signalling) and guaranteed a specific level of service end-to-end. Guaranteed traffic will not be impacted, regardless of the amount of additional traffic on the network. This is accomplished by establishing QoS requirements when the connection is first established (like a phone call) and by using a connection-oriented technology such as ATM or frame relay. If adding additional traffic to the network will impact existing services, then the new communication will be disallowed (or relegated to a lower priority). This end-to-end approach still uses CoS as a way of grouping sessions together with similar characteristics, but at each hop, the session is checked for usage and the node is forced to abide by the QoS parameters it has negotiated [4].

The OMG has standardised a real-time CORBA specification [117] and issued a proposal for dynamic scheduling [122] to support QoS. Examples of QoS-enabled middleware include the abovementioned “Quality Objects” (QuO) [164] (soft real-time) and the CORBA ORB TAO [149] (hard real-time), as well as the Global Resource Management System (GRMS) [62], the Realize resource management system [75], the QoS provisioning service (QPS) [58], the MULTE ORB [132], Quartz [153], and the Distributed Interactive Multi-Media Architecture (DIMMA) [33].

Security is generally not considered to be part of classical QoS (despite some work that has been done to add security – in particular encryption – as a QoS dimension [65, 70, 29]), and in general QoS support in middleware and in the network does not impact the security of the overall system. The only attack related to QoS is denial-of-service, whereby an attacker interferes with the QoS service to steal resources. Such attacks need to be countered by the QoS service (e.g., by using secure resource reservation protocols) and not by the middleware.

QoS sometimes also includes the aspect that applications or the middleware need to adapt to unavoidable QoS degradation (e.g., by increasing the compression). In this case, (some of) the observations made above regarding adaptation may apply.

7.7 Middleware and Access Control Models

The security literature has come up with a number of different access control models, such as role-based access control, discretionary access control, and mandatory access control. A good introduction to various security models can be found in [50]. This section gives an overview of how easily some of these models can be integrated into middleware.

7.7.1 Discretionary Access Control

Discretionary access control (DAC) is based on user identities and access rules. Users can protect what they own and may grant access to others; the owner can define the type of access given to others [145]. The standard CORBASec access control comprises a discretionary access control matrix [83] that specifies which clients have what access rights on which targets.

7.7.2 Role-Based Access Control

A popular access control model is role-based access control (RBAC). RBAC is a flexible form of access control [129, 131, 113, 140] that by itself it is policy neutral [144].

Traditional RBAC [140] comprises a family of reference models in which permissions are associated with roles, and users are assigned to appropriate roles. RBAC is an important concept for handling large-scale authorisation policies. RBAC includes the capability to establish relationships between roles, between permissions and roles, and between users and roles. There are four established RBAC reference models (sometimes called RBAC96): unrelated roles ($RBAC_0$), role-hierarchies ($RBAC_1$), user and role assignment constraints ($RBAC_2$), and both hierarchies and constraints ($RBAC_3$). These RBAC models support the three security principles of least privilege, separation of duties and data abstraction, in varying degrees.

It has been demonstrated that $RBAC_{0-3}$ can be implemented well with the standard CORBASec access control model [14, 57]. However, some extra functionality (i.e., not specified in CORBASec) is required to support RBAC96. For $RBAC_1$, the authentication services need to support roles and their hierarchies. To support constraints ($RBAC_2$), the authentication infrastructure has to enforce them. In addition, tools to administer user-to-role and role-to-permissions assignments are needed.

Another implementation of RBAC for CORBA [82] uses the OMG Common Secure Interoperability v2 (CSIv2) [120] protocol and the OMG Authorization Token Layer Acquisition Service (ATLAS) [125]. The authorisation layer of the CSIv2 protocol provides two services: identity assertion and authorisation. The approach uses the identity assertion service for a client to activate a role, and the authorisation service to transport the role authorisation. The ATLAS is used to retrieve and deliver role certificates of an authorisation domain. This approach does not allow clients to activate multiple roles simultaneously.

RBAC$_{0-3}$ has also been mapped to the Enterprise Java Beans access model [57], requiring the same extra functionality as for the CORBASec mapping.

Some form of RBAC for Web services can be supported using the Security Assertion Markup Language (SAML) 1.0, an upcoming standard by the Organisation for the Advancement of Structured Information Standards (OASIS) (see section 7.5.2) [46].

The Open Architecture for Secure, Interworking Services (OASIS) [60] is another RBAC architecture used to deliver secure interoperation of independently managed services in open, distributed environments. OASIS roles deviate somewhat from the RBAC96 definition given in [140]. For example, roles are service-specific (i.e., no globally centralised administration), parameterised, and non-hierarchical. Furthermore, roles are activated within sessions, and "appointment" (i.e., the appointer does not need to possess the delegated privileges) is used instead of privilege delegation. A client activates a role by presenting credentials to a service, which supplies a "role membership certificate" (e.g., X.509) that can be used for subsequent requests to use that service. OASIS uses "credential records" and event channels to implement rapid role revocation.

7.7.3 Mandatory Access Control

Mandatory access control (MAC) is "a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorisation (e.g., clearance) of subjects to access information of such sensitivity" [111]. MAC was first formalised by Bell and LaPadula in 1973 [10]. The Bell-LaPadula model supports MAC by determining the access rights from the security levels associated with subjects (called "clearance") and objects (called "classification"). Two rules define MAC: The "simple security property" (read-down) states that a subject can read an object only if the security level of that subject is higher or equal to the security of the object. The "*-property" (write-up) allows writing on an object only if the security level of the object is higher or equal to the security level of the subject. There is also assumption that security labels, once assigned, cannot be changed, which is known as "tranquillity" [143]. In MAC, only administrators and not owners of resources make changes to the policy. MAC is sometimes also referred to as lattice-based access control (LBAC), because it is concerned with enforcing one directional information flow in a lattice of security labels, or as multi-level security.

The expressive power of the CORBA access control mechanisms was analysed in [76], where it was shown to be capable of supporting lattice-based MAC.

It was also demonstrated in [174] that mandatory security policies can be easily implemented by extending the CORBASec access control model. This approach uses client-side (request-level interceptor) access control and adds clearance fields for subjects to the `DomainAccessPolicy` table, classification fields to the `RequiredRights` table, and classifications of information to requests. The prototype implements Bell-LaPadula (and techniques to avoid the over-classification of information).

