Chapter 2

MIDDLEWARE

This chapter introduces the reader to the notion of middleware. The essential role of *middleware* is to manage the complexity and heterogeneity of distributed infrastructures. On the one hand, middleware offers programming abstractions that hide some of the complexities of building a distributed application. On the other hand, there is a complex software infrastructure that implements these abstractions. With very few exceptions, this infrastructure tends to have a large footprint. The trend today is toward increasing complexity, as products try to provide more and more sophisticated programming abstractions and incorporate additional layers.

We advance chronologically and discuss briefly the earliest types of middleware targeted at *distributed application development* in Section 1. They are also referred to as *conventional middleware* and comprise the remote procedure call (RPC), transaction processing monitors, object brokers, object monitors and message-oriented middleware.

Conventional middleware is intended to facilitate the development of distributed applications from scratch. With the proliferation of distributed applications in companies, there arose the need for the integration of such applications as opposed the development from scratch. That triggered further the evolution of middleware leading to message brokers and workflow management systems to support *enterprise application integration.* Both types are discussed in Section 2.

The need to integrate applications is not limited to the boundaries of a single company, however. Similar advantages can be obtained from inter-enterprise (or business-to-business, short B2B) application integration as from intra-enterprise application integration. Therefore, the latest breed of middleware was developed to enable *B2B integration.* Application servers and Web services belong in this category. We have a closer look at both in Section

3. In order to limit the scope and hence the size of the problem we focus on application servers and Web services and neglect newer kinds of middleware. Examples for newer kinds are grid and peer-to-peer middleware [Junginger and Lee, 2004], which are also not yet mature enough.

A closer look at application servers and Web services reveals that both types are suffering from increasing complexity. Application servers bundle more and more functionality. Web services are almost universally being built as additional tiers over existing middleware platforms, e.g., application servers, which are already too complex and cumbersome. The complexity of developing and managing distributed applications with application servers is countered by the usage of deployment descriptors. Deployment descriptors are usually XML-files that reduce the amount of coding by specifying orthogonal issues in an declarative and application-independent way. In a similar vain, the Web service community is currently developing a set of standards, denoted WS*, to manage aspects, such as coordination or composition.

Although deployment descriptors and WS* descriptions constitute a very flexible way of developing and administrating a distributed application, we demonstrate by example that there are still many management efforts to be expended by developers and administrators. The reason is that the conceptual model underlying the different descriptions is *only implicit*. Hence, its bits and pieces are difficult to retrieve, survey, check for validity and maintain. This observation serves as input to Chapter 4 where we propose semantic management with the help of *explicit* conceptual models, i.e., ontologies (cf. Chapter 3).

Parts of this chapter provide an overview of middleware based on the significant book of [Alonso et al., 2004]. There are also parts based on [Mahmoud, 2004], as well as [Bernstein, 1996, Campbell et al., 1999]. The example of deployment descriptors is taken from [Oberle et al., 2005c], the one of WS* descriptors from [Oberle et al., 2005a].

1. Middleware for Distributed Application Development

The essential role of *middleware* is to manage the complexity and heterogeneity of distributed infrastructures, thereby providing a simpler programming environment for distributed application developers. It is therefore useful to define middleware as any software layer that is placed above the infrastructure of a distributed system — the network and operating system — and below the application layer [Campbell et al., 1999].

Middleware platforms appear in many guises and it is sometimes difficult to identify their commonalities. Before addressing concrete types of middleware, it is worthwhile to spend some time clarifying the general aspects underlying all middleware platforms.

On the one hand, middleware offers programming abstractions that hide some of the complexities of building a distributed application. Instead of the programmer having to deal with every aspect of a distributed application, it is the middleware that takes care of some of them. Through these programming abstractions, the developer has access to functionality that otherwise would have to be implemented from scratch.

On the other hand, there is a complex software infrastructure that implements the abstractions mentioned above. With very few exceptions, this infrastructure tends to have a large footprint. The trend today is toward increasing complexity, as products try to provide more and more sophisticated programming abstractions and to incorporate additional layers. This makes middleware platforms very complex software systems [Alonso et al., 2004].

This section discusses the middleware used to construct distributed systems from scratch, i.e., middleware for *distributed application development* (also called *conventional middleware*). We further discuss middleware for enterprise application integration and business-to-business (B2B) integration in Sections 2 and 3, respectively. During our discussion we keep an eye on the paradigm shifts regarding the types and granularity of software building blocks because they influenced the evolution of middleware. As depicted in Figure 2.1, software building blocks evolved from *procedures* to *objects*, *workflows*, *components* and finally to *services*.

