5 Web Services

In previous chapters we have discussed the architecture of information systems (Chapter 1), middleware and enterprise application integration (Chapters 2 and 3), and the basics of Web technology (Chapter 4). These chapters have shown a chronological evolution of the technology used for EAI and for building distributed applications. All these technologies have been rather successful in addressing several of the problems created by application integration. The success, however, has been restricted to certain settings (e.g., LAN-based systems, homogeneous middleware platforms, etc.). True application integration requires tools that go one step beyond what conventional middleware and EAI platforms have achieved. Web services and the associated technology are being leveraged to take such a step.

To establish the context for the rest of the book, we provide in this chapter an introduction to Web services. We look at Web services as a way to expose the functionality of an information system and make it available through standard Web technologies. The use of standard technologies reduces heterogeneity, and is therefore key to facilitating application integration. Furthermore, we show that Web services naturally enable new computing paradigms and architectures, and are specifically geared toward service-oriented computing, a paradigm often touted in the past but never quite realized.

This chapter is structured along a continuum that goes from the needs that motivate the introduction of Web services to the solutions that Web services provide. We begin by showing the limitations of conventional technology in tackling some of the application integration challenges, thereby raising the need for a novel technology–a need addressed by Web services. We then describe the essential concepts behind Web services and how they tackle the application integration problem (Section 5.1). Next, we provide an overview of Web services middleware, focusing in particular on the functionality that this middleware must provide to support the development of distributed applications based on Web services (Section 5.2). Finally, we discuss Web services architectures (Section 5.3).

5.1 Web Services and their Approach to Distributed Computing

Before describing the problems Web services try to solve and how they address them, we define what Web services are.

5.1.1 Defining Web Services

The term *Web services* is used very often nowadays, although not always with the same meaning. Nevertheless, the underlying concepts and technologies are to a large extent independent of how they may be interpreted.

Existing definitions range from the very generic and all-inclusive to the very specific and restrictive. Often, a Web service is seen as an application accessible to other applications over the Web (see e.g., [72, 133]). This is a very open definition, under which just about anything that has a URL is a Web service. It can, for instance, include a CGI script. It can also refer to a program accessible over the Web with a stable API, published with additional descriptive information on some service directory.

A more precise definition is provided by the UDDI consortium, which characterizes Web services as *"self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces"* [203]. This definition is more detailed, placing the emphasis on the need for being compliant with Internet standards. In addition, it requires the service to be open, which essentially means that it has a published interface that can be invoked across the Internet. In spite of this clarification, the definition is still not precise enough. For instance, it is not clear what it is meant by a modular, self-contained business application.

A step further in refining the definition of Web services is the one provided by the World Wide Web consortium (W3C), and specifically the group involved in the Web Service Activity: *"a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols"* [212].

The W3C definition is quite accurate and also hints at how Web services should work. The definition stresses that Web services should be capable of being "defined, described, and discovered," thereby clarifying the meaning of "accessible" and making more concrete the notion of "Internet-oriented, standards-based interfaces." It also states that Web services should be "services" similar to those in conventional middleware. Not only they should be "up and running," but they should be described and advertised so that it is possible to write clients that bind and interact with them. In other words, Web services are components that can be integrated into more complex distributed applications. This interpretation is very much in line with the perspective we take in this book, and explains why we place so much emphasis on the need to understand middleware as the first step toward understanding Web services.

The W3C also states that XML is part of the solution. Indeed, XML is so popular and widely used today that, just like HTTP and Web servers, it can be considered as being part of Web technology. There is little doubt that XML will be the data format used for many Web-based interactions.

Note that even more specific definitions exist. For example, in the online technical dictionary *Webopedia*, a Web service is defined as *"a standardized way of integrating Web-based applications using the XML, SOAP, WSDL, and UDDI open standards over an Internet protocol backbone. XML is used to tag the data, SOAP is used to transfer the data, WSDL is used for describing the services available, and UDDI is used for listing what services are available"* [110]. Specific standards that could be used for performing binding and for interacting with a Web service are mentioned here. These are the leading standards today in Web services. As a matter of fact, many applications that are "made accessible to other applications" do so through SOAP, WSDL, UDDI, and other Web standards. However, these standards do not constitute the essence of Web services technology: the problems underlying Web services are the same regardless of the standards used. This is why, keeping the above observations in mind, we can adopt the W3C definition and proceed toward detailing what Web services really are and what they imply.

5.1.2 Motivating the Need for B2B Integration

Before describing in more detail what Web services are about, we introduce an example that shows why the middleware and EAI platforms discussed in the previous chapters are not sufficient in certain application integration scenarios. The limitations of these systems are what led to the current efforts around Web services as well as the shift to a service-oriented paradigm in application development.

Consider again the supply chain scenario introduced in Chapter 3. When describing the problem behind supply chain automation, we observed that the main issues were the integration of several autonomous and heterogeneous systems and the automation of business processes spanning across these systems. Until very recently, such processes were executed manually for the most part. The advent of EAI platforms made it possible to automate most of such processes.

In that example, we assumed that all the components belonged to one company. Consider now the same problem in the general case, where the systems are not all running within a company but are instead managed and operated by different companies. Specifically, consider a procurement scenario, where a company (acting as a customer) needs to order goods from another company (a supplier). The supplier then processes the order and delivers the goods, either directly (if it has goods in stock) or by requesting that the goods be shipped by a third party (in this example, a warehouse that serves several suppliers and delivers goods upon requests). Once the order is processed, the customer makes the payment.

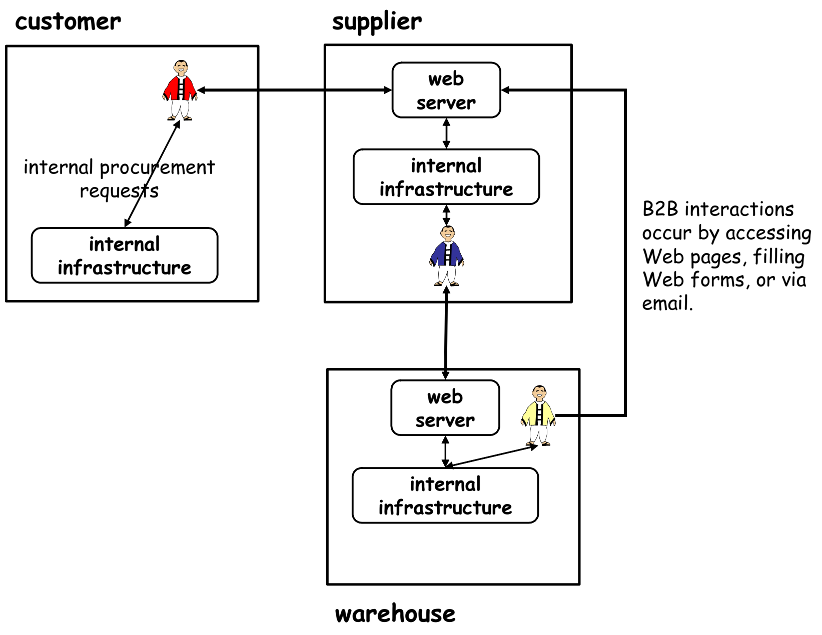

Fig. 5.1. Very often, integration across companies is still done manually

For all the parties involved (customers, suppliers, and the warehouse), it would be very beneficial if the whole procurement process was automated, all the way from requesting quotes to processing payments. Today, in all but very few cases, even if business processes within a company are automated, business processes across companies are carried out manually. This is exemplified in Figure 5.1, which emphasizes the fact that the "integration" is performed manually by means of employees who access the internal systems (for example to retrieve the list of products to be ordered) and then communicate with other companies by filling out Web forms (e.g., to order the goods) or via email or fax. The attentive reader will observe that Figure 5.1 is very similar to Figure 3.1 of Section 3.1, the only difference being that in this case we are focusing on a B2B scenario rather than an EAI one. The problems are similar and the need for automation is driven by the same goals, such as lower costs, streamlined and more efficient processes, ability to monitor and track process executions, and ability to detect and manage exceptions. Yet, while the problems are the same, the solution needed in this case is different. In fact, none of the technologies that we have described so far either for EAI or for wide area integration have been able to fully address the challenges posed by the above scenario and become widely adopted.