It has also been demonstrated in [129] that RBAC can be configured to enforce MAC models. Moreover, a number of access constraints have been proposed in [114] that would realise the equivalent of Bell-LaPadula "read-down" and "write-up" rules. The flexibility of RBAC and its ability to enforce MAC policies by suitable configuration of role hierarchies and constraints has also been demonstrated in [144] (where the strong similarity between the concept of a MAC security label and a RBAC role is stressed).

Because it has been illustrated above that RBAC can be easily integrated into middleware (e.g., CORBA/CORBASec), and because MAC can easily be expressed using RBAC, MAC can consequently be easily integrated into middleware such as CORBA/CORBASec.

7.7.4 Clark-Wilson

Furthermore, it has been argued that the Clark-Wilson integrity model [28] can be supported using MAC [91, 150]. We have already shown above that MAC can be expressed well in the context of middleware, and consequently Clark-Wilson can also be integrated into middleware such as CORBA/CORBASec.

7.7.5 Chinese Wall

It has also been demonstrated in [141] and [142] that the Chinese Wall policy, which prevents information flows that cause conflict of interest [19], can be interpreted as just another lattice-based information policy. It can be easily represented within the Bell-LaPadula MAC framework (if careful distinction between users, principals and subjects is maintained), which, as we have already identified above, can be easily integrated into middleware such as CORBA/CORBASec.

7.8 Middleware Design for Security

This chapter concludes with some general observations of how middleware should be designed to accommodate security. To delimit the scope of this discussion, we assume that the term middleware is defined as in definition 1, as the security architecture cannot dictate all features of the architecture because then the resulting system would not have the common features of middleware anymore. Furthermore, this discussion does not cover application layer security, because it is by definition not middleware layer security. This implies that middleware security cannot achieve end-to-end security between application layer entities, as discussed in chapter 4.

From a security perspective, the middleware should provide identifiers for target instances, because these are required by the security service to represent target instances in the access control and auditing policy (we have elaborated in section 4.7.1. that interface-based access control is insufficient for many application scenarios). As described in section 4.6., such "resource descriptors" should uniquely and persistently describe targets at a per-instance granularity, should be coupled to the invocation mechanism, and should not involve any authentication mechanism (because of layering constraints).

However, from a software engineering perspective, persistent object references are problematic because they require clients to be able to handle lost state if a target instance is terminated and re-instantiated. Another reason is that most programming languages do not support different instances of the same interface to have separate identifiers (e.g., object references). For these reasons, CORBA and the other middleware technologies examined in this chapter do not provide instance identifiers, and as a consequence, the security systems do not support instance-based access control either.

The resource descriptor mapping described in section 4.11 is a good compromise between these two approaches - on the one hand it allows object references to be transient, and on the other hand it represents instances by their persistent resource descriptor in the access control and auditing policy. We believe that it would be useful to implement the mapping for the middleware technologies examined in this chapter (EJB, CCM, .NET).

Another important feature of the middleware architecture for security is the concept of interceptors. Interceptors allow the security system to intercept the message path at various points during the invocation. The middleware architecture should cater for a whole range of interceptors at different layers of the middleware, at least at the following eight points: Both the client side and at the target side should have inbound and outbound interceptors. At each of these four points, there should be request-level interceptors that give structured access to the invocation before marshalling (mainly to accommodate the access control reference monitor and auditing), and message-level interceptors that allow modification of the marshalled buffer (mainly for peer authentication and message protection).

In addition to the message header, request-level interceptors should make the invocation parameters available to the security system, so that fine-grained policies can be enforced (however this leads to the question how complex parameters such as serialised objects should be handled).

It would also be useful to have connection level interceptors that allow the security service to obtain information about the connection established by the underlying transport. For example, a different security policy could be applied depending on the SSL/TLS connections to different hosts or networks.

If other services such as transaction processing are also integrated into the message path through interceptors (e.g., in CORBA), then there needs to be a stack of interceptors at each interception point. It might be necessary to intercept the traffic (at the request level) before the interceptor invokes other services, as well as afterwards. This results in a whole stack of interceptors that call the security system before and after other services are invoked by the interceptor.

The communications protocols used by the middleware also need to cater for security. In particular, protocols should allow security tokens to be transferred embedded in normal messages (e.g., in the protocol service context) to support the transfer of identity certificates, authorisation tokens, and delegation tokens. For example, CORBA's CSIv2 protocol allows the transfer of tokens in the IIOP service context, and SAML allows the transfer of authorisation tokens embedded in SOAP messages.

The middleware should also allow for extra messages (i.e., not sent together with normal requests and replies) to be sent from within the interceptors. This way, the security system can implement security protocols that require extra messages, such as challenge-response protocols and mechanism negotiation protocols.

In addition, the middleware should support the flexible replacement of the underlying transport mechanism (so-called "pluggable protocols") to allow the integration of security protocols such as SSL/TLS. We have already pointed out earlier in this dissertation that the use of secure communications protocols cannot protect local communications, which renders the security system ineffective for invocations that do not go across the network.

Moreover, object references should allow the inclusion of security information, such as the security mechanisms and cryptographic algorithms supported by the target side (as done in CORBASec).

Some middleware architectures support adaptation/reflection. Although this is a useful feature that increases flexibility (and can improve survivability), it severely inhibits security. This is because it is hard to achieve an acceptable level of assurance (i.e., guarantees about the security of the system) for a system that is built on top of a dynamically changing middleware layer. And since security is generally not very useful without assurance, adaptation of the middleware should be avoided.

To separate the duties of the application developer from the duties of the security engineer and administrator, the software development and deployment process should allow the security metadata (such as required roles or rights) to be supplied in separate files and not in the actual application code (cf. EJB or CCM deployment descriptors). This should allow the deployer to specify the security policy for code that has been written by the application provider, and frees the application provider from coding security into the application.

Although not related to this discussion, it is also worth noting that the middleware architecture as such is not the only constraint for middleware security. Often the underlying security mechanisms introduce additional limitations. For example, SSL/TLS does not provide a rich set of security features and is a socket-to-socket protocol, which restricts the functionality of the middleware security architecture.