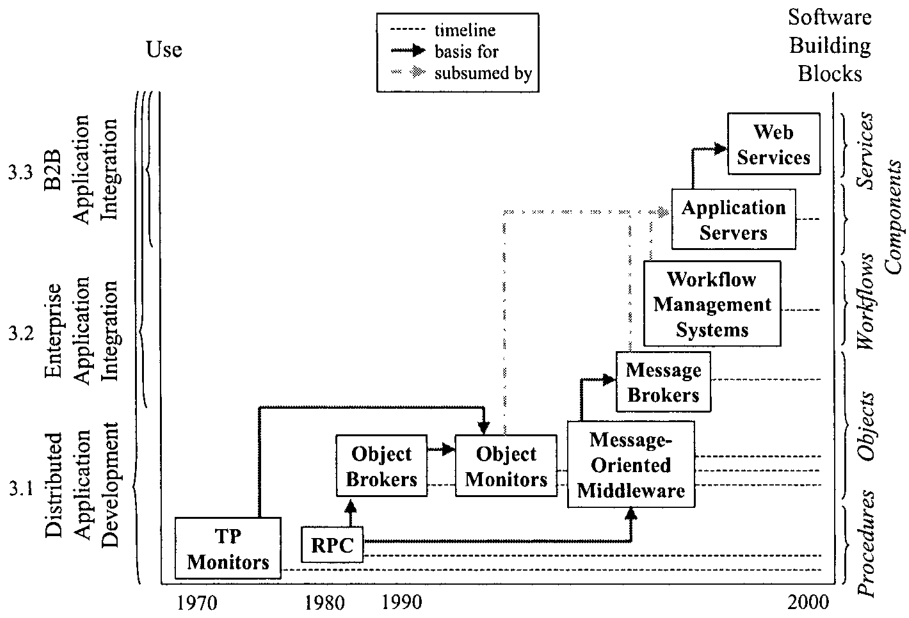

Figure 2.1. Types of middleware and historical overview.

TP Monitors In the early days of corporate IT, computer architectures were mainframe-based and interaction took place through terminals that only displayed the information as prepared by the mainframe.

Transaction processing monitors (TP Monitors), also called transaction processing middleware or simply transaction middleware, were initially designed to allow mainframes to support as many concurrent users as possible. As part of this task, TP monitors also needed to deal with multi-threading and data consistency, thereby extending core functionality with the concept of transactions. They are the oldest and best-known form of middleware. Today, distributed transaction monitors are prevailing to enable transactions spanning several isolated database management systems. [Gray and Reuter, 1993,Tai, 2004]

IBM CICS[1] was the first commercial product offering transaction protected distributed computing on an IBM mainframe. Nowadays, every major software vendor offers its own product, e.g., Microsoft Transaction Server (MTS)[2] or BEA Tuxedo.[3] Sun's Java Transaction API (JTA)[4] specifies standard Java interfaces between transaction monitors and involved parties.

RPC-based systems When the decentralization of corporate IT took place as a consequence of the introduction of the PC, functionality began to be distributed across a few servers. In order to realize distributed applications, developers were in need of a powerful abstraction mechanism to hide the tedious communication details.

The remote procedure call (RPC) responded to this need and was originally presented in [Birrell and Nelson, 1984] as a way to transparently call a procedure located on another machine. RPC established first the notion of a client (the program that calls a remote procedure) and a server (the program that implements the remote procedure being invoked). It also introduced many concepts still widely used today: the interface definition language (IDL), name and directory services, dynamic binding and service interfaces. Today, RPC systems are used as a foundation for almost all other forms of middleware, including Web services middleware (cf. Section 3.2).

Several RPC middleware infrastructures were developed that supported a wealth of functionality, e.g., the Distributed Computing Environment (DCE) provided by the Open Software Foundation (OSF) [Houston, 1996].

[1]Customer Information and Control System, cf. http://www.ibm.com/software/htp/cics/
[2]http://msdn.microsoft.com/library
[3]http://www.beasys.com/products/tuxedo
[4]http://java.sun.com/products/jta/

Object Brokers RPC was designed and developed at a time when the predominant programming languages were procedural languages, i.e., software building blocks were *procedures*. With the advent of object-oriented (OO) languages, the *object* became the software building block, encapsulating data and behavior.

Platforms were developed to support the invocation of remote objects, thereby leading to object brokers. These platforms were more advanced in their specification than most RPC systems, but they did not significantly differ from them in terms of implementation. In practice, most of them used RPC as the underlying mechanism to implement remote object calls. [Alonso et al., 2004]

The most popular class of object brokers are those based on the Common Object Request Broker Architecture (CORBA),[5] defined and standardized by the Object Management Group (OMG).

Object Monitors When object brokers tried to specify and standardize the functionality of middleware platforms, it soon became apparent that much of this functionality was already available from TP Monitors. At the same time, TP monitors, initially developed for procedural languages, had to be extended to cope with object-oriented languages.

The result of these two trends was the convergence between TP monitors and object brokers that resulted in hybrid systems called object monitors. Object monitors are, for the most part, TP monitors extended with object-oriented interfaces. Vendors found it easier to make a TP monitor look like a standard-compliant object broker than to implement object brokers with all the features of a TP monitor and the required performance. [Alonso et al., 2004]

Examples of object monitors are Iona's OrbixOTM.[6] The aforementioned TP monitors, MTS from Microsoft and Tuxedo from BEA, can be classified as object monitors as well.

Message-oriented Middleware (MOM) The previous types of middleware are based on synchronous method invocation, where a client application invokes a method offered by a specific service provider. When the service provider has completed its job, it returns the response to the client. This rather "closely coupled" and "blocking" interoperability soon became too limiting for software developers.

[5] `http://www.omg.org/corba/`

[6] `http://www.iona.com/products/orbix.htm`

The answer to this limit was message-oriented middleware, enabling clients and servers[7] to communicate via messages, i.e., structured data sets typically characterized by a type and name-value-pairs. This kind of communication is made possible by message queues controlled by the MOM. Queues can be shared among multiple applications; recipients can decide when to process messages and do not have to listen continuously; priorities can be assigned, to name but a few advantages of this approach. [Curry, 2004b]

TIB/ETX from Tibco has been a popular product throughout the nineties.[8] Implementations of the Java Message Service (JMS)[9] can be regarded as message oriented middleware. Also, CORBA provides its own messaging service.