5.1.3 Limitations of Conventional Middleware in B2B Integration

There are several reasons why conventional middleware platforms cannot be used in this setting. The first one is that in cross-organizational interactions there is no obvious place where to put the middleware. The basic idea for conventional middleware was for it to reside between the applications to be integrated and to mediate their interactions. While the applications were distributed, the middleware was centralized (at least logically), and it was controlled by a single company. Adopting the same solution in this context would require that the customer, supplier, and warehouse agree on using and cooperatively managing a certain middleware platform (e.g., a specific message broker, a specific workflow system, and a specific name and directory server) and on implementing a "global workflow" that drives the whole business process. This approach is presented in Figure 5.2, and is analogous to Figure 3.10 of Section 3.3, which described how EAI middleware can integrate applications.

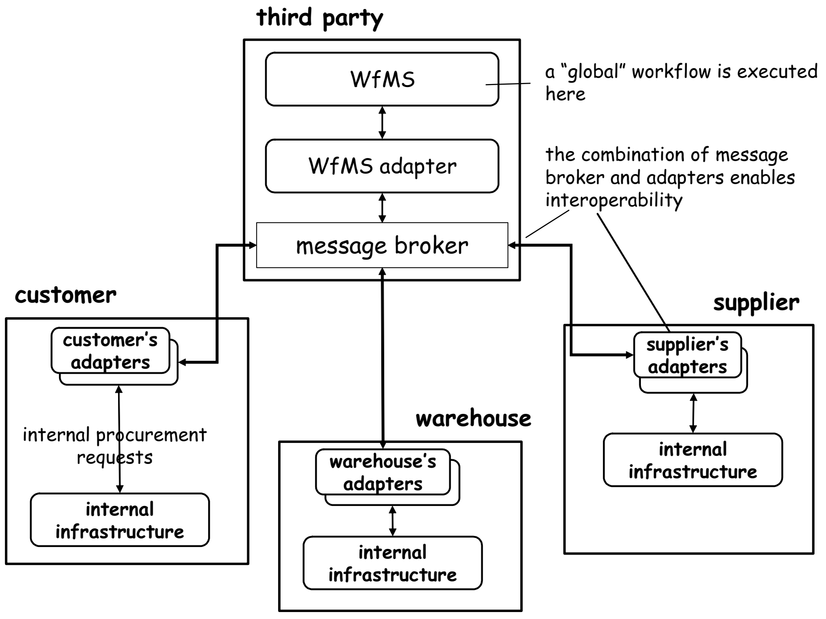

Fig. 5.2. B2B integration performed in the same way EAI is done. While this is conceptually possible, it rarely happens in practice due to lack of trust, autonomy, and confidentiality reasons

While this approach is feasible in some restricted settings (e.g., a very small number of companies that have frequent, close cooperation), in the general case it turns out to be an unlikely proposition. In fact, the lack of trust between companies, the autonomy that each company wants to preserve, and the confidentiality of the business transactions play against the idea of having a centralized middleware hosted by one of the participating companies or by a third party. Each company wants to control its own business operations and how they are carried out, and does not want its business transaction data to be seen by anybody other than its intended recipient.

An alternative solution for a company would be to address the problem in a point-to-point fashion, by separately tackling the integration problem with each of the partners. This means that whenever two parties (the customer and the supplier) want to communicate, they agree on using certain middleware protocols and infrastructure. For example, they can both deploy a message broker and use it to send messages to each other (Figure 5.3), as long as this message broker provides the necessary support for wide area integration (e.g., firewall traversal, discussed in the previous chapter). We have already seen in Section 3.2.5 how two or more applications sitting on top of two distinct but homogeneous message brokers can communicate. With this approach, there is no third party involved and confidentiality is preserved, as only the intended recipient can see the business transactions.

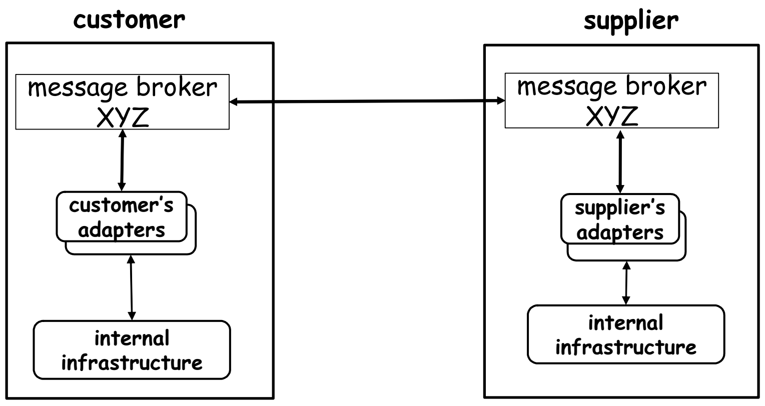

Fig. 5.3. Point-to-point integration across companies

However, since a company typically interacts with many different partners and each partner could require the use of a different middleware platform, this leads to a scenario where a company has to support many heterogeneous middleware systems. The result is that each company must integrate these different middleware systems (not to mention purchasing and maintaining them), which were instead intended to facilitate the integration (Figure 5.4).

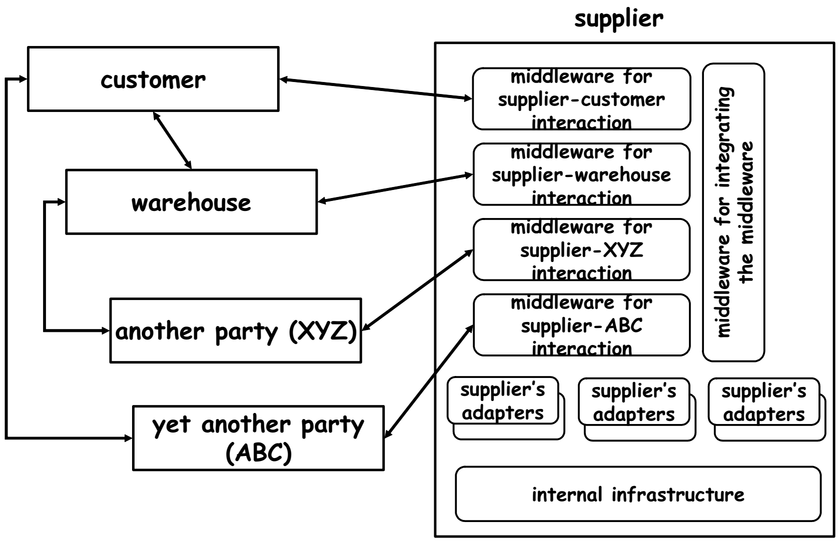
supplier
customer
warehouse
another party (XYZ)
yet another party (ABC)
middleware for supplier-customer interaction
middleware for supplier-warehouse interaction
middleware for supplier-XYZ interaction
middleware for supplier-ABC interaction
middleware for integrating the middleware
supplier's adapters
supplier's adapters
supplier's adapters
internal infrastructure

Fig. 5.4. The lack of a central middleware platform means that interactions are managed in a point-to-point manner, possibly using different middleware platforms to communicate with different parties

Another reason that makes conventional middleware unsuitable is that many assumptions that were valid in EAI do not hold here. One such difference is that EAI interactions are typically short lived, while cross-organizational interactions last longer, and sometimes much longer. Rather than calling a procedure, a method, or a function, interactions involve coarse-grained operations lasting possibly for hours or days. As an example of the delays involved, the supplier may confirm that the order has been processed only after the requested goods have been physically picked up by a shipping company. Such delays explain why cross-organizational interactions are mostly implemented as asynchronous exchanges. However, asynchronous interactions introduce their own problems. For example, consider the problem of providing transactional properties to the interaction between two or more parties. If the operations are long-lasting, then conventional protocols such as 2PC are not applicable, as they would lock resources for long period of time and therefore severely limit the possibility of executing concurrent operations. Yet, this are the protocols supported by conventional middleware and EAI tools.