7.9 Summary

This chapter examines middleware technologies that have evolved since the specification of CORBA and CORBASec. It gives an overview of Enterprise Java Beans, the CORBA Component Model, Microsoft .NET, and XML Web services, and discusses the relevance of the resource descriptor mapping introduced in chapter 4. Furthermore, reflection and adaptation in the context of middleware, as well as their impact on middleware security, is examined. Next, the implementation of several access control models in the context of middleware is analysed. Finally, the chapter concludes with some general observations how middleware should be structured to accommodate security.

Chapter 8

Conclusion

This dissertation analyses the difficulties of fitting security functionality into a layered middleware architecture (consisting of application layer, middleware layer, and underlying technology layer) that is designed in accordance with a number of design requirements. The three most important requirements are: firstly, from an application perspective the object-oriented programming model should be preserved, which involves automatically taking care of all communication details (automation), while at the same time hiding all underlying communications tasks from the application (abstraction). Secondly, application code should be portable across different implementations of the underlying middleware and underlying technology (portability). Thirdly, it should be possible to replace underlying technology without affecting any of the higher layers (flexibility). Additional requirements include interoperability across different middleware implementations, and support for large-scale systems. Throughout this dissertation, the Common Object Request Broker Architecture (CORBA) was used as a reference architecture for such a middleware.

The middleware security architecture should provide the basic security functions authentication, message protection, access control, audit (and optionally non-repudiation). In this dissertation, the CORBA Security Services (CORBASec) specification is used as a reference technology for such a middleware security architecture.

In order to fit into the middleware architecture, the middleware security architecture has to be designed to preserve the abovementioned middleware design requirements. To hide the security architecture (abstraction), and to support automatic security policy evaluation and enforcement (automation) as well as application portability, all the security features have to reside below the application layer. In addition, the underlying security technology (i.e., authentication and message protection) has to reside underneath the middleware layer to allow flexible replacement of middleware-unspecific underlying security technology (flexibility). As a result, the middleware security architecture tries to solve the middleware security problem on several layers:

- Authentication and message protection are done below the middleware layer by security mechanisms such as SSL, Kerberos, or SESAME. To allow for flexibility, the security architecture should ideally abstract their exact nature from the higher layers.
- The two other main functional components, access control and auditing, are implemented in the middleware layer and – because there is no suitable notion available in

the middleware layer – rely on the security attributes established by the underlying security technology, in particular the authenticated identities for the client and the target.

Unfortunately, the granularity and semantics of these identities do not match with the representations required to express effective middleware layer access control and audit policies. In particular, the authentication mechanism cannot provide useful identities at the granularity of individual targets, it can only authenticate the middleware component that resides below the target object. This is because the security association ends at the middleware component, and not at the individual target.

As a result, the concept of middleware layer separation from the underlying security technology breaks: introducing the middleware layer not only separates the application from the underlying network, it also separates the security problem from the security solution. The generic nature of our discussion and the used models suggest that this problem is common to many middleware technologies, not just CORBA/CORBASec.

In this dissertation, we describe an approach that solves this problem on the target side. We incrementally construct a corresponding "N-to-1 Middleware Security Model with Local Target Descriptors" that illustrates how authenticated identities, principals, middleware, clients and targets, and descriptors are related. All these terms are explicitly defined to avoid confusion with the often conflicting terminology used throughout the information security literature.

Our approach allows the expression of individual access control (and audit) policies on a per-target basis. We introduce the notion of descriptors that express individual targets and have to satisfy the following four properties:

- Descriptors should be unique within the scope of a middleware component and describe targets on a per-target granularity
- Descriptors should be independent of the object lifecycle of the targets (i.e., persistent)
- Descriptors should not involve any authentication mechanism
- Descriptors should be coupled to the addressing mechanism used by the middleware to forward a request to the targets

We then analyse the usability of the two descriptor options available to describe targets: the interface type, and the instance identification in the request header. Unfortunately the target interface does not fulfil the uniqueness and coupling properties, and the target instance identifier does not fulfil the persistency property.

As a result, we introduce the notion of "resources", which are defined as services offered to a client by a target instance. The instance that provides a resource can change over time, but it will always be protected by the same access rule. Descriptors for such resources would fulfil all properties outlined above, but they are not directly available in the middleware architecture.

To solve this problem, we propose a mapping mechanism that ties the instance identification to a persistent resource descriptor that is supplied by the user or the server when the target servant is created. At configuration time, this descriptor is stored in a mapping table together with the instance information that is part of the object reference. Access (and audit) policies are expressed for individual targets using the resource descriptors.

At invocation time, the access control and audit functions in the middleware layer can obtain the resource descriptor associated with the instance information taken from the incoming request, and enforce the access rule for the invoked resource.

Our proof-of-concept implementation shows that this approach is useable in practice, even for complex architectures such as CORBA/CORBASec. The implementation, which is part of the MICOSec Open Source CORBA security services implementation, is inspired by an early draft of the OMG's Security Domain Membership Management Service submission but contains a number of improvements. In particular, the POA name is used instead of the proposed POA reference, which allows the persistent configuration of the mapping table for security-unaware applications. MICOSec also supports the dynamic mapping configuration for security-aware applications.

Our analysis of the security approaches of several middleware technologies that have emerged since the specification of CORBA and CORBASec shows that these technologies lack instance-based access control and would therefore also benefit from the resource descriptor mapping.

8.1 Further Work

Further work could proceed in several directions. Firstly, it could be tried to improve the quality of the information used for the resource descriptor mapping, to allow the specification of more expressive access policies. Would it be feasible to also consider the content of invocation parameters by intercepting the invocation at the skeleton (e.g., by using automatic code re-writing to achieve reflection)? Could this parameter information first be mapped into some generic format, and then be used in the policy? Or maybe an extended architecture could be designed that establishes (additional?) identities at a client/target granularity?

Another direction for further work would be to implement the resource descriptor mapping for middleware technologies other than CORBA, in particular component architectures such as the OMG CORBA Components Model (CCM) [126] and Enterprise Java Beans (EJB) [158]. Maybe the container layer can provide (additional?) more expressive security attributes for an improved resource descriptor mapping.

Although not quite related to this dissertation, a further unsolved problem of the middleware security architecture described is that authenticated identities are only available if the invocation involves network communications between two separate middleware components (what is the meaning of "separate" in the context of dynamically linked libraries?).