2. Middleware for Enterprise Application Integration

The types of middleware discussed so far were originally intended to develop applications from scratch or to integrate database or file servers. The increasing use of such middleware led to the proliferation of distributed applications in companies. Each of the applications provided a higher level of abstraction, and, thus an added value. However, the functionality provided by these applications soon became the subject of further integration. The advantage of application integration is a higher level of abstraction that can be used to hide complex application and integration logic. The disadvantage is that now integration is not limited to database or file servers, but also to applications themselves. Unfortunately, while for databases there has been a significant effort to standardize the interfaces of specific types of databases, the same cannot be said of applications. As long as the integration of applications takes place within a single middleware platform, no significant problem should appear. Once the problem became the integration of applications provided by different middleware platforms, there was almost no infrastructure available that could help reduce the heterogeneity and standardize the interfaces, as well as the interactions between the systems.

The need for such *enterprise application integration (EAI)* further triggered the evolution of middleware, extending its capabilities to cope with application integration, as opposed to the development of new application logic. Such extensions involve significant changes in the way middleware is used. This section briefly discusses message brokers as the most versatile platform for integration and workflow management systems as the tools to make the integration logic explicit. Note that both types of middleware can also be used to develop distributed applications anew instead of integrating existing ones.

[7]Note that the distinction between clients and service providers becomes purely conceptual in the case of MOM. From the perspective of the middleware, all objects look alike.

[8]`http://www.tibco.com`

[9]`http://java.sun.com/products/jms/`

Message Brokers Message-oriented middleware (MOM) is rather static with regard to the selection of the queues to which the messages are delivered. For a generic EAI setting however, we need flexible and dynamic means for communication between arbitrary heterogeneous applications.

In response to those needs, message brokers extend MOM with the capability of routing, filtering and even processing the messages. In addition, most message brokers provide adapters that mask the heterogeneity and make it possible to access all kinds of applications with the same programming model and data exchange format. The combination of these two factors is seen as the key to supporting EAI. [Alonso et al., 2004]

Some of the best-known message brokers include IBM WebSphere MQ,[10] MSMQ by Microsoft[11] or BEA WebLogic Integration.[12]

Workflow Management Systems (WfMS) While message brokers are successful in providing flexible communication among heterogenous applications, the integration logic is still hard-coded and, thus, difficult to maintain.

Workflow management systems tackle the other side of the application integration problem: that of facilitating the definition and maintenance of the integration logic. Business processes are formally defined as a *workflow* and executed by a workflow engine. Workflows are seen as software building blocks for "programming in the large" because they compose coarse-grained activities and applications that can last hours. In addition, workflows compose large software modules, which are typically entire applications. [van der Aalst and van Hee, 2002, Georgakopoulos et al., 1995]

Examples of leading commercial workflow systems include WebSphere MQ Workflow[13] by IBM and Microsoft BizTalk Orchestration.[14]

3. Middleware for B2B Application Integration

So far we have studied middleware for creating and integrating distributed applications within the boundaries of a company. The need to integrate, however, is not limited to the systems within a single company. Similar advantages can be obtained from inter-enterprise (or business-to-business, short B2B) application integration as from intra-enterprise application integration.

With the Web being pervasively available, it goes without saying that some of the same technologies that enabled information sharing on the Web also form the basis for this kind of *B2B application integration*. In particular, HTTP is the

[10]`http://www.ibm.com/software/integration/wmq/`
[11]`http://www.microsoft.com/msmq`
[12]`http://www.bea.com/products/weblogic/integration`
[13]`http://www.ibm.com/webspheremq/workflow`
[14]`http://msdn.microsoft.com/library/`

basic protocol for applications to interact, and XML documents are the standard way to exchange information.

The need for B2B application integration triggered the evolution of middleware. Application servers and Web services provided the solution to the new requirements. Because this work focuses on application servers and Web services, we discuss them in more detail in the following sections. Note that both types of middleware can, of course, be used to develop distributed applications anew and to integrate applications within the boundaries of an enterprise. Most of the work on workflow management of the early nineties migrated to Web-based infrastructure in the late nineties to provide technical capabilities required for B2B applications.

3.1 Application Servers

The increasing use of the Web as a channel to access information systems forced conventional middleware platforms to provide support for Web access. This support is typically associated with application servers. Also, they foster component-based software engineering and introduce the use of deployment descriptors, all of which are discussed below.

The core functionality of an application server can be described by examining the major competing alternatives: application servers based on Sun's J2EE[15] and Microsoft's .NET.[16] Both are similar in terms of their functionality. However, we focus on J2EE in this section without loss of generality. Basically, J2EE is defined by a set of API specifications that is implemented by vendors. Examples are IBM WebSphere[17] or the open-source application server JBoss.[18]

Components and Frameworks

With the increasing complexity in system requirements and the tight development budget constraints, the process of programming applications from scratch is becoming less feasible. As we have seen throughout this chapter, the granularity of software building blocks ever increased and also influenced the evolution of middleware. Constructing applications from a collection of reusable components and frameworks is emerging as a popular approach to software development. This way of constructing applications can be seen as a new paradigm proposing that software should be built by gluing prefabricated components together as in the field of electronics or mechanics.