Furthermore, while EAI interactions occur in the same trust domain, cross-organizational interactions occur across trust domains, and there is an implicit lack of trust between interacting entities. Not only does this require authentication and encryption of messages, but it also implies that companies will severely restrict what clients can do on their system. Referring again to the transactional example, service providers will want to control and limit the resources that can be locked, and are certainly not going to give up the control of the locking mechanisms to a possibly malicious outside entity.

5.1.4 B2B Integration before Web Services

The limitations just described arise from the fundamental assumption behind conventional middleware that the middleware platform can be centralized and is trusted by the components to be integrated. This is not the case in B2B exchanges. In addition, the lack of standardization across middleware platforms makes point-to-point solutions (i.e., solutions tailored to concrete middleware platforms) costly to realize in practice.

Note that this does not mean that B2B integration has never been technically possible or was never achieved. Indeed, there are several successful examples of cross-enterprise integration. Some of them are represented by broker companies such as Ariba or CommerceOne. The purpose of these brokers is to facilitate integration by performing functions analogous to those of centralized EAI middleware, from supporting binding to routing messages among the services provided by the different companies. However, the lack of support by major software vendors for the formats and protocols defined by these brokers and the trust-related problems that undermine any centralized solution have resulted in limited acceptance for these solutions.

Other successful examples of B2B integration are systems based on EDIFACT (discussed in the previous chapter). For instance, the US retailer Wal-Mart was able to automate some of its cross-enterprise processes, in particular with respect to co-managing the inventory with its suppliers, setting stock levels, and automatically ordering supplies when in-stock levels were low.

In spite of the success stories, standards like EDIFACT and systems based on such standards have never become widely adopted for a variety of reasons. First, designing such systems is typically an ad hoc endeavor and the result of a one time programming effort. The lack of standards and the lack of an appropriate infrastructure (from middleware to networks) made each one of these systems unique in that each one of them had to implement everything almost from scratch. In addition, the underlying hardware and communication support was very heavy-handed. In terms of networks, before the Web appeared communication often used to take place through leased lines to obtain the necessary bandwidth and security guarantees. In terms of computer cycles, most of these systems were very heavy. As a result, such systems were expensive to develop, almost impossible to reproduce, difficult to maintain, and could not be adapted to new technologies. Moreover, because of the development effort and costs involved, only large companies could afford deploying such systems.

The Internet alleviated some of these design problems by allowing designers to replace leased lines with a network that was pervasive and more cost-efficient. Nevertheless, the lack of standardization at the system and communication protocol levels still remained a significant hurdle in the path toward reducing the cost and complexity of building and deploying such systems. This problem was recognized years ago and there were many standardization attempts but, for reasons not always entirely rational, they had only limited success. At the core of these efforts were technologies that would allow homogeneous middleware platforms to communicate with each other (such as Inter-ORB communication via GIOP/IIOP). These technologies can be easily extended to act as the middleware for the Web. However, as it often happens, these early approaches were never widely used and, in time, were obscured by new developments.

The Web constituted an important step toward facilitating application integration. In fact, it probably was the crucial step toward systems that were more than isolated, ad hoc efforts. The Web brought standard interaction protocols (HTTP) and data formats (XML) that were quickly adopted by many companies, thereby creating a base for establishing a common middleware infrastructure that reduces the heterogeneity among interfaces and systems. However, HTTP and XML by themselves are not enough to support application integration. They do not define interface definition languages, name and directory services, transaction protocols, and the many other abstractions that, as the previous chapters have shown, are crucial to facilitate integration. It is the gap between what the Web provides (HTTP, XML) and what application integration requires that Web services are trying to fill.

5.1.5 B2B Integration with Web Services

The contribution of Web services toward resolving the limitations of conventional middleware involves three main aspects: service-oriented architectures, redesign of middleware protocols, and standardization.

Service-oriented Paradigm

Web services work on the assumption that the functionality made available by a company will be exposed as a service. In middleware terms, a service is a procedure, method, or object with a stable, published interface that can be invoked by clients. The invocation, *and this is very important*, is made by a program. Thus, requesting and executing a service involves a program calling another program.

In terms of how they are used, Web services are no different from middleware services, with the exception that it should be possible to invoke them across the Web and across companies. As a consequence, Web services assume that services are loosely-coupled, since in general they are defined, developed, and managed by different companies. As Web services become more popular and widely adopted, they are likely to lead to a scenario where service-oriented architectures, advocated for many years, finally become a reality. In fact, with Web services, designers and developers are led to think in the direction that "everything is a service," and that different services are autonomous and independent (as opposed to being, for example, two CORBA objects developed by the same team). As we will see, this interpretation has important implications in that it leads to decoupling applications and to making them more modular. Therefore, individual components can be reused and aggregated more easily and in different ways.

Note that not every service available through the Web is a Web service. This is a common mistake that leads to quite a lot of confusion when discussing Web services technology. There is a difference between services in the software sense and services in the general sense, i.e., activities performed by a person or a company on behalf of another person or company. Take as examples bookstores, restaurants, or travel agencies. They all provide services. In some cases, a customer might even be able to obtain such services through the Web server of the company. Strange as it might seem at first, this is not what Web services are about. A Web service is a software application with a published a stable programming interface, not a set of Web pages.

Middleware Protocols

The second aspect of the Web services approach is the redesign of the middleware protocols to work in a peer-to-peer fashion and across companies. Conventional middleware protocols, such as 2PC, 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 ad libitum. As we have seen, lack of trust and confidentiality issues often make a case against a central coordinator, and therefore 2PC 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 middleware, such as reliability and guaranteed delivery. What was then achieved by a centralized platform must be now redesigned in terms of protocols that can work in a decentralized setting and across trust domains.

Standardization

The final key ingredient of the Web services recipe is standardization. In conventional application integration, the presence of standards helped to address many problems. CORBA and Java, for example, have enabled the development of portable applications, have fostered the production of low cost middleware tools, and have considerably reduced the learning curves due to the widespread adoption of common models and abstractions. Whenever standardization has failed or proved to be inapplicable due to the presence of legacy systems, the complexity and cost of the middleware has remained quite high and the effectiveness rather low. For Web services, where the interactions occur across companies and on a global scale, standardization is not only beneficial, but a necessity. Having a service-oriented architecture and redefining the middleware protocols is not sufficient to address the application integration problem in a general way, unless these languages and protocols become standardized and widely adopted.

This problem was recognized by major software vendors that have recently showed an unprecedented commitment to standardization. Many standardization efforts in Web services have been initially driven by a small, focused group of companies, and have then been adopted by different organizations such as OASIS (Organization for the Advancement of Structured Standards) or the W3C. These consortia attempt to standardize all the different aspects of the interaction, ranging from interface definition languages to message formats and interaction protocols. We will see many examples of these standards in the following chapters.