List of Publications

1. U. Lang and R. Schreiner. *Developing Secure Distributed Systems with CORBA* (ISBN 1580532950), hardcover 332 pages, Artech House Publishers, Boston, MA, February 2002.

2. U. Lang, D. Gollmann, and R. Schreiner. Verifiable Identifiers in Middleware Security. *17th Annual Computer Security Applications Conference (ACSAC) Proceedings*, pages 450–459, December 2001.

3. U. Lang, D. Gollmann, and R. Schreiner. Cryptography and Middleware Security. *Information and Communications Security (ICICS) 2001 Proceedings LNCS 2229*, pages 408–418, November 2001.

4. R. Schreiner and U. Lang. CORBA as a Secure Platform for Mobile Applications. *EURESCOM 3G Technologies and Applications 2001 Proceedings*, November 2001.

5. U. Lang and R. Schreiner. MICOSec: An Open Source Implementation of the CORBA Security Services. *Information Security Solutions Europe (ISSE) Conference 2001 Proceedings*, September 2001.

6. R. Schreiner and U. Lang. Eine Open-Source Implementierung der CORBA Sicherheitsdienste (An Open Source Implementation of the CORBA Security Services). *7. Deutscher IT-Sicherheitskongress (7. German IT Security Conference) Proceedings*, May 2001.

7. M. Schumacher, U. Lang, A. Alireza, M. Padelis, R. Schreiner. The Challenges of CORBA Security. *GI-Workshop Proceeedings "Sicherheit in Mediendaten" (Security in Media Data)*, Berlin, September 2000.

8. U. Lang and R. Schreiner. Flexibility and Interoperability in CORBA Security. *Electronic Notes in Theoretical Computer Science*, Vol. 32, Elsevier, Netherlands, February 2000.

9. U. Lang. CORBA Security on the Web. *Future Generation Computer Systems, Special Issue: Security on the Web*, Elsevier, Netherlands, August 1999.

10. U. Lang and R. Schreiner. Sicherheit in CORBA-Systemen (Security in CORBA Systems). *iX Magazin*, Heinz Heise Verlag, Hannover, Germany, October 1998.

11. U. Lang and D. Gollmann. Secure CORBA based Electronic Commerce Systems. *Elsevier/Zergo Information Security Technical Report*, Vol. 3, No. 2, July 1998.

12. U. Lang. CORBA Security. University of London (Royal Holloway), M.Sc. Dissertation, September 1997.

List of Presentations

1. Enforcement of Enterprise-Wide Security Policies with CORBASec. *DOCsec 2002 Workshop, Baltimore, MD, USA, March 2002.*
2. Middleware Security – Current Research and Future Work. *Security Seminar, Cambridge University Computer Laboratory, Cambridge, UK, March 2002.*
3. Verifiable Identifiers in Middleware Security. *ACSAC (Annual Computer Security Application Conference) 2001, New Orleans, LA, USA, December 2001.*
4. Cryptography and Middleware Security. *ICICS2001 (Third International Conference on Information and Communications Security), Xi'an, China, November 2001.*
5. MICOSec: An Open Source Implementation of the CORBA Security Services. *2001 Information Security Solutions Europe (ISSE) Conference, London, UK, September 2001.*
6. MICOSec: CORBA Security Reality Check. *DOCsec 2001 Workshop, Annapolis, MD, USA, March 2001.*
7. Security in Distributed Systems. *Lecture for M.Sc. Information Security, University of London (Royal Holloway), UK, November 2000.*
8. Security Attributes in CORBA. *Security Seminar, Cambridge University Computer Laboratory, Cambridge, UK, November 2000.*
9. CORBA Security in a Telecommunications Environment. *DOCsec 2000 Workshop, Boston, MA, USA, April 2000.*
10. Security in Distributed Systems. *Lecture for M.Sc. Information Security, University of London (Royal Holloway), UK, December 1999.*
11. Why CORBA Security (Still) Fails. *DERA Security Workshop 1999, Malvern, UK, December 1999.*
12. CORBA Security in a Large Banking Environment. *DOCsec 1999 Workshop, Baltimore, MD, USA, July 1999.*
13. Distributed Access Control. *DERA, Malvern, UK, February 1999.*
14. The Current State of CORBA Security Implementations. *DERA Workshop on Secure Architectures, University of London (Royal Holloway), UK, December 1998.*
15. Security in Distributed Systems. *Lecture for M.Sc. in Information Security, University of London (Royal Holloway), UK, November 1998.*
16. CORBA, CORBA Security, and CORBA Security in Practice. *Security Seminar, Cambridge University Computer Laboratory, Cambridge, UK, April 1998.*

Bibliography

[1] C. Adams. The Simple Public-Key GSS-API Mechanism (RFC 2024), 1996.

[2] Adobe Systems. Acrobat 5 Overview. Adobe Systems Inc., March 2001.

[3] G. Agha. The structure and semantics of actor languages. In *Proceedings of REX School/Workshop Foundations of Object-Oriented Languages (J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds.), vol. 489 of Lecture Notes in Computer Science, Noordwijkerhout, The Netherlands, Springer-Verlag*, pages 1–59, May/June 1990.

[4] Alcatel Internetworking. Alcatel Enterprise – Quality of Service (QoS) – An Alcatel Executive Briefing, February 2002.

[5] A. Andersen, G.S. Blair, and F. Eliassen. OOPP: A reflective component-based middleware. NIK 2000, Bodø, Norway, November 2000.

[6] R. Anderson. *Security Engineering – A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, New York, 2001.

[7] J. Bacon and K. Moody. Toward Open, Secure, Widely Distributed Services. *Communications of the ACM*, Vol. 45, No. 6, June 2002.

[8] M. Barnett. .NET Remoting Security Solution, Part 1: Microsoft.Samples.Security.SSPI Assembly, August 2002.

[9] M. Barnett. .NET Remoting Security Solution, Part 2: Microsoft.Samples.Runtime.Remoting.Security Assembly, August 2002.

[10] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. *MITRE Report MTR*, 2547, v2, 1973.

[11] T. Bellwood, L. Clément, D. Ehnebuske, A. Hately, M. Hondo, Y. Leng, K. Januszewski, S. Lee, B. McKee, J. Munter, and C. von Riegen et al. UDDI Version 3.0 Published Specification, July 2002.