A (software) *component* is a functional discrete block of logic. Components can be full applications or encapsulated functionality that can be used as part of

[15]Java 2 Enterprise Edition, cf. `http://java.sun.com/j2ee/`
[16]`http://www.microsoft.com/net/`
[17]`http://www.ibm.com/software/websphere/`
[18]`http://www.jboss.org`

a larger application, enabling the construction of applications using components as software building blocks. Components have a number of benefits as they simplify application development and maintenance, allowing systems to be more adaptive and to respond rapidly to changing requirements. Reusable components are designed to encompass a reusable block of software, logic or functionality.

If components are analogous to building blocks, frameworks can be seen as the cement that holds them together. Frameworks are a collection of interfaces and interaction protocols that define how components interact with each other and the framework itself. In essence, frameworks allow components to be plugged into them. Examples of component frameworks include Enterprise JavaBeans (EJB)[19] in the case of J2EE and the Component Object Model (COM)[20] from Microsoft. Frameworks are most often integrated in application servers. [Curry, 2004a]

Application Servers as "Web-enabled" Middleware and Frameworks

Application servers incorporate the Web as a key access channel to the functionality implemented using conventional middleware, leading to "Web-enabled" middleware. Incorporating the Web as an access channel has several important implications. The most significant one is that the presentation logic of the application acquires a much more relevant role than in conventional middleware. This is a direct consequence of how HTTP and the Web work, where all forms of information exchange take place through documents. Preparing, dynamically generating, and managing these documents constitute main requirements to be met by an application server. An application server intends to support multiple types of clients including mobile phones, applications, such as those encountered in conventional middleware, Web services clients, i.e., applications that interact with the server through standard Web services protocols (cf. Section 3.2) and Web browsers. Web browsers are by far the most common type of clients. They interact with the application server via its Web server and receive statically or dynamically generated HTML pages.

Figure 2.2 depicts the API's of the presentation logic layer in the case of J2EE. Dynamic pages are generated by servlets,[21] viz., Java code that handles HTTP requests and generally responds with HTML to be rendered by a requesting browser. A closely related technology is the JavaServer Pages (JSP).[22] JSP is based on servlets, but is more convenient by including Java-code in an HTML page. Support for parsing and transforming XML documents independent of

[19]`http://java.sun.com/products/ejb/`
[20]`http://www.microsoft.com/com/`
[21]`http://java.sun.com/products/servlet`
[22]`http://java.sun.com/products/jsp`

a specific XML processing implementation is provided by Java API for XML Processing (JAXP).[23] JavaMail[24] provides platform-independent and protocol-independent means to build mail and messaging applications. Furthermore, the Java Authentication and Authorization Service (JAAS)[25] enables developers to authenticate users and enforce access controls upon those users in their applications. By abstracting from the complex underlying authentication and authorization mechanisms, JAAS minimizes the risk of creating security vulnerabilities in application code.

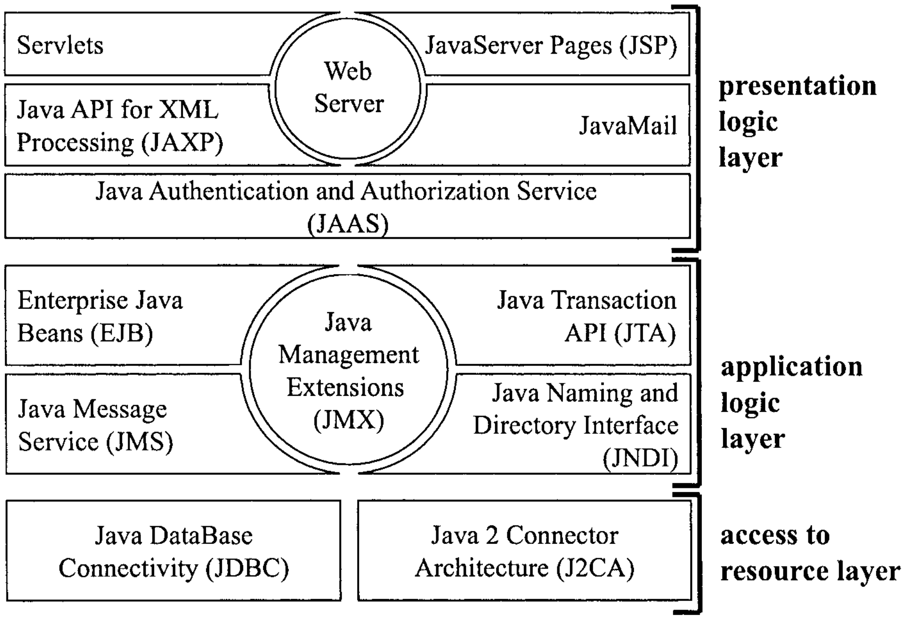
Servlets
Web Server
JavaServer Pages (JSP)
Java API for XML Processing (JAXP)
JavaMail
Java Authentication and Authorization Service (JAAS)
presentation logic layer
Enterprise Java Beans (EJB)
Java Management Extensions (JMX)
Java Transaction API (JTA)
Java Message Service (JMS)
Java Naming and Directory Interface (JNDI)
application logic layer
Java DataBase Connectivity (JDBC)
Java 2 Connector Architecture (J2CA)
access to resource layer

Figure 2.2. J2EE API's divided into layers. [Alonso et al., 2004]

At the application layer, application servers conceptually resemble conventional middleware. The functionality provided is similar to that of TP monitors, CORBA and message brokers. However, component-based software engineering is typically fostered by application servers, which therefore provide a corresponding framework.

