Gustavo Alonso • Fabio Casati
Harumi Kuno • Vijay Machiraju

Web Services

Concepts, Architectures and Applications

With 143 Figures

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Springer

1

Distributed Information Systems

Web services are a form of distributed information system. Many of the problems that Web services try to solve, as well as the design constraints encountered along the way, can be understood by considering how distributed information systems evolved in the past. As part of this evolution process, a key aspect to keep in mind is that while the technology has changed, the problems that need to be solved are to a large extent the same. Thus, the first step toward looking at Web services from the correct perspective is to develop a comprehensive understanding of distributed information systems.

In this chapter we introduce the most basic aspects of distributed information systems: their design, architecture, and communication patterns. We do so in an abstract manner, to avoid the distractions created by the myriad of details found in individual systems and products. The objective is to identify a number of design guidelines and concepts that in subsequent chapters will help the reader compare Web services with traditional information systems.

The chapter begins by addressing several aspects related to the design of an information system (Section 1.1). First we discuss the different layers involved and how they can be designed in either a bottom-up or a top-down manner. Throughout this book we will deal almost exclusively with information systems designed bottom-up. It is therefore important that we establish from the beginning what this implies in terms of how information systems are organized. We then describe possible architectures for an information system (Section 1.2). We follow a historical perspective from 1-tier to N-tier architectures placing special emphasis on why they appeared and their advantages as well as drawbacks. The relations between these different tiers are very important elements in the evolution of distributed information systems, and the issues raised here appear again and again throughout the book. The chapter concludes with a discussion on the differences between synchronous and asynchronous interaction as well as of the consequences of using each of them (Section 1.3).

1.1 Design of an Information System

In spite of the complexity and variety of distributed information systems, we can abstract a few characteristic design aspects. Understanding these aspects will be a great help when we get into detailed descriptions of how concrete systems work and operate.

1.1.1 Layers of an Information System

At a conceptual level, information systems are designed around three layers: *presentation*, *application logic*, and *resource management* (Figure 1.1). These layers may sometimes exist only as design abstractions in the minds of developers who, for performance reasons, produce nevertheless tangled implementations. In many cases, however, these layers are clearly identifiable and even isolated subsystems, often implemented using different tools.

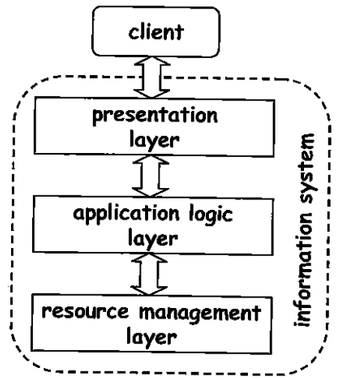

Fig. 1.1. The different layers of an information system

In general, we define the three layers as follows:[1]

- **Presentation layer.** Any information system needs to communicate with external entities, be they human users or other computers. A large part of this communication involves presenting information to these external entities and allowing the external entities to interact with the system by submitting operations and getting responses. The components of an information system that are devoted to these tasks form the *presentation layer*. The presentation layer may take many guises. For example, it could

[1] There have been many alternative definitions, but they are all equivalent. See, for instance, [86] for definitions in the context of the so-called *three-ball model*.

be implemented as a graphical user interface, or it could be a module that formats a data set into a given syntactical representation. The presentation layer is sometimes referred to as the *client* of an information system, which is not exactly correct. All information systems have clients, which are entities that use the services provided by the information system. The clients can be completely external and independent of the information system. In this case, they are not part of the presentation layer. The best examples of this design are systems accessed through Web browsers using plain HTML documents. The client is a Web browser that only displays the information prepared by the Web server.
The presentation layer of the information system in this case is the Web server and all the modules in charge of creating the HTML documents (e.g., a Java servlet). It can also be the case that the client and presentation layers are merged into one. This is typical of *client/server* systems that, being so widely used, are a source of the confusion between the client and the presentation layer. In these systems, there is an actual program that acts as both presentation layer and client. To continue with the Web browser example, *Java applets* are an instance of clients and presentation layer merged into one.
- **Application logic layer.** Information systems do more than simply deliver information and data. The vast majority of systems perform some data processing behind the results being delivered. This processing involves a program that implements the actual operation requested by the client through the presentation layer. We refer to these programs and to all the modules that help to deploy and run such programs as the *application logic layer*. We also often refer to these programs as the *services* offered by the information system. A typical example of such a service is a program that implements a withdrawal operation from a bank account. This program takes the request, checks whether there are enough funds, verifies whether withdrawal limits are exceeded, creates a log entry for the operation, performs the operation against the current balance, and gives the approval for handing out the money. All these steps are opaque to the client but reflect the logic behind a withdrawal operation *from the point of view of the service provider* (the bank, in this case). Depending on the complexity of the logic involved and on the selected implementation technique, this layer can also be referred to as *business processes*, *business logic*, *business rules*, or simply *server*. In all cases, these names apply only to particular implementations. Hence, in the following we use the term *application logic* to refer to this layer.
- **Resource management layer.** Information systems need data with which to work. The data can reside in databases, file systems, or other information repositories. A conventional *resource management layer* encompasses all such elements of an information system. From a more abstract perspective, the resource management layer deals with and implements the different data sources of an information system, independently of the