[12] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, August 1998.

[13] K. Beznosov. *Engineering Access Control for Distributed Enterprise Applications.* PhD thesis, Florida International University, Miami, FL, 2000.

[14] K. Beznosov and Y. Deng. A Framework for Implementing Role-based Access Control Using CORBA Security Service. Fourth ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, October 1999.

[15] G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The Design and Implementation of Open ORB 2. *IEEE Distributed Systems Online*, Vol. 2, No. 6, 2001. http://dsonline.computer.org/0106/features/bla0106_print.htm.

[16] D. Box. Understanding GXA. Microsoft Developer Network, July 2002.

[17] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1, May 2000.

[18] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, October 2000.

[19] D.F.C. Brewer and M.J. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, pp. 206-214*, 1989.

[20] G. Brose. Reflection in Java, CORBA und JacORB. Java-Informations-Tage, Frankfurt/Main, 1998.

[21] G. Brose. *Access Control Management in Distributed Object Systems*. PhD thesis, Freie Universität Berlin, Berlin, Germany, 2001.

[22] A. Brown, M. Fuchs, J. Robie, and P. Wadler (eds.). XML Schema: Formal Description. W3C Working Draft, September 2001.

[23] T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, Reading, MA, 1996.

[24] R. Campbell and T. Quian. Dynamic Agent-based Security Architecture for Mobile Computers. In *Proceedings of the Second International Conference on Parallel and Distributed Computing and Networks (PDCN'98), Australia*, pages 291–299, December 1988.

[25] W. Cazzola and M. Ancona. mChaRM: A Reflective Middleware for Communications-based Reflection. Technical Report DISI-TR-00-09, DISI, Universita defli Studi di Milano, Milan, May 2000.

[26] W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet Security – Repelling the Wily Hacker*. Addison Wesley, Reading, MA, 1994.

[27] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, March 2001.

[28] D.R. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, CA*, pages 184–194, 1987.

[29] COACH Consortium. Component Based Open Source Architecture for Distributed Telecom Applications, EU IST Programme (Project IST-2001-34445, 1 April 2002 to 31 March 2004), January 2003. (http://www.ist-coach.org).

[30] G. Coulson. What is Reflective Middleware? *IEEE Distributed Systems Online*, Vol.2, No. 8, 2001. (http://dsonline.computer.org/middleware/RMarticle1.htm).

[31] M. A. Davidson, J. Heimann, P. Needham, and K. Browder. Oracle 9i Database Security for eBusiness – An Oracle White Paper. Oracle Corporation, June 2001.

[32] T. Dierks and E. Rescorla. The TLS Protocol Version 1.1 (RFC 2246), October 2002. Internet Engineering Task Force.

[33] DIMMA Team. DIMMA Design and Implementation, ANSA Phase III. Technical Report APM.2063.01, ANSA, Cambridge, UK, September 1997.

[34] B. Shannon (ed.). Java 2 Platform Enterprise Edition Specification, v1.4, Proposed Final Draft, August 2002.

[35] C. Kaler (ed.). Web Services Security (WS-Security), Version 1.0 05, April 2002.

[36] L. Kristiansen (editor). TINA-C Service Architecture Version: 5.0. TINA-C, June 1997. (www.tinac.com).

[37] U. Erlingsson and F. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *New Security Paradigms Workshop Proceedings*, pages 87 – 95, Ontario, Canada, September 1999.

[38] G. Della-Libera et al. Security in a Web Services World: A Proposed Architecture and Roadmap. A Joint White Paper from IBM Corporation and Microsoft Corporation, V1.0, April 2002.

[39] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *IEEE Security and Privacy Proceedings*, pages 32 – 45, Oakland, CA, May 1999.

[40] J. Ferber. Computational Reflection in Class based Object Oriented Languages. In *Proceedings OOPSLA 1989, SIGPLAN Notices, ACM Press*, pages 317–326, 1989.

[41] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1 (RFC 2616), June 1999.

[42] C.A. Fonseca. Extend JAAS for class instance-level authorization. *IBM DeveloperWorks*, April 2002.

[43] J. Fontana. Web Services: Where middleware and XML converge. *Network World*, September 2001.

[44] J. Fontana. Securing Web services. *Network World*, September 2002.

[45] J. Fontana. Top Web services worry: Security. *Network World*, January 2002.

[46] J. Fontana. XML-based security protocol wins approval from OASIS. *Network World*, November 2002.

[47] W. Ford. *Computer Communications Security – Principles, Standard Protocols and Techniques.* Prentice Hall PTR, New Jersey, 1994.

[48] Foundstone/CORE Security Technologies. Security in the Microsoft .NET Framework, 2002.

[49] S. Garfinkel and G. Spafford. *Practical Unix &Internet Security, 2nd edition.* O'Reilly & Associates, Inc., Sebastopol, 1996.

[50] D. Gollmann. *Computer Security.* John Wiley & Sons, UK, 1999.

[51] L. Gong. *Inside Java 2 Platform Security.* Addison Wesley, Reading, MA, June 1999.

[52] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of hte New Security Architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California*, December 1997.

[53] S.R. Gopalan. The CORBA Component Model (CCM), 1999. (http://www.exec-pc.com/∼gopalan/CORBA/ccm.html).

[54] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition (The Java Series).* Addison Wesley, Reading, MA, June 2000.

[55] GotDotNet. About .NET Security, 2002. (http://www.gotdotnet.com/te-am/clr/about_security.aspx).

[56] Halcyon Software. iNET A Java Implementation of Microsoft .NET Framework, White Paper, 2002.

[57] B. Hartman, D.J. Flinn, and K. Beznosov. *Enterprise Security with EJB and CORBA.* John Wiley & Sons, New York, NY, 2001.

[58] F.J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. AspectIX: a quality-aware, object-based middleware architecture. In *Proceedings of the 3rd IFIP International Conference on Distributed Applications and Interoperable Systems – DAIS, Krakow, Poland*, September 2001.

[59] R. Hayton. FlexiNet Open ORB Framework. Technical Report 2047.01.00, APM Ltd., Cambridge, UK, October 1997.

[60] R. Hayton, J. Bacon, and K. Moody. OASIS: Access Control in an Open, Distributed Environment. In *Proceedings IEEE Symposium on Security and Privacy, Oakland CA*, pages 3–14, May 1998.