This need for standardization is also why we speak today about *Web* services. The Web has itself been characterized by a high degree of standardization, which has allowed it to function and prosper without centralized coordination (with the exception of the Domain Name System or DNS) and has enabled its expansion at an unthinkable rate. Web technologies are now widely accepted and are very successful in enabling the interaction between humans and applications (through Web browsers and Web servers). It is therefore natural for this novel application integration technology to use the Web as its basic foundation and to try to proceed along the same, successful path taken by the Web in terms of standardization.

Observe that the commitment to standardization does not necessarily mean that there will be only one specification for each aspect of the interaction. As we will see throughout the following chapters, sometimes competing and conflicting specifications appear, possibly developed at the same time by different groups or as a result of slightly different needs. This is natural in the early days of a new technology, and does not severely limit its adoption as long as the number of such competing specifications remains relatively small, especially if they eventually converge into one commonly adopted specification. Indeed, the need for a unique solution is already bringing some order in the initially fragmented Web services landscape, and it is likely that in the end a limited number of specifications will emerge as winners.

Figure 5.5 summarizes this discussion on how Web services address the B2B integration problem. The figure shows that each party exposes its internal operations as (Web) services, which therefore act as entry points to the local information systems. The interactions between companies occur in a peer-to-peer fashion (although we will see later in this chapter that some middleware components can indeed be centralized) and take place through standardized protocols, which are designed to provide the interaction with the same properties that conventional middleware provided, but without the presence of a central middleware platform. It will be up to the Web services middleware, as we will see, to facilitate the execution of such protocols and hide from the programmer the complexities intrinsic in application integration problems.

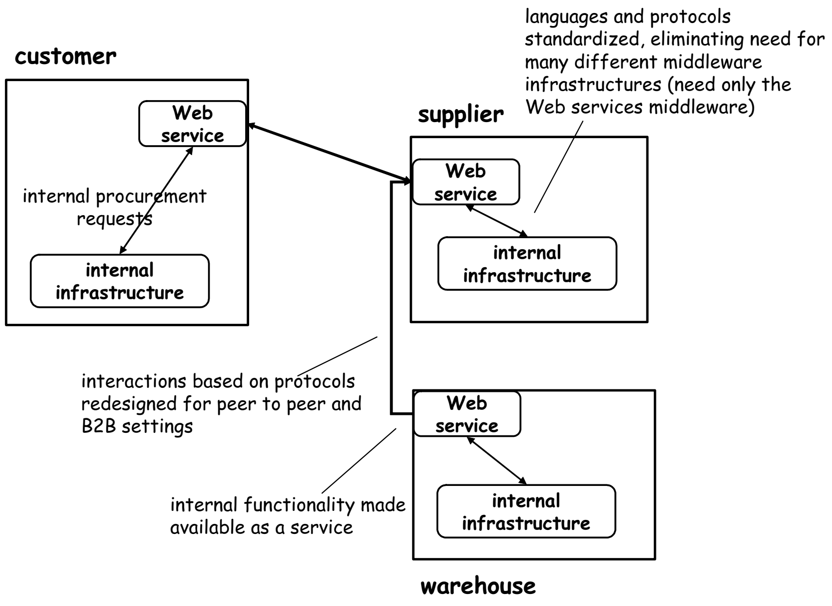
customer
Web service
internal procurement requests
internal infrastructure
supplier
languages and protocols standardized, eliminating need for many different middleware infrastructures (need only the Web services middleware)
Web service
internal infrastructure
interactions based on protocols redesigned for peer to peer and B2B settings
Web service
internal functionality made available as a service
internal infrastructure
warehouse

Fig. 5.5. Service-oriented architectures, redesigned (peer-to-peer) middleware protocols, and standardization are the main ingredients of the solution offered by Web services

5.1.6 Web services and EAI

Web services and their technologies are being developed with one specific use in mind: that of being entry points to the local information system. Thus, the primary use of a Web service is that of exposing (through the Web service interface) the functionality performed by internal systems and making it discoverable and accessible through the Web in a controlled manner. Web services are therefore analogous to sophisticated *wrappers* that encapsulate one or more applications by providing a unique interface and a Web access (Figure 5.6). Of course this is a simplification and the reality is a little more complex, but this interpretation helps in clarifying how Web services are used.

We have already encountered and discussed wrappers and adapters in the context of EAI. In particular, we have observed that wrapping components and hiding heterogeneity is the key to enabling application integration. From the perspective of the clients, the wrappers are actually the components to

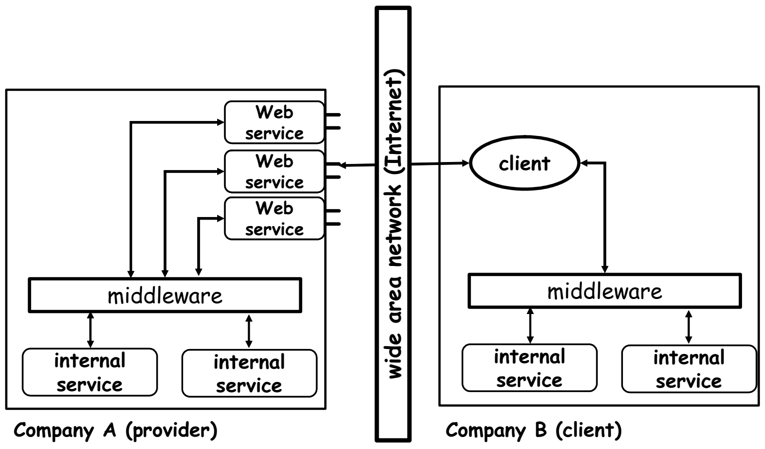

Fig. 5.6. Web services provide an entry point for accessing local services

be integrated, as they are what the integrating application can see of the underlying system.

Having homogeneous components considerably reduces the difficulties of integration. This is also true for Web services, which are indeed wrappers and are homogeneous, in that they interoperate through Web standards. As such, they constitute the base on which we can construct middleware supporting application integration on the Web, by allowing designers to avoid the problems generated by the lack of standardization typical of previous approaches.

This is an interesting aspect of Web services in spite of the fact that most of the literature on the subject is strongly biased toward B2B applications and that B2B integration is what generated the need for Web services. Indeed, Web services do not need to be accessed through the Internet. It is perfectly possible to make Web services available to clients residing on a local LAN (Figure 5.7). At the time of this writing, many Web services are used in this context, i.e., intra-company application integration rather than inter-company exchanges. This trend will be reinforced as software vendors enhance and extend their support for Web services. In fact, if software applications come out of the box with a Web services interface, then their integration is considerably simplified, as all components are homogeneous. Comparing Figure 5.7 with Figure 3.6 of Section 3.2.5, both related to EAI, the reader will notice that there is no need for adapters in this case. Nevertheless, the challenge and ultimate goal of Web services is inter-company interactions. Existing efforts around Web services go in this direction although this is clearly a long-term goal.