The middle section of Figure 2.2 depicts the API's of the application logic layer in the case of J2EE. We can find conventional middleware, such as JTA and JMS, together with directory services accessible via JNDI (cf. Section

[23]`http://java.sun.com/xml/jaxp/`
[24]`http://java.sun.com/products/javamail/`
[25]`http://java.sun.com/products/jaas/`

1). The framework for software components in the form of the Enterprise JavaBeans (EJB) container is a basic part of J2EE-based application servers. Specific EJB components are deployed in this container and contain the bulk of application logic. Some application servers use the recent Java Management Extensions (JMX)[26] technology to put EJB container, directory services and the like in coarser grained components, called managed beans (short MBeans). In contrast to EJB, JMX provides its own framework for such managed beans. The difference is that MBeans can be deployed, undeployed and monitored at run time. They also support interface evolution by a looser coupling.

Finally, J2EE addresses the problem of connecting to the resource layer. Two standards are leveraged in this case: (*i*) Java Database Connectivity (JDBC)[27] that enables developers to access almost any relational database, and (*ii*) the J2EE Connector Architecture (J2CA)[28] that is a generalization of JDBC in that it defines how to build arbitrary resource adapters.

As the complexity of J2EE shows, a significant aspect of application servers is the bundling of more and more functionality within the middleware platform. This is consistent with the trend toward providing integrated support for many different middleware abstractions that we have witnessed in conventional middleware. In fact, as software vendors continue to extend their middleware offerings and package them in many different ways, it becomes hard even to distinguish what is inside an application server and what is not. In many cases, the name originally given to the application server (e.g., IBM WebSphere) has been progressively used to label every middleware product offered by a company. For example, IBM messaging and workflow platforms are now marketed under the name WebSphere MQ.

Deployment Descriptors

Application servers try to tame the increasing complexity of their bundled functionality by managing orthogonal issues in an application independent way. They introduce vertical services, e.g., load balancing, pooling, caching, transactions, session management, user rights and persistence, that span all layers. Thus, the responsibility is shifted from the development to the deployment process, i.e., "the process whereby software is installed into an operational environment" according to the J2EE glossary.

XML files are used to describe how components and applications should be deployed and how vertical services should be configured. Such deployment descriptors[29] direct deployment tools to deploy a component, an application

[26]`http://java.sun.com/products/JavaManagement/`
[27]`http://java.sun.com/products/jdbc`
[28]`http://java.sun.com/j2ee/connector/`
[29]J2EE deployment descriptor, `http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf`

or a vertical service with specific options and describe specific configuration requirements that a deployer must resolve.

While it is always a good idea to reduce the amount of source code that has to be written, the deployment process can be quite tricky in itself. Deployment tools merely act as an input mask, which generates the specific XML syntax for the user. This is definitely a nice feature; however, the developer must fully understand the quite complicated concepts that lie behind the options for the transactional behavior, for instance, and juggle all of them at the same time. The current deployment tools do not help to avoid or even actively repair configurations that may cause harmful system behavior. Even worse, this problem is duplicated, as there is a plethora of deployment descriptors for different kinds of components (servlets, EJBs, MBeans) and vertical services (security, transactions, etc.).

We here present a case of how tricky the deployment process can become. It is the interesting case of indirect permissions due to context switches (cf. Figure 2.3). As an example, consider the `anonymous` user who accesses a Web shop by the HTTP basic authentication. The script on this page, say a servlet, might connect to the `CustomerEntityBean`, an EJB, which in turn accesses the `Customer` table in the database. We assume that the database is only accessible by `dbuser`. Therefore, the EJB performs an explicit context switch (which is frequently described as the run-as paradigm). The call succeeds, because the user information will be propagated and the call will also be executed using the `dbuser`'s credentials. This case is definitely not a bug; however, it remains a pure manual and tedious task for the administrator of the application server to keep track of such indirect permissions. [Oberle et al., 2005c]

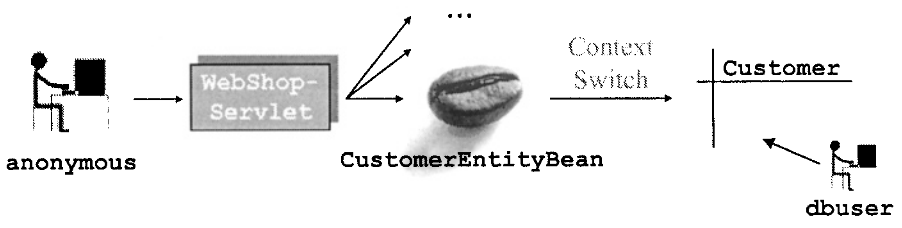

Figure 2.3. Example of indirect permission. [Oberle et al., 2005c]

In this example, the administrator needs to analyze two different deployment descriptors, as well as the source code to discover the situation outlined above. First, the deployment descriptor of the servlet container (`web.xml`) states that only authenticated users may access the `WebShopServlet`:

Example 2.1 (`web.xml`)

```
...
<security-constraint>
 <web-resource-collection>
  <web-resource-name>WebShopServlet</web-resource-name>
  <url-pattern>/servlet/WebShopServlet</url-pattern>
  <http-method>GET</http-method>
 </web-resource-collection>
</security-constraint>

<login-config>
 <auth-method>BASIC</auth-method>
</login-config>
...
```

Second, the `WebShopServlet` itself accesses the `CustomerEntityBean`. The servlet's `doGet()` method serves the incoming HTTP requests. In our case it queries user account information out of the `Customer` table by means of the bean in order to display it to the user. After retrieving a handle to the bean via the Home interface, the `getCustomerName()` method of the bean is invoked by the servlet.