[61] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++.* Addison-Wesley, Reading, MA, 1999.

[62] J. Huang and Y. Wang et al. On Developing Distributed Middleware Services for QoS- and Criticality-Based Resource Negotiation and Adaptation. *Journal of Real-Time Systems (Special Issue on Operating Systems and Services)*, 1998.

[63] P. Humenn. Summary of MDI Discussion. OMG SecSIG Mailinglist, March 1999.

[64] P. Humenn. A Language for Access Control in CORBA Security. In *Fourth Workshop on Distributed Object Computing Security (DOCsec) Online Proceedings*, Boston, MA, April 2000.

[65] C. Irvine and T. Levin. Quality of Security Service. 23rd National Information Systems Security Conference, Baltimore Convention Center, Baltimore, MD, October 2000.

[66] ISO 7498-2. Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 2: Security Architecture. International Organization for Standardization, 1989.

[67] ISO/IEC 10181-1. Information Technology – Open Systems Interconnection – Security Frameworks in Open Systems: Overview. International Organization for Standardization/International Engineering Consortium, 1997.

[68] ISO/IEC 10181-4. Information Technology – Open Systems Interconnection – Security Frameworks in Open Systems: Non-Repudiation Framework. International Organization for Standardization/International Engineering Consortium, 1997.

[69] ISO/IEC 8879. Information processing – Text and office systems – Standard Generalized Markup Language (SGML). International Organization for Standardization/International Engineering Consortium, 1986.

[70] ISO/IEC JTC1/SC21 N QoS1. Open System Interconnection, Data Management and Open Distributed Processing, Working Draft. International Organization for Standardization/International Engineering Consortium, July 1997.

[71] J. Gosling and F. Yellin and Java Team. Java API Documentation Version 1.0.2. Sun Microsystems, Inc., 1996.

[72] Java Team. Java 2 SDK Documentation. Sun Microsystems, Inc., 1996–1999.

[73] Java Team. JDK 1.1.8 Documentation. Sun Microsystems, Inc., 1996–1999.

[74] P Kaijser, T Parker, and D Pinkas. SESAME: The solution to security for open distributed systems. *Computer Communications, Vol. 17, No. 7*, pages 501 – 518, 1994.

[75] V. Kalogeraki and P.M. Melliar-Smitm et al. Dynamic Scheduling of Distributed Method Invocations. In *Proceedingsof the 21st IEEE Real-Time Systems Symposium, Orlando, Florida*, November 2000.

[76] G. Karjoth. Authorization in CORBA Security. *Journal of Computer Security, Vol. 8, Nos. 2, 3*, pages 89 – 108, October 2000.

[77] G. Kiczales, J. des Rivires, and D. G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA, USA, 1991.

[78] M.O. Killijian and J.C. Fabre. Implementing a Reflective Fault-Tolerant CORBA System. *19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), Nürnberg, Germany*, October 2000.

[79] M. Kirtland. A Platform for Web Services. Microsoft Developer Network, January 2001.

[80] J. Kohl and C. Neumann. The Kerberos Network Authentication Service V5 (RFC 1510), 1993.

[81] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L.C. Magalhaes, and R.H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), New York, April 2000.

[82] C.J. Kuo and P. Humenn. Dynamically Authorized Role-Based Access Control for Secure Distributed Computation. Sixth Annual Workshop On Distributed Objects and Components Security (DOCSec2002), Baltimore, MD, March 2002.

[83] B. Lampson. Protection. *ACM Operation Systems Review Vol. 8, No. 1 (Reprint from: Fifth Princeton Symposium on Information Sciences and Systems, pp. 437 – 443, Princeton University, March 1971)*, pages 18 – 24, January 1974.

[84] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems, Vol. 10, No. 4*, pages 265 – 310, November 1992.

[85] T. Landgrave. Planning Web Services Security. ZDNet Australia, October 2002.

[86] U. Lang, D. Gollmann, and R. Schreiner. Cryptography and Middleware Security. In *Information and Communications Security (ICICS) Proceedings, LNCS 2229*, pages 408 – 418, Xi'an, China, November 2001. Springer, Berlin, Germany.

[87] U. Lang, D. Gollmann, and R. Schreiner. Verifiable Identifiers in Middleware Security. In *17th Annual Computer Security Applications Conference (ACSAC) Proceedings*, pages 450 – 459. IEEE Press, December 2001.

[88] U. Lang and R. Schreiner. Flexibility and Interoperability in CORBA Security. *Electronic Notes in Theoretical Computer Science*, 32, February 2000. Elsevier, Netherlands.

[89] U. Lang and R. Schreiner. *Developing Secure Distributed Systems with CORBA*. Artech House, February 2002.

[90] T. Ledoux. OpenCorba: A Reflective Open Broker. *Lecture Notes in Computer Science*, Vol. 1616, pages 197–215, 1999.

[91] T.M.P Lee. Using mandatory integrity to enforce "commercial" security. In *Proceedings IEEE Computer Society Symposium on Security and Privacy, Oakland, CA*, pages 140–146, May 1988.

[92] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. Network Working Group, March 1996.

[93] D. Levine, D.C. Schmidt, and N. Wang. Optimizing the CORBA Component Model for High-performance and Real-time Applications. Middleware 2000 Conference, IFIP/ACM, Palisades, New York, 2000.

[94] J. Linn. Generic Security Service Application Program Interface, Version 2 (RFC 2078), 1997.

[95] P. Liu. The Design and Implementation of a Reference Monitor for the 2K Operating System (Master's thesis). Department of Computer Science, University of Illinois at Urbana-Champaign, July 1999.

[96] J. Lowy. Unify the Role-Based Security Models for Enterprise and Application Domains with .NET. MSDN Magazine, May 2002.

[97] S. MacRoibeaird. Developer Connection: Universal Description, Discovery & Integration (UDDI) An Executive Summary. Sun Microsystems XML Technology Center, 2002.

[98] P. Maes. Computational Reflection. Technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, Belgium, 1987.

[99] P. Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA 1987, 22, SIGPLAN Notices, ACM Press*, pages 147–155, 1987.