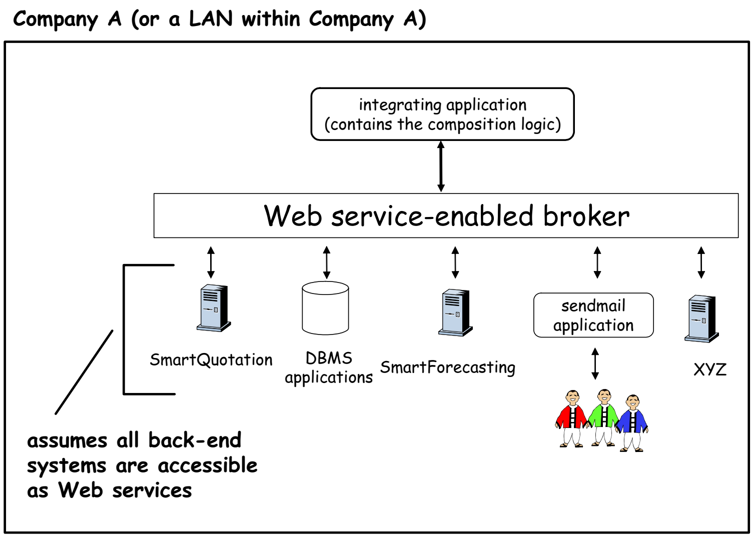
Company A (or a LAN within Company A)
integrating application (contains the composition logic)
Web service-enabled broker
SmartQuotation
DBMS applications
SmartForecasting
sendmail application
XYZ
assumes all back-end systems are accessible as Web services

Fig. 5.7. Web services can be also used within the enterprise or even a LAN, to integrate enterprise applications

5.2 Web Services Technologies

After describing what Web services are, we now go into more detail over the different aspects addressed by Web services.

5.2.1 Service Description

Since the Web services approach is centered on the notion of "service", one of the first issues to be addressed by its technology is what exactly a service is and how it can be described. Service description in conventional middleware is based on interfaces and interface definition languages. In that context, IDL specifications are needed to automatically generate stubs and to constitute the basis for dynamic binding. The semantics of the different operations, the order in which they should be invoked, and other (possibly non-functional) properties of the services are assumed to be known in advance by the programmer developing the clients. This is reasonable, since clients and services are often developed by the same team. In addition, the middleware platform defines and constrains many aspects of the service description and binding process that are therefore implicit, and they do not need to be specified as part of the service description. In Web services and B2B interactions, such implicit context is missing. Therefore, service descriptions must be richer and more detailed, covering aspects beyond the mere service interface.

The stack of Figure 5.8 illustrates the different aspects involved in Web services description. The figure essentially shows a stack of languages, where elements at higher levels utilize or further qualify the descriptions provided by elements at the lower levels.

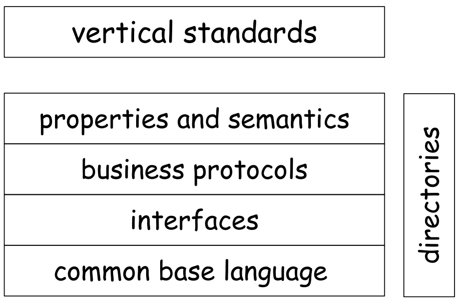

Fig. 5.8. Service description and discovery stack

- **Common base language.** The first problem to be addressed is the definition of a common meta-language that can be used as the basis for specifying all the languages necessary to describe the different aspects of a service. XML is used for this purpose, both because it is a widely adopted and commonly accepted standard and because it has a syntax flexible enough to enable the definition of service description languages and protocols.
- **Interfaces.** Interface definition languages are at the base of any service-oriented paradigm. In Web services, interface definitions resemble CORBA-like IDLs, although there are a few differences between the two. The availability of different interaction modes in the interface definition language and the XML schema-driven type system are two of them. In addition, since (as mentioned above) Web services lack an implicit context (often there is no centralized middleware), their description needs to be more complete. For example, it is necessary to specify the address (URI) of the service and the transport protocol (e.g., HTTP) to use when invoking the service. With this information, it is possible to construct a client that invokes the operations offered by a Web service. The dominant proposal for IDLs in this area is the Web Services Description Language (WSDL) [49].
- **Business protocols.** A Web service often offers a number of operations that clients must invoke in a certain order to achieve their goals. In the procurement example, the customer will have to first request a quote, then order the goods, and finally make a payment. Such exchanges between clients and Web services are called *conversations.* Service providers typically want to impose rules that govern the conversation, stating which conversations are valid and understood by the service. This set of rules is specified as part of the so called *business protocol* supported by the service (where the word "business" is used to differentiate it from a communication protocol). Business protocols are examples of why simple interface

description is not enough in Web services. In fact, to completely describe a service, it is necessary to specify not only its interface but also the business protocols that the service supports. In this regard, there are several proposals to standardize the *languages* for defining business protocols (as opposed to standardizing the protocols, discussed next). Examples are the Web Services Conversation Language (WSCL) [14] and the Business Process Execution Language for Web Services (BPEL) [7]. This is nevertheless a rather immature area in terms of standardization at the time of writing.

- **Properties and semantics.** Most conventional middleware platforms do not include anything but functional interfaces in the description of a service. Again, this is because the system context allows designers to infer other information needed to bind to a service, and because services are tightly coupled. Web services provide additional layers of information to facilitate binding in autonomous and loosely-coupled settings, where the service description is all that clients have at their disposal to decide whether to use a service or not. For instance, this may include non-functional properties such as the cost or quality of a service, or a textual description of the service such as the return policy when making a purchase. This is information that is crucial for using the service but is not part of what we traditionally understand as the interface of the service. In Web services, such information can be attached to the description of a service by using the Universal Description, Discovery, and Integration (UDDI) [20] specification. This specification describes how to organize the information about a Web service and how to build repositories where such information can be registered and queried.
- **Verticals.** All the layers explained so far are generic. They standardize neither the contents of the services nor their semantics (e.g., the meaning of a certain parameter or the effect of a certain operation). Vertical standards define specific interfaces, protocols, properties, and semantics that services offered in certain application domains should support. For example, RosettaNet [171] describes commercial exchanges in the IT world, standardizing all the aspects described above. These vertical standards complement the previous layers by tailoring them to concrete applications, further facilitating the use of standard tools for driving the exchanges. Specifically, they enable the development of client applications that can interact in a meaningful manner with any Web service that is compliant with a certain vertical standard.

5.2.2 Service Discovery

Once services have been properly described, these descriptions must be made available to those interested in using them. For this purpose, service descriptions are stored in a service directory (represented by the vertical pillar in Figure 5.8). These directories allow service designers to register new services and allow service users to search for and locate services. Service discovery can be done both at design-time, by browsing the directory and identifying the most relevant services, and at run-time, using dynamic binding techniques. These directories can be hosted and managed by a trusted entity (centralized approach) or otherwise each company can host and manage a directory service (peer-to-peer approach). In both cases, APIs and protocols are needed for clients to interact with the directory service or for the local directory services to exchange information among themselves in a peer-to-peer fashion. The above-mentioned UDDI specification defines standard APIs for publishing and discovering information into service directories. It also describes how such directories should work.

5.2.3 Service Interactions

Service description and discovery are concerned with static and dynamic binding. Once the binding problem has been addressed, a set of abstractions and tools that enable interactions among Web services is needed. In Web services, these abstractions take the form of a set of standards that address different aspects of the interactions at different levels. Figure 5.9 summarizes these different aspects by presenting them as a protocol stack since, as we will see, each aspect is characterized by one or more protocols defined on top of the lower layers. Unlike the verticals discussed in the previous section, these protocols are useful to any Web service and are therefore implemented by the Web services middleware. As such, they are transparent (for the most part) to the developers, just as the interactions established internally between two CORBA objects are hidden from the programmers, who can then focus on the business logic.

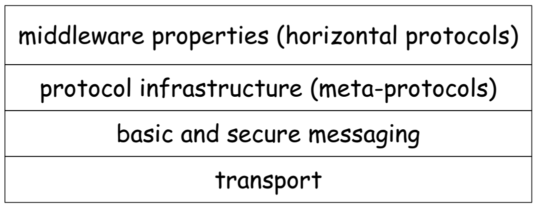
middleware properties (horizontal protocols)
protocol infrastructure (meta-protocols)
basic and secure messaging
transport

Fig. 5.9. Service interaction stack

The different types of protocols that compose the service interaction stack are:

- **Transport.** From the point of view of Web services, the communication network is hidden behind a transport protocol. Web services consider the use of a wide range of transport protocols, the most common one being HTTP.