Example 2.2 (`WebShopServlet.java`)

```
public class WebShopServlet extends HttpServlet {
 public void doGet(HttpServletRequest request,
  HttpServletResponse response)
 {
  ...
  //get customer info via CustomerEntityBean
  CustomerObject cObject = cHome.create()
  out.println(cObject.getCustomerName())
  ...
 }
...
}
```

Third, the deployment descriptor of the `CustomerEntityBean`, called `ejb--jar.xml`, states that the bean performs a context switch via the `<run-as--specified-identity>` tag. It thus accesses the database table with `dbuser`'s credentials:

Example 2.3 (ejb-jar.xml)

```
...
<ejb-jar>
<enterprise-beans>
 <entity>
  <ejb-name>CustomerEntityBean</ejb-name>
  <ejb-class>edu.unika.aifb.CustomerEntityBean</ejb-class>
  ...
  <security-identity>
   <run-as-specified-identity>
    <role-name>dbuser</role-name>
   </run-as-specified-identity>
  </security-identity>
 </entity>
</enterprise-beans>
</ejb-jar>
...
```

Assessing such situations for any user, any EJB and any database table becomes an impossible task for developers and administrators. Rather, it is desirable to query a system from different perspectives, e.g., "*Are there any users with indirect permission to resources? And if yes, what are those resources?*" or "*Are there any indirect permissions on the* `Customer` *table? And if yes, who are the users?*" Such a system requires the explication of the conceptual model underlying the different descriptions. Each deployment descriptor introduces its own conceptual model implicitly in the corresponding XML-DTD. Therefore, it is difficult to arrive at conclusions that are a result of an integration of such descriptors. Consequently, Chapter 4 proposes the usage of ontologies to support developers and administrators in these tasks.

As we introduce in Chapter 3, ontologies are a means to formally specify a coherent conceptual model with logic-based semantics. The modelling of the computational domain has to be done rigorously, because we encounter fundamental ontological questions: *What is the difference between the users in the operating system, in the database system and within the application server's realm (where users are called principals)? Are there any conceptual differences except their placement in a different realm?* Also, we might be interested in the relationship between a user in an information system and the corresponding natural person. To infer the total of access rights granted for a natural person who might have several user accounts in and across information systems, might reveal further security holes.

3.2 Web Services

The types of middleware discussed so far are all based on tightly-coupled software building blocks (procedures, objects, workflows and components). That means interfaces between the different software building blocks of an application are closely interrelated in function and form, thus making them brittle when any form of change is required to parts or the whole application.

The need for B2B applications to adapt to changing environments is a key reason that made loosely-coupled systems attractive. In this section we explain how Web services came about and how they may meet the new requirements. First, one has to understand the paradigm of service-oriented architectures, which factorizes the functionality in loosely-coupled services. A second aspect is the way that Web services redesign the conventional middleware protocols. Finally, standardization plays a major role, which led to a set of specifications of different Web services aspects, labelled WS*.

Service-Oriented Architectures (SOA)

Today, businesses have to adopt quickly to changing environments, such as changing policies, business strengths, business focus, partnerships or industry standing. Businesses that are able to act flexibly in relation to their environment where change occurs as required, are called "on demand" businesses. They triggered the need for loosely-coupled systems in order to become more agile with respect to changing environments.

The SOA paradigm is the answer to this and other needs. The functionality of a distributed system is split into services instead of tightly-coupled objects or components. *Services* are loosely-coupled, autonomous and independent software building blocks. In order to work on a global scale, standards have to be defined for service invocation, description, discovery, coordination and composition.

SOA-based systems do not exclude the possibility that individual services can themselves be built with object-oriented design. It allows objects within the system and is as such object-based, but not as a whole object-oriented. The difference is that many aspects that were hard-coded before have to be specified dynamically and declaratively. One needs to specify how the overall application performs its workflow between services. The workflow may include services not just between departments, but even with other external partners. Policies have to be defined as to how relationships between services should transpire. All this has to work in an environment of trust and reliability, which is given implicitly when business partners know each other and agree on terms beforehand. [IBM developerWorks, 2004a]

Web Services as Middleware for SOA-based Systems

The Web-based middleware for SOA-based systems is called Web services. Web services subsume a set of protocols and XML-languages for interface description, invocation, discovery and composition of services. The minimalist Web services middleware is comprised of SOAP (Simple Object-based Access Protocol [Gudgin et al., 2003]), the standard for the invocation, and WSDL (Web Service Description Language [Christensen et al., 2001]), the standard for the interface description. Further standards for discovery, coordination and composition are being developed at the time of writing, as discussed below.[30]

The *evolutionary* nature of Web services presents them as extensions to conventional middleware that provides a set of simple interfaces for interactions across the Internet. These extensions make Web-based integration possible at least in simple scenarios (such as EAI or closed communities of business partners). SOAP and WSDL constitute yet another tier on the internal middleware of an organizational unit (cf. Figure 2.4).