[100] P. Maes. *Issues in computational reflection. In: P. Maes and D. Nardi (eds.): Meta-Level Architectures and Reflection.* Elsevier, Netherlands, 1988.

[101] V. Matena and B. Stearns. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform.* Addison Wesley Longman, Inc., Boston, MA, December 2000.

[102] J. McAffer. Meta-level architecture support for distributed objects. In *Proceedings of Reflection'96 (G. Kiczales, ed.), San Francisco*, pages 39–62, 1996.

[103] MICO Project. MICO is CORBA, 2002. (www.mico.org).

[104] MICOSec Project. Open Source CORBA Security Services Project (for MICO), 2002. (www.micosec.org).

[105] Microsoft Corporation. The Security Support Provider Interface. White Paper, 1999.

[106] Microsoft Corporation. A Guide to Reviewing the Microsoft .NET Framework: A Platform for Rapidly Building and Deploying XML Web Services and Applications to Solve Today's Business Challenges. Microsoft Corp, Redmond, WA, 2001. (msdn.microsoft.com/netframework/).

[107] Microsoft Corporation. Internet Explorer 6 Technical Overview. Microsoft Corp., Redmond, WA, August 2001. (msdn.microsoft.com/netframework/).

[108] Microsoft Corporation. Microsoft .NET Framework Security Overview. Visual Studio .NET Technical Resources, 2002.

[109] Microsoft Corporation. The Microsoft .NET homepage, January 2003. (http://www.microsoft.com/net).

[110] Mono Project. Mono Project Website, January 2003. (http://www.go-mono.com).

[111] National Computer Security Center. Department of Defense Trusted Computer Security Evaluation Criteria. DOD 5200.28-STD, December 1985.

[112] D. Neimke. Implementing Role-Based Security with ASP.NET, Part 2, December 2001. (http://www.4guysfromrolla.com/webtech/121901-1.2.shtml).

[113] M. Nyanchama and S. Osborn. Role-Based Security: Pros, Cons & Some Research Directions. *ACM SIGSAC Review*, Vol. 2, No. 2, pages 11–17, June 1993.

[114] M. Nyanchama and S. Osborn. Modeling mandatory access control in role-based security systems. In *Proceedings of the IFIP Working Group 11.3 Working Conference on Database Security. Elsevier, North-Holland, Amsterdam, Netherlands*, pages 37–56, 1996.

[115] P. Obermeyer and J. Hawkins. Microsoft .NET Remoting: A Technical Overview. MSDN Library, July 2001.

[116] Object Management Group. A Discussion of the Object Management Architecture, rev. 3.0. Needham, MA, June 1997.

[117] Object Management Group. Real-Time CORBA 1.0 Adopted Specification. Needham, MA, June 1999. (document ptc/99-06-02).

[118] Object Management Group. Common Object Request Broker Architecture, v2.3. Needham, MA, 2000. (document: formal/98-12-01).

[119] Object Management Group. Common Object Request Broker: Architecture and Specification, v2.5. Needham, MA, 2001. (documents: formal/01-09-01, formal/01-09-34).

[120] Object Management Group. Common Secure Interoperability Version 2 Specification. Needham, MA, 2001. (document: formal/01-12-30).

[121] Object Management Group. CORBA Security Services, v1.7. Needham, MA, 2001. (document: formal/01-03-08).

[122] Object Management Group. Dynamic Scheduling, Joint Final Submission. Needham, MA, April 2001. (document orbos/01-04-01).

[123] Object Management Group. Resource Access Decision Facility Specification v1.0. Needham, MA, April 2001. (document: formal/01-04-01).

[124] Object Management Group. Security Domain Membership Management Service (final submission). Needham, MA, July 2001. (document: orbos/2001-07-20).

[125] Object Management Group. The Authorization Token Layer Acquisition Service Specification. Needham, MA, October 2001. (document ptc/2001-10-22).

[126] Object Management Group. CORBA Component Model v3.0. Needham, MA, June 2002. (document: formal/2002-06-65).

[127] Open Group. Distributed Computing Environment Overview. Woburn, MA, December 1996. (www.opengroup.org/dce/tog-dce-pd-1296.htm).

[128] Open Group. Press Release: CORBA Open Brand, June 1999. (www.opengroup.org/press/7jun99_a.htm).

[129] S. Osborn, R. Sandhu, and Q. Nunawer. Configuring Role-Based Access Control To Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security*, Vol. 3, No. 2, 2000.

[130] Parlay Group. Open Service Access; Application Programming Interface; Part 1: Overview, Final draft, ETSI Standard ES 201 915-1 V1.1.1. Parlay Group, December 2001. (www.parlay.org).

[131] C.E. Philips, T.C. Ting, and S.A. Demurjian. Information Sharing and Security in Dynamic Coalitions. Seventh ACM Symposium on Access Control Models and Technologies, Monterey, California, USA, 2002.

[132] T. Plagemann, F. Eliassen, B. Hafskjold, T. Kristensen, R.H. Macdonald, and H.O. Rafaelsen. Flexible and Extensible QoS Management for Adaptive Middleware. International Workshop on Protocols for Multimedia Systems (PROMS 2000), Cracow, Poland, October 2000.

[133] A. Pruneda. Windows Media Technologies Using Windows Media Rights Manager to Protect and Distribute Digital Media. *MSDN magazine*, page 5, March 2001.

[134] D. Raggett, A. Le Hors, and I. Jacobs (World Wide Web Consortium). HTML 4.0 Specification, April 1998.

[135] R. Rao. Implementational Reflection in Silica. In *Proceedings ECOOP 1991, LNCS, Springer, Berlin*, pages 251–267, 1991.

[136] T. Riechmann. *Sicherheit in verteilten, objektorientierten Systemen (Security in Distributed, Object-Oriented Systems)*. PhD thesis, Univeristy of Erlangen-Nürnberg, Erlangen, Germany, 1999.

[137] M. Roman, D. Mickunas, F. Kon, and R. Campbell. LegORB and Ubiquitous CORBA. Workshop on Reflective Middleware (RM'2000), IBM Palisades Executive Conference Center, New York, USA, April 2000.

[138] K. Römer, A. Puder, and F. Pilhofer. *MICO is CORBA, An Open Source CORBA 2.3 Implementation.* Morgan Kaufman, 1999.