nature of these data sources. In a restrictive interpretation of the term, the resource management layer is also known as *data layer* to indicate that it is implemented using a database management system. For instance, again using the banking example, the resource management layer could be the account database of the bank. This perspective, however, is rather limiting as it considers only the data management aspects. Many architectures include as part of the resource management layer any external system that provides information. This may include not only databases, but also other information systems with presentation, application, and resource management layers of their own. By doing so, it is possible to build an information system recursively by using other information systems as components. In such architectures, the resource management layer refers to all the mechanisms and functionality used to interact with these low-level building blocks.

1.1.2 Top-down Design of an Information System

When designing an information system, a very useful strategy is to proceed *top-down*. The idea is to start by defining the functionality of the system from the point of view of the clients and of how the clients will interact with the system. This does not imply that the design starts by defining the user interfaces. Rather, it means that design can be almost completely driven by the functionality the system will offer once it becomes operational. Once the top-level goals are defined, the application logic needed to implement such functionality can then be designed. The final step is to define the resources needed by the application logic. This strategy corresponds to designing the system starting from the topmost layer (the presentation layer), proceeding downwards to the application logic layer, and then to the resource management layer (Figure 1.2).

Top-down design focuses first on the high-level goals of the problem and then proceeds to define everything required to achieve those goals. As part of this process, it is also necessary to specify how the system will be distributed across different computing nodes. The functionality that is distributed can be from any of the layers (presentation, application logic, or resource management). To simplify system development and maintenance, distributed information systems designed top-down are usually created to run on homogeneous computing environments. Components distributed in this way are known as *tightly coupled*, which means that the functionality of each component heavily depends on the functionality implemented by other components. Often, such components cannot be used independently of the overall system. That is, the design is component-based, but the components are not stand-alone (Figure 1.3).

Parallel database management systems are an example of top-down design. A parallel database is designed so that different parts of its functionality

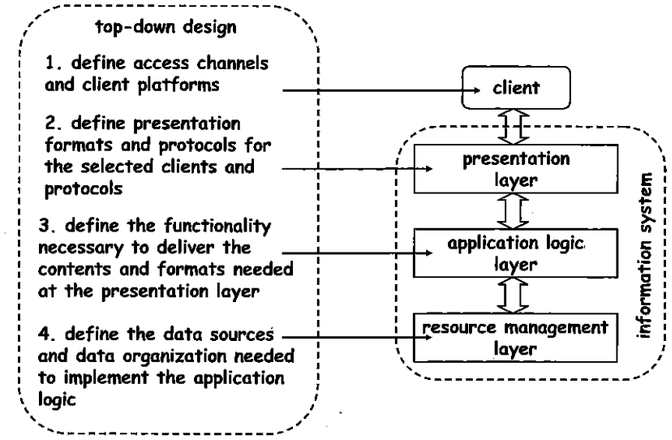

Fig. 1.2. Top-down design of an information system

can be distributed and, hence, used in parallel. Each distributed element is typically designed to work exclusively within the context of one particular parallel database and, as such, is only a subpart of a whole without meaning of its own. In the vast majority of such systems, the design focuses on how to build the system on a set of homogeneous nodes (e.g., PCs running Linux) as heterogeneity would make the design significantly more complex.

Top-down design has considerable advantages. In particular, the design emphasizes the final goals of the system and can be tailored to address both functional (what operations the system supports) and non functional issues (such as performance and availability). The drawback of top-down design is that, in its full generality, it can only be applied to systems developed entirely from scratch. As a result, few information systems are nowadays designed in a purely top-down fashion.