- **Messaging.** Once a transport protocol is in place, there needs to be a standard way to format and package the information to be exchanged. In Web services, this role is played by the Simple Object Access Protocol (SOAP) [32]. SOAP does not detail what properties are associated with the exchange (e.g., whether it is transactional or encrypted). It simply specifies a generic message template to add to the top of the application data. Additional specifications standardize the way to use SOAP to implement particular features. For instance, *WS-Security* [13] describes how to implement secure exchanges with SOAP.
- **Protocol infrastructure (meta-protocols).** We have emphasized above that Web services are characterized not only by an interface but also by the business protocols they comply with. While business protocols are application specific, mcuh of the software required to support such protocols can be implemented as generic infrastructure components. For example, the infrastructure can maintain the state of the conversation between a client and a service, associate messages to the appropriate conversation, or verify that a message exchange occurs in accordance to the rules defined by the protocols. Part of the task of the infrastructure is also the execution of *meta-protocols*, which are protocols whose purpose is to facilitate and coordinate the execution of business protocols. For example, before the actual interaction can begin, clients and services need to agree on what protocol should be executed, who is coordinating the protocol execution, and how protocol execution identifiers are embedded into messages to denote that a certain message exchange is occurring in the context of a protocol. *WS-Coordination* [41] is a specification that tries to standardize these meta-protocols and the way WSDL and SOAP should be used for conveying information relevant to the execution of a protocol.
- **Middleware (horizontal) protocols.** Ideally, the Web service middleware should provide the same properties as conventional middleware (e.g., reliability and transactions), as these are likely to be useful in this context as well. Since Web services and their supporting infrastructure are distributed in nature, middleware properties that go beyond basic communication are achieved by means of standardized peer-to-peer protocols, called *horizontal* since they are generally applicable to many Web services. For example, reliability and transactions require the execution of protocols (e.g., 2PC) among the interacting entities. Just like business protocols, horizontal ones can be supported by the meta-protocols described above. However, they are likely to be hidden from the Web service developers and users, and to be entirely managed by the infrastructure. This is why we do not list them as part of the service description stack, but as part of the interaction stack: they are not used to describe a service, but to provide higher-level properties to any sort of interaction. The first protocol of this kind that has been defined is *WS-Transaction* [42], which builds upon WS-Coordination to define how to implement transactional properties when dealing with Web services.

5.2.4 Combining Web Services: Composition

A Web service can be implemented by invoking other Web services, possibly provided by different companies. For example, a reseller of personal computers may offer a Web service that allows customers to request quotes and order computers. However, the implementation of the *requestQuote* operation may require the invocation of several Web services, including for example those provided by PC manufacturers and shippers, for the latest prices and delivery schedules. A Web service implemented by invoking other Web services is called a *composite service*, while a Web service implemented by accessing the local system is called *basic service*. Observe that whether a Web service is basic or composite is irrelevant from the perspective of the clients, as it is only an implementation issue. In fact, they are all Web services and can be described, discovered, and invoked in the same way.

As the number of available Web services continues to grow and as the business environment keeps demanding newer applications that have to be rolled out according to very tight schedules, both the opportunity and the need for service composition technologies provided as part of the Web services middleware arises. These technologies resemble those of workflow systems, and have the potential to enable the rapid development of complex services from basic ones, as well as to simplify the maintenance and evolution of such complex services. In terms of standardization, this area is still rather immature, although the above-mentioned BPEL seems to be emerging as the leading service composition language.

5.3 Web Services Architecture

Now that we have described the main ingredients of Web services middleware, we show how they can be combined in an overall architecture.

5.3.1 The Two Facets of Web Services Architectures

The discussion in the previous section has shown that there are two different aspects to be considered when analyzing Web services architectures.

The first aspect is related to the fact that Web services are a way to expose internal operations so that they can be invoked through the Web. Such an implementation requires the system to be able to receive requests through the Web and to pass them to the underlying IT system. In doing this, the problems are analogous to those encountered in conventional middleware. We will refer to such an infrastructure as *internal middleware* for Web services. Correspondingly, we will use the term *internal architecture* to refer to the organization and structure of the internal middleware (Figure 5.10).

The other facet of Web services architectures is represented by the middleware infrastructure whose purpose is to integrate different Web services. We will refer to such an infrastructure as *external middleware* for Web services (Figure 5.10). Correspondingly, we will use the term *external architecture* to refer to the organization and structure of the external middleware. The external architecture has three main components:

- **Centralized brokers.** These are analogous to the centralized components in conventional middleware that route messages and provide properties to the interactions (such as logging, transactional guarantees, name and directory services, and reliability). However, as we will see, in practice the name and directory server is often the only centralized component present in Web services architectures.
- **Protocol infrastructure.** This refers to the set of components that coordinate the interactions among Web services and, in particular, implement the peer-to-peer protocols (such as the horizontal protocols and the metaprotocols discussed in the previous section) whose aim is to provide middleware properties in those B2B settings where a centralized middleware platform cannot be put in place due to trust and privacy issues.
- **Service composition infrastructure.** This refers to the set of tools that support the definition and execution of composite services.

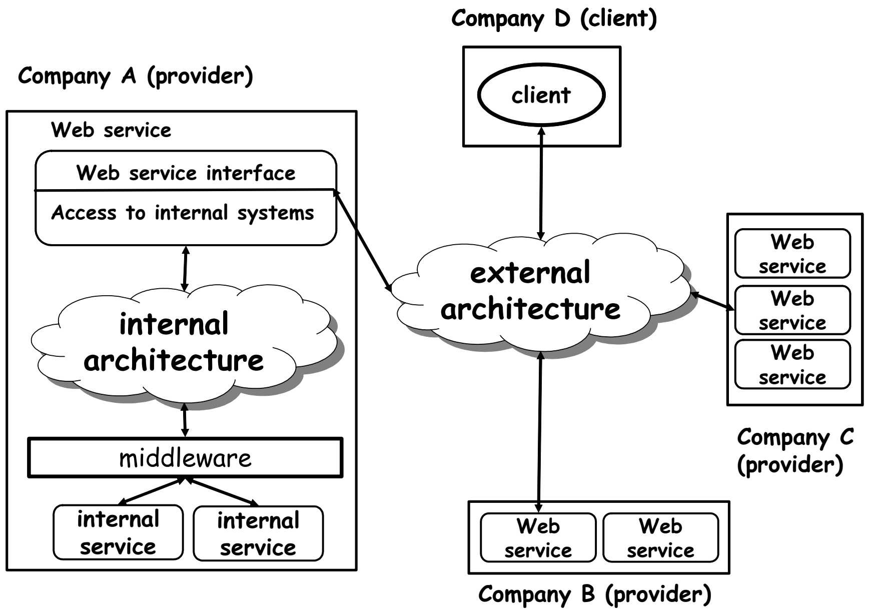

Fig. 5.10. Web services require an internal and an external architecture, along with corresponding middleware support

Observe that, in our definition, whether a component is part of the internal or external architecture is independent, to a large extent, on whether the component is deployed by the service provider or by a third party. Indeed, the same differentiation between internal and external architectures can be made when Web services are used for EAI, i.e., within the same company.

This distinction between internal and external architecture is crucial to understanding much of what is happening around Web services. There are Web services technologies and products that take care solely of the internal architecture of a Web service. There are also technologies and products that address only the external architecture of a Web service. Standardization efforts, however, mainly revolve around the external architecture. In practice both the internal and the external architecture must work together so that a Web service can make its functionality accessible to clients.