[139] J. Rushby and B. Randell. A Distributed Secure System. *IEEE Computer, Vol. 16, No. 7*, pages 55 – 67, 1983.

[140] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. *IEEE Computer, Vol. 29, No. 8*, pages 38 – 47, 1996.

[141] R.S. Sandhu. A Lattice Interpretation of the Chinese Wall Policy. In *Proceedings 15th NIST-NCSC National Computer Security Conference, Baltimore, MD*, pages 329–339, 1987.

[142] R.S. Sandhu. Lattice-Based Enforcement of Chinese Walls. *Computers & Security*, Vol. 11, No. 8, pages 753–763, December 1992.

[143] R.S. Sandhu. Lattice-based access control models. *IEEE Computer*, Vol. 26, No. 11, pages 9–19, 1993.

[144] R.S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In *Proceedings of the Fourth European Symposium on Research in Computer Security (ESORICS96, Rome, Italy), E. Bertino, H. Kurth, G. Martella, E.Montoliva (eds.) Springer, New York, NY*, pages 65–79, September 1996.

[145] R.S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications*, Vol. 32, No. 9, pages 40–48, 1994.

[146] R. Schantz, F. Webber, P. Pal, J. Loyall, and D.C. Schmidt. Protecting Applications Against Malice with Adaptive Middleware. Certification and Security in E-Services stream of the 17th IFIP World Computer Congress, Montreal, Canada, August 2002.

[147] R. Scheier. Scheier's Security Product Roundup – Web Services Require New Approach to Security, May 2002. (http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci823485,00.html).

[148] J. Schimmel. CORBA Component Model – Seminarausarbeitung Kommunikation in verteilten Systemen. TU Darmstadt, July 2000.

[149] D. Schmidt, D. Levine, and S. Mungee. The design of the TAO realtime Object Request Broker. *Computer Communications*, Vol. 21, No. 4, April 1998.

[150] W.R. Schockley. Implementing the Clark/Wilson integrity policy using current technology. NIST-NCSC National Computer Security Conference, 1988.

[151] R. Schreiner. CORBA Firwewalls. ObjectSecurity Ltd., August 1999.

[152] A. Singhai, A. Sane, and B. Campbell. Quarterware for Middleware. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS'98), Amsterdam*, pages 192–201, May 1998.

[153] F. Siqueira, V. Cahill, and V. Quartz. A QoS Architecture for Open Systems. 20th International Conference on Distributed Computing Systems (ICDCS'00), Teipei, Taiwan, April 2000.

[154] B.C. Smith. Reflection and Semantics in a Procedural Language. Technical Report MIT/LCS/TR-272, MIT, Cambridge, MA, 1982.

[155] Sourceforge. Introducing OCL, August 2002. (http://foundries.sourceforge.net/article.pl?sid=02/04/15/0956238).

[156] Sun Microsystems. RPC: Remote Procedure Call, Protocol Specification, Version 2 (RFC 1057), June 1988.

[157] Sun Microsystems. JavaTM 2 SDK, Standard Edition Documentation Version 1.4.1, 2002. (http://java.sun.com/j2se/1.4.1/docs/index.html).

[158] Sun Microsystems. Enterprise JavaBeans 2.1, January 2003. (http://java.sun.com/products/ejb/docs.html).

[159] Sun Microsystems. The J2EE homepage, January 2003. (http://java.sun.com/j2ee).

[160] T. Berners-Lee and L. Masinter and M. McCahill. Uniform Resource Locators (URL). RFC 1738, December 1994.

[161] T. Lockhart. PostgreSQL Administrator's Guide. PostgreSQL Inc., 2000.

[162] D. Tidwell. Web services – the Web's next revolution. IBM DeveloperWorks, November 2000.

[163] Trusted Information Systems, Inc. TIS Internet Firewall Toolkit (FWTK), 1997. (source: ftp.tis.com/pub/firewalls/toolkit).

[164] R. Vanegas, J.A. Zinky, J.P. Loyall, D.A. Karr, R.E. Schantz, and D.E. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Lake District, UK*, September 1998.

[165] V. Vasudevan. A Web Services Primer. XML.com, April 2001.

[166] L. von Schweber. Web Services – Simply The New Middleware. Infomaniacs.com, October 2002.

[167] N. Wang, D.C. Schmidt, and C. O'Ryan. *An Overview of the CORBA Component Model, In: Component-Based Software Engineering (G. Heineman and B. Councill, eds.).* Addison-Wesley, Reading, MA, 2000.

[168] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings OOPSLA '88, vol. 28 of Sigplan Notices, ACM Press*, pages 306–315, 1987.

[169] R. Waters. Microsoft to drop .Net in product branding. Financial Times Online (FT.com), January 2003.

[170] D. Watkins and S. Lange. An Overview of Security in the .NET Framework, January 2002. (http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28000519).

[171] S. Webber. MSDN Talk. *MSDN News*, Vol. 10, No. 4, July/August 2001.

[172] M. Wegdam and A. van Halteren. Experiences with CORBA Interceptors, Position Paper. Reflective Middleware 2000, IBM Palisades Executive Conference Center, New York, USA, April 2000.

[173] I. Welch and R. J. Stroud. Using Reflection as a Mechanism for Enforcing Security Policies in Mobile Code. In *6th European Symposium on Research in Computer Security (ESORICS) Proceedings*, pages 309 – 323, Toulouse, France, October 2000.

[174] C. M. Westphall, J. da Silva Fraga, C. B. Westphall, and S. C. S. Bianchi. Mandatory Security Policies for CORBA Security Model. IFIP TC11 17th International Conference on Information Security (SEC2002): Security in the Information Society – Visions and Perspectives. Cairo, Egypt, May 2002.

[175] R. Wolter. XML Web Services Basics. Microsoft Developer Network, December 2001.

[176] K. Wray. Generic Security Service API : C-bindings (RFC 1509), September 1993.

[177] S.S. Yau and F. Karim. Adaptive Middleware for Ubiquitous Computing Environments. DIPES 2002, Montral, Qubec, Canada, 2002.

[178] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of OOPSLA'92, vol. 28 of Sigplan Notices, ACM Press*, pages 414–434, 1992.

[179] A. Zakinthinos. On the Composition of Security Properties. University of Toronto and Centre for Communications Systems Research/University of Cambridge, 1996.