1.1.3 Bottom-up Design of an Information System

Bottom-up designs occur from necessity rather than choice. Information systems are built nowadays by integrating already existing systems, often called *legacy applications* or *legacy systems*. A system or an application becomes legacy the moment that it is used for a purpose or in a context other than the one originally intended. Any information system will inevitably become a legacy system at one point or another during its life time.

The problem with legacy systems is how to integrate their functionality into a coherent whole. This cannot be done top-down because we do not have the freedom of selecting and shaping the functionality of the underlying

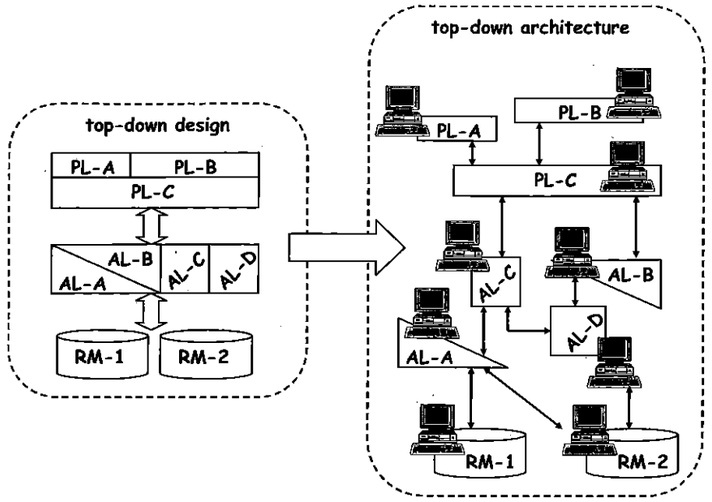

Fig. 1.3. Architecture of a top-down information system. The acronyms PL, AL, RM denote the presentation layer, application logic layer, and resource management layer. PL-A, AL-A, RM-1, and so on indicate distinct modules within each layer

systems. The functionality provided by these components is predefined and, more often than not, cannot be modified. Re-implementing the functionality provided by legacy systems so that a top-down approach can be used is in most cases not a viable option due to the development and deployment efforts required.

As a result, when legacy systems are involved, the design is mostly driven by the characteristics of the lower layers. What can be done with the underlying systems is as important as the final goal. That is, designers start by defining high-level goals as in a top-down design. Unlike in top-down designs, the next step is not necessarily defining the application logic (Figure 1.4). Rather, the next step is looking at the resource management level (which is where the legacy systems are) and figuring out the cost and feasibility of obtaining the necessary functionality from the basic components. The underlying components are then *wrapped* so that proper interfaces are made available and can be exposed to the application logic layer. Only then is it possible to design the application logic. The result is a bottom-up process where developers proceed from the resource management layers upwards toward the application logic layer and the presentation layer. In fact, bottom-up designs often begin with a thorough investigation of existing applications and processes, followed by an analysis and restructuring of the problem domain until it becomes clear which high-level objectives can be achieved.

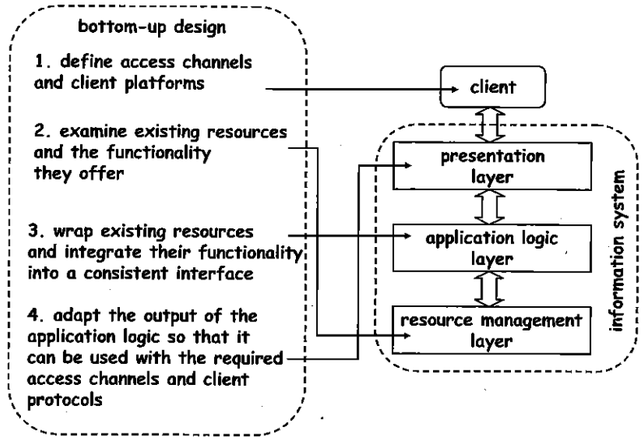

Fig. 1.4. Bottom-up design of an information system

By design, bottom-up architectures yield *loosely coupled* systems, where most components can also be used as stand-alone systems independent of the rest of the system. Often, part of the problem is how to use the legacy system as a component and, at the same time, maintain its functionality as a stand-alone system (Figure 1.5).

It does not make much sense to talk about advantages and disadvantages when discussing bottom-up designs. In many cases there is no other choice. Bottom-up design is often frequently dictated by the need to integrate underlying legacy systems. Nearly without exception, most distributed information systems these days are the result of bottom-up designs. This is certainly the case for the systems we discuss in this book. To a large extent, we find that the advantage of Web services lies in their ability to make bottom-up designs more efficient, cost-effective, and simpler to design and maintain.