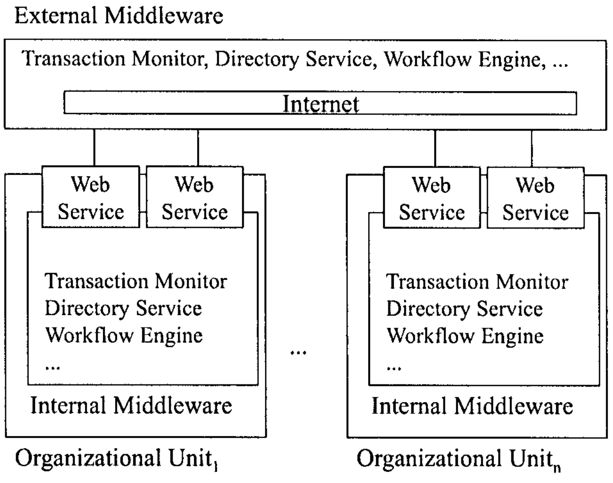

Figure 2.4. Internal vs. external middleware. [Alonso et al., 2004]

Two organizational units are able to perform application integration if they both agree on using SOAP and WSDL, even if they use different internal middleware. For example, Web services might draw from components residing in application servers (internal middleware) distributed over different organizational units and heterogeneous platforms. Application servers are an obvious

[30]Message-oriented middleware is sometimes considered as middleware for SOA-based systems, too. In fact, it defines similar concepts, but lacks the standardization necessary to realize SOA-based systems on a global scale. We discuss these matters in the next section.

target to support such a "wrapping" by SOAP and WSDL, as they provide the basic infrastructure (Web server, XML parsers, etc.). In most cases, the developer is only required to mark a certain method with meta-tags in the source code. The application server cares for automatically generating the WSDL description and handling the SOAP messages.

The new tiers Web services add to the already overly complex internal middleware lead to significant performance overhead and increase the complexity of developing, tuning, maintaining, and evolving multi-tier systems. Translation to and from XML, tunnelling of invocations through SOAP, clients embedded in Web servers and many of the technologies typical of Web services do not come for free. Furthermore, Web services will introduce additional, external middleware, thus adding extra complexity.

The *revolutionary* view sees Web services as radically changing the way integration is achieved. The assumption seems to be that once SOAP and WSDL are used, then Web services will facilitate the development of infrastructures that support programmatic application integration, dynamic B2B marketplaces and the seamless integration of IT infrastructures from different cooperations.[31] However, the autonomous nature of such SOA-based systems demands the redesign of the middleware protocols to work in a loosely-coupled fashion and across organizational units.

Internal middleware protocols were designed based on assumptions that do not hold in cross-organizational interactions. For example, they assumed a central transaction coordinator and the possibility for this coordinator to lock resources indefinitely. Lack of trust and confidentiality issues often make a case against a central coordinator and, therefore, middleware protocols must now be redesigned to work in a fully distributed fashion and must be extended to allow more flexibility in terms of locking resources. Similar arguments can be made for all interaction and coordination protocols and, in general, for many of the other properties provided by conventional and internal middleware, such as reliability and guaranteed delivery. What was then achieved by a centralized platform must now be redesigned in terms of protocols that can work in a decentralized setting and across trust domains. One example of such "external" middleware is UDDI (Universal Description Discovery & Integration [UDDI Coalition, 2000]), allowing the discovery of Web services.

In order to facilitate application integration with Web services on a global scale, the external Web services middleware must rely on standards. These standards shape the current Web services landscape to a large extent. We have introduced SOAP, WSDL, as well as UDDI so far. We introduce additional ones in the next subsection.

[31]Today, Web services are not as revolutionary as one may think. They are mostly used in the evolutionary way for conventional EAI.

WS*

Having an SOA and redefining the middleware protocols is not sufficient to address loosely-coupled and dynamic application integration on a global scale, unless the language and protocols become standardized and widely adopted. Consortia, such as the Organization for the Advancement of Structured Information Standards (OASIS)[32] or the World Wide Web Consortium (W3C),[33] attempt to standardize all the different aspects beyond invocation (SOAP), description (WSDL) and discovery (UDDI). The commitment for standardization does not necessarily mean that there will be one specification for each aspect, however. Below, we give an incomplete overview of the aspects that are currently being specified. Altogether, they form an inscrutable set and are labelled *WS**. [Alonso et al., 2004]

WS-Coordination The primary goal of this specification is to create a framework for supporting coordination protocols. In this regard, it is intended as a meta-specification that will govern specifications that implement concrete forms of coordination protocols. [Cabrera et al., 2003]

WS-Transaction WS-Transaction is an example of a concrete coordination protocol specified by means of WS-Coordination. WS-Transaction is split into the WS-AtomicTransaction protocol for short duration transactions and WS-BusinessActivity to enable existing workflow systems to wrap their proprietary mechanisms and interoperate across trust boundaries. [Cabrera et al., 2004]

WS-BPEL The Business Process Execution Language for Web Services (WS-BPEL) is the de facto standard for specifying service composition. It also allows specifying coordination between Web services, thus acting as an alternative to WS-Coordination. [Andrews et al., 2005]

WS-Security WS-Security is an extension to SOAP for end-to-end application-level security that is otherwise ignored by underlying protocols, such as HTTPS. It adds to SOAP the mechanisms of signatures and encryption. [Atkinson et al., 2002]

WS-Policy is a proposal for a framework through which Web services can express their requirements, capabilities and preferences (commonly referred to as "policies") to each other in an interoperable manner. It defines a set of generic constructs for defining and grouping policy assertions. [Bajaj et al., 2004, Alonso et al., 2004]

[32] `http://www.oasis-open.org`
[33] `http://www.w3.org`

WS-Trust The Web Services Trust Language (WS-Trust) uses the secure messaging mechanisms of WS-Security to define additional primitives and extensions for security token exchange to enable the issuance and dissemination of credentials within different trust domains. [BEA Systems et al., 2004]

Other aspects and specifications include WS-Addressing, WS-Attachments, WS-Eventing, WS-Federation, WS-Inspection, WS-Manageability, WS-MetaDataExchange, WS-Notification, WS-Routing, and many more. An overview is given in [IBM developerWorks, 2004b].

The advantages of WS* are multiple and have already benefited some industrial cases. Similar to deployment descriptors in application servers, WS* descriptions manage orthogonal aspects in an application independent way. XML-files declaratively describe how Web services should be deployed and configured. Thus, WS* descriptions are exchangeable and developers may use different implementations for the same Web service description. The disadvantages of WS*, however, are also visible; even though the different standards are complementary, they must overlap and one may produce models composed of different WS* descriptions, which are inconsistent, but do not easily reveal their inconsistencies. The reason is that there is no coherent formal model of WS* and, thus, it is impossible to ask for conclusions that come from integrating several WS* descriptions. Thus, discovering such Web Service management problems or asking for other kinds of conclusions that derive from the integration of WS* descriptions remains a purely manual task of the software developers accompanied by little or no formal machinery.

As an example for a trivial conclusion derived from both a WS-BPEL and WS-Policy description, consider the following case. Let's return to Example 2.1 on page 23 of a web shop and assume we have realized it with internal and external Web services composed and managed by a WS-BPEL engine. After the submission of an order, we have to check the customer's credit card for validity, depending on the credit card type (VISA, MasterCard, etc.). We assume that credit card providers offer this functionality via Web services. The corresponding WS-BPEL process `checkAccount` thus invokes one of the provider's Web services, depending on the customer's credit card. Example 2.4 shows a snippet of the WS-BPEL process definition.

Example 2.4 (WS-BPEL)

```
...
<process name="checkAccount">
 <switch ...>
  <case condition="getVariableData('card')='VISA'">
   <invoke partnerLink="toVISA"
    portType="visa:CCPortType"
    operation="checkCard"...>
   </invoke>
  </case>
  <case condition="getVariableData('card')='MasterCard'">
   <invoke partnerLink="toMastercard"
    portType="mastercard:CCPortType"
    operation="validateCardData"...>
   </invoke>
  </case>
 ...
 </switch>
</process>
...
```

Suppose now that the Web service of one credit card provider, say MasterCard, only accepts authenticated invocations conforming to Kerberos or X509. It states such policies in a corresponding WS-Policy document, such as the one sketched in Example 2.5. The invocation will fail unless the developer ensures that the policies are met. The developer has to check the policies manually at development time or has to implement this functionality to react to policies at run time, assuming that no policy matching engine is in place.

Example 2.5 (WS-Policy)

```
...
<wsp:Policy>
 <wsp:ExactlyOne>
  <wsse:SecurityToken>
   <wsse:TokenType>wsse:Kerberosv5TGT</wsse:TokenType>
  </wsse:SecurityToken>
  <wsse:SecurityToken>
   <wsse:TokenType>wsse:X509v3</wsse:TokenType>
  </wsse:SecurityToken>
 </wsp:ExactlyOne>
</wsp:Policy>
...
```

As we may recognize from this small example, it is desirable to support the developer with unambiguous specifications and formal machinery to arrive at such conclusions automatically. This is particularly helpful when we think of more sophisticated examples where we have large indirect process cascades or additional WS* descriptors to consider. However, it remains a manual task for the developer to discover and assess such situations. The reason is that there is no coherent conceptual model underlying the WS* descriptions — very similar to the case of deployment descriptors in application servers. As a consequence, Chapter 4 proposes the usage of ontologies in Web services middleware to support developers and administrators in performing such tasks.

Ontologies are a means to formally specify conceptual models with logic-based semantics. The domain of Web services demands a rigorous modelling because we are confronted with fundamental ontological questions. *What is the difference between a policy of a Web Service and an access right on a software component? Are they the same? Can workflows of Web services be modelled such as the invocation chain of software components?* Such questions call for a concise and fundamental introduction of ontologies, which is given in Chapter 3.

4. Summary

In this chapter we have discussed the evolution of middleware providing a brief overview for the reader. We have advanced from the earliest types of middleware targeted at distributed application development. With the proliferation of distributed applications in companies there arose the need for enterprise application integration. That triggered further the evolution of middleware resulting in middleware for enterprise application integration. Finally, we have had a closer look at the current state-of-the-art, viz., middleware for business-to-business (B2B) application integration. Application servers and Web services belong in this category. Both offer a wealth of functionalities for realizing business-to-business application integration via the Web. Application servers bundle more and more functionality and Web services are almost universally being built as additional layers over existing middleware platforms, which are already too complex and cumbersome. The complexity is countered by the usage of deployment descriptors that reduce the amount of coding by specifying orthogonal issues in an application independent way. In a similar vein, the Web service community is currently developing a set of standards, WS*, to manage aspects such as coordination or composition.

Though deployment descriptors and WS* descriptions constitute a very flexible way of developing and administrating a distributed application, we have demonstrated that developers and administrators still need to expend significant efforts. The reason is that the conceptual model underlying the different descriptions is *only implicit.* Hence, its bits and pieces are difficult to retrieve, survey and check for validity and maintain. It remains a manual task to arrive at conclusions that are the result of combining such descriptions. Hence, Chapter 3 introduces the reader to *ontologies* as a means to formally specify conceptual models with logic-based semantics. We have also demonstrated that the domain of software components and Web services demands a careful and rigorous ontological modelling.