5.3.2 Internal Architecture of a Web Service

The easiest way to understand the internal architecture of Web services is to view them as yet another tier on top of the other tiers of the enterprise architecture.

In EAI, conventional middleware is used to build multi-tier architectures. In these architectures, individual programs or applications are hidden behind service abstractions that are combined into higher order programs or applications by using the functionality provided by the underlying middleware. The resulting higher order programs can in turn be hidden behind new service abstractions and can be used as building blocks for new services. Since the composition of service abstractions can be repeated ad libitum, the result is a multi-tier system in which services are implemented atop other services and basic programs. The corresponding architecture is shown in Figure 5.11.

When multiple middleware instances are stacked on top of each other, the middleware used at each level does not need to be the same. The important point is to have compatible service abstractions or to make them compatible using wrappers. The middleware simply acts as the glue necessary to make all the components in a given level interact with each other to form services that can be used by clients or higher levels in the hierarchy. Although it is not strictly necessary, usually the basic components of each middleware instance reside on a LAN and the resulting application also runs on the same LAN.

Web services or, better, the technologies supporting Web services, play the same role as conventional middleware, but on a different scale. The basis for composition is service abstractions very similar in nature to those used in conventional middleware, so that implementing a Web service essentially requires an extra tier on top of the others to enable access using standard Web services protocols. Figure 5.12 shows a typical example of such an internal architecture. Note that the figure emphasizes the fact that the implementation does not occur at the Web services layer, but within conventional middleware.

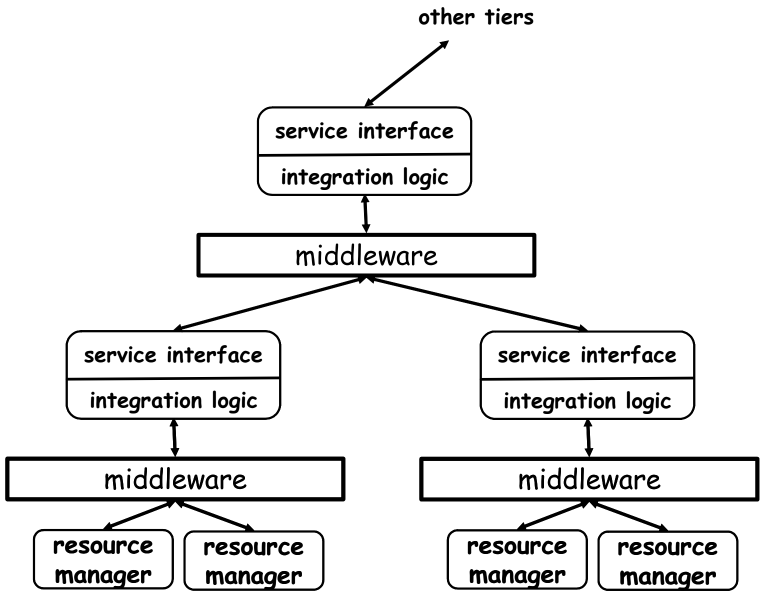

Fig. 5.11. Conventional middleware as an integration platform for basic programs and applications

As observed earlier, Web services are just wrappers. They invoke internal services that implement whatever application logic is needed, and then collect the results.

Today, much of the internal middleware for Web services revolves around packing and unpacking messages exchanged between Web services and converting them into the format supported by the underlying middleware. This is similar to how an application server maps data into HTML pages and back. Observe that the presence of this additional tier and the need for converting messages causes an overhead in processing the messages. This is also why Web services tend to be used for coarse-grained operations, where the overhead caused by the conversion is small compared to the operation execution time.

5.3.3 External Architecture of a Web Service

Using conventional middleware platforms to implement the internal architecture of a Web service is a natural step. However, what we have discussed until now is related to wrapping internal functionality as a Web service, and not to integrating these “wrappers.” This aspect, which was addressed by message brokers and workflow management systems (WfMSs) in conventional middleware, should be the job of the external middleware.

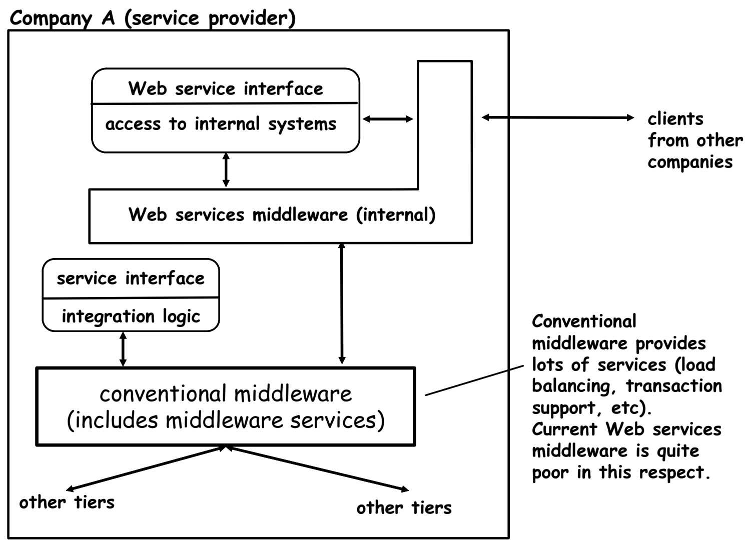
Company A (service provider)
Web service interface
access to internal systems
Web services middleware (internal)
clients from other companies
service interface
integration logic
conventional middleware (includes middleware services)
Conventional middleware provides lots of services (load balancing, transaction support, etc). Current Web services middleware is quite poor in this respect.
other tiers
other tiers

Fig. 5.12. Basic architecture of a set of Web services implemented atop a tiered architecture

However, as we have observed earlier, it is not clear where this middleware should reside. Consider as an example the implementation of name and directory services (this problem was already briefly discussed in Chapter 4). In LAN-based systems, the middleware and the applications developed using the middleware run next to each other. Thus, it is easy for the middleware to provide the necessary brokerage for name and directory services to all parties involved. In Web services, the parties can reside in different locations, and there is therefore no obvious place where to locate the middleware.

There are two solutions to this problem. One is to implement the middleware as a peer-to-peer system where all participants cooperate to provide name and directory services. Conceptually, this is a very appealing approach; but it is not obvious how to provide the degree of reliability and trustworthiness required in industrial strength systems. The other solution is to introduce intermediaries or brokers acting as the necessary middleware. Assuming we find a site somewhere in the network that we can trust and that is reliable enough, the site could act as a name and directory server for Web services. If we see such servers as part of the Web services middleware infrastructure, it follows that the participants and (part of) the middleware may reside at different locations.

Currently, there is only one type of Web services broker that has been standardized and that is used in practice, although to a very limited extent: the name and directory server. As a result, much of the existing literature on the architecture of Web services revolves around such a broker.

Figure 5.13 shows the external architecture of Web services as it is commonly understood today, with respect to centralized components [115, 83]. This representation emphasizes that abstractions and infrastructures for Web services discovery are part of the external middleware.