1.2 Architecture of an Information System

The three layers discussed above are conceptual constructs that logically separate the functionality of an information system. When implementing real systems, these layers can be combined and distributed in different ways, in which case we refer to them not as conceptual layers, but rather as *tiers*. There are four basic types of information systems depending on how the tiers are organized: *1-tier*, *2-tier*, *3-tier*, and *N-tier*.

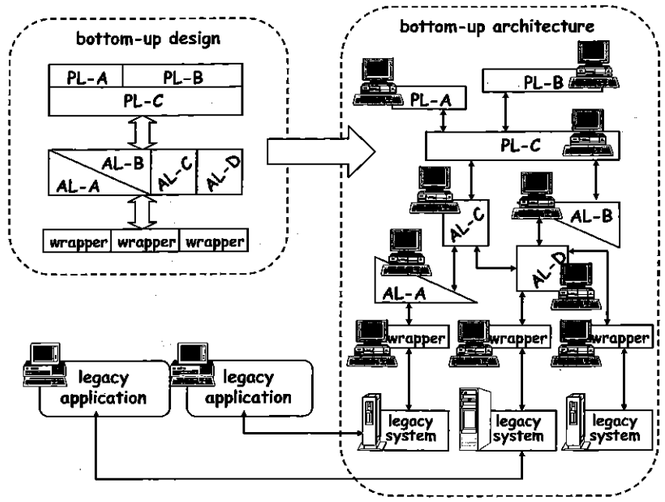

Fig. 1.5. Architecture of a bottom-up information system. The acronyms PL, AL, RM denote the presentation layer, application logic layer, and resource management layer. PL-A, AL-A, RM-1, etc. indicate distinct modules within each layer

1.2.1 One-tier Architectures

From a historical perspective, 1-tier architectures are the direct result of the computer architectures used several decades ago. These were mainframe-based and interaction with the system took place through dumb terminals that only displayed the information as prepared by the mainframe. The main concern was the efficient use of the CPU and of the system.

Information systems running on such hardware settings had no choice but to be monolithic. That is, the presentation, application logic, and resource management layers were merged into a single tier because there was no other option; hence the name 1-tier architectures (Figure 1.6). As an example of what this implies, interaction with the system was through dumb terminals, which were barely more than keyboards and computer screens. These dumb terminals were the clients. The entire presentation layer resided in the mainframe. It controlled every aspect of the interaction with the client, including how information would appear, how it would be displayed, and how to react to input from the user.

Many such systems are still in use today and constitute the canonical example of legacy systems. Because they were designed as monolithic entities, 1-tier information systems do not provide any entry point from the outside except the channel to the dumb terminals. Specifically, such systems do not

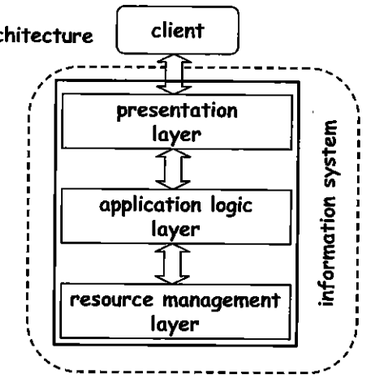

Fig. 1.6. One-tier architectures combine all layers in a single tier. In many early systems, the client in the figure was a dumb terminal

provide an application program interface (API), a stable service interface that applications or other systems can use to interact with the system in question. Therefore, they can only be treated as black boxes. When they need to be integrated with other systems, the most popular method is the dreaded *screen scraping*. This method is based on a program that poses as a dumb terminal. It simulates an actual user and tries to parse the *screens* produced by the 1-tier system. In this way it is possible to extract the necessary information in an automated manner. Needless to say, this procedure is neither elegant nor efficient. It is also highly ad hoc and, therefore, quite expensive to develop and maintain. In this regard, 1-tier architectures illustrate very well the notion of legacy system.

There are, however, obvious advantages to 1-tier systems. For instance, designers are free to merge the layers as much as necessary to optimize performance. Where portability is not an issue, these systems liberally use assembly code and low level optimizations to increase throughput and reduce response time. Moreover, since the entire system shares a single execution context, there are no penalties in the form of context switches and calls between components. Since there is no need to publish and maintain an interface, there is also no reason to invest in complex data transformations or to worry about compatibility issues. These characteristics can result in extremely efficient systems whose performance remains, in many cases, unmatched. Another advantage of historical 1-tier architectures is essentially zero client development, deployment, and maintenance cost