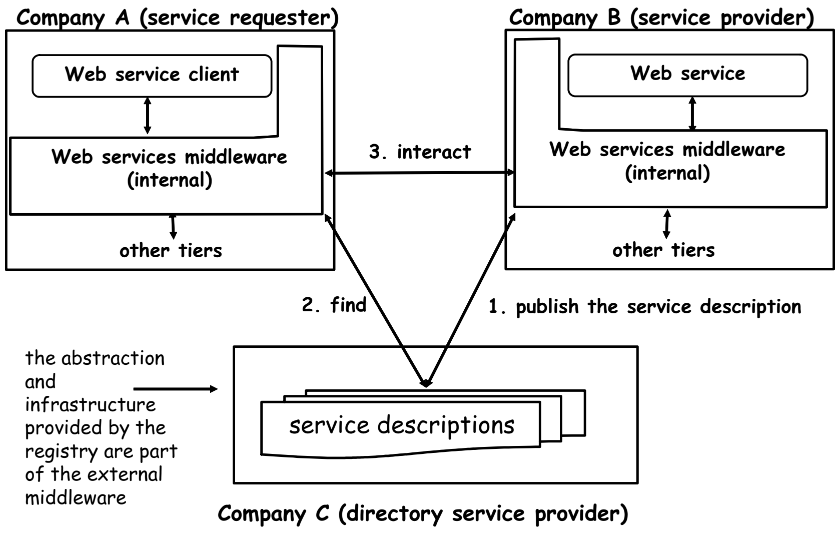
Company A (service requester)
Web service client
Web services middleware (internal)
other tiers
Company B (service provider)
Web service
Web services middleware (internal)
other tiers
3. interact
2. find
1. publish the service description
the abstraction and infrastructure provided by the registry are part of the external middleware
service descriptions
Company C (directory service provider)

Fig. 5.13. External architecture of Web services

The figure describes how service discovery takes place in a Web service environment. Ignoring the details of the protocols and the syntax underlying the corresponding exchanges, the procedure shown is a normal mechanism for service discovery. It could apply not just to Web services, but also to almost any form of middleware, including the earliest versions of RPC. The idea is for *service providers* to create Web services and to define an interface for invoking them. The service provider also has to generate *service descriptions* for those services. The service provider will then make its services known to the world by *publishing* the corresponding service descriptions in a *service registry.* The information included with the service description is used by the service registry to catalog each service and search for it when requests from *service requesters* arrive. When a service requester tries to *find* a service, it queries the service registry. The service registry answers with a service description that indicates where to locate the service and how to invoke it. The service requester can then *bind* to the service provider by invoking the service. The directory itself is very likely available as a Web service, whose address and interface are assumed to be known a priori by the requester.

What is relevant about Figure 5.13 is not so much the mechanism it describes but the fact that it depicts service discovery as the only component of

Web services middleware. Where are the transaction management features offered by a TP monitor? Where are all the different services offered by CORBA? In some cases they are simply not there, and in others they are there but not in a centralized architecture. As in the case of service registry, a centralized architecture requires brokers that provide a middleware service to be located in well-known locations. But in some cases we do not know how to build such brokers and still provide a sufficient degree of efficiency. In other cases, companies simply cannot permit a broker to mediate their business interactions.

Consider again the example of transaction management. We could use exactly the same approach of conventional middleware and have a centralized transaction broker for Web services, much in the same way the name and directory server operates. Figure 5.13 would be extended by adding a new party corresponding to the hypothetical *transaction broker*, whose implementation would resemble that of brokers in conventional middleware. Such a solution is technically feasible but opens up many issues that are very difficult to sort out in practice. For one, it requires a standard way of running transactions accepted by everybody so that transactional semantics are not violated. Since the transactional semantics at each endpoint of the transaction are dictated by the underlying middleware platform, this amounts to standardizing transactional interactions across middleware tools. There are ongoing efforts to do just that [42] but they have just started and it would be quite some time before middleware platforms follow a common transactional interface.

A much more important constraint is that this approach assumes that all participants trust the broker. Except in restricted settings, this is highly improbable. There is a big difference between consulting a service registry provided by an external company and giving away a complete track of every transaction run as part of the Web services of a company. Companies jealously guard this information and it is not reasonable to assume they will rely on some external system to keep track of it. Even the very basic functionality of service registry that is available in a centralized architecture is not widely used today, as central directories are mostly applicable in closed settings, where there is trust among the companies acting as clients, providers, and directory servers. This may change in the future, if trusted brokers appear that can play a role similar to that played by companies like Yahoo! or Lycos for online shopping.

An alternative solution is to implement our hypothetical transaction broker as a peer-to-peer system. The idea here is that each service requester will have its own transaction manager. When the service requester invokes Web services in a transactional manner, its own transaction manager becomes responsible for orchestrating the execution so that the transactional semantics are preserved. As in the centralized solution, this also requires standardized transactional interaction [42]. However, the functionality provided by this solution will likely be a subset of that provided by conventional middleware systems.

Similar arguments can be made for other components that would normally be centralized in any external architecture. This is why middleware properties are in general provided through peer-to-peer protocols and through an infrastructure that supports the execution of such protocols. Such an infrastructure is part of the external architecture, but it is typically owned and controlled by the service requestor and by the service provider, not by a third party (Figure 5.14).

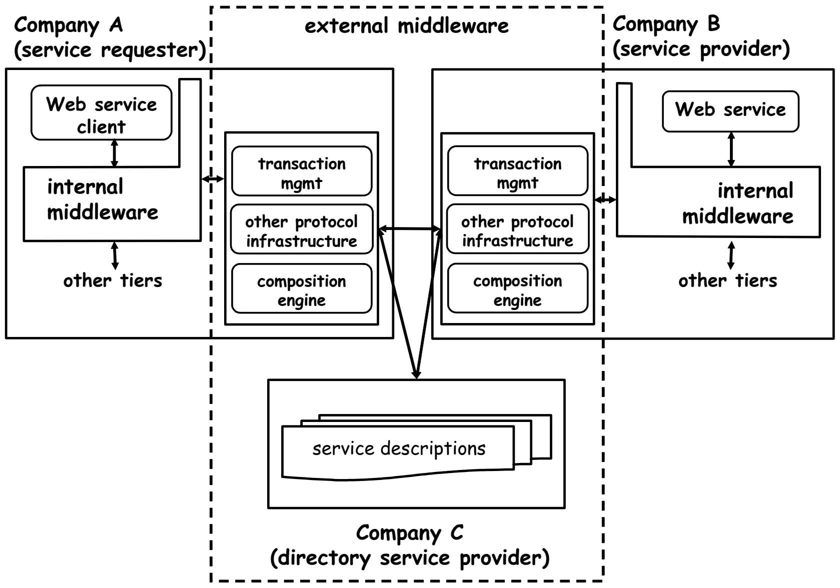

Fig. 5.14. External architecture of a Web service augmented with peer-to-peer protocol execution capabilities and with service composition

Service composition tools are another ingredient of external Web service architectures, likely to gain more importance as technology matures. We consider composition as part of the external architecture since it is about the integration of other Web services. Technically, the service composition infrastructure could be centralized. However, since the implementation is often proprietary and confidential, it is likely that such an infrastructure will be deployed by the service provider, and not by a third party (Figure 5.14).

5.4 Summary

In this chapter we have tried to clarify what Web services are and how they are being built. We have followed the W3C definition of a Web service, described as *"a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts.*

A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols." While this definition is quite open in terms of what a Web service can be, current practice indicates that Web services are being used as sophisticated wrappers over conventional middleware platforms. As such, they are an additional tier that allows middleware services to be invoked as Web services. This implies that Web services can be characterized by an internal architecture (supported by internal middleware), defining its connection with the local information systems, and an external architecture (supported by external middleware), defining how Web services discover and interact with each other. The external architecture is particularly critical in that it requires cross-organizational interactions across the Internet, often without centralized control.

The external architecture relies on standards. These standards configure to a large extent the current Web services landscape. In the chapter we have outlined many of these standards and explained how they relate to each other. In the following chapters we will describe these standards in more detail. We will start from the basics (binding and interaction) and then proceed to more advanced topics (and correspondingly more advanced forms of Web services middleware) that become possible, and even necessary, once a basic interoperability infrastructure is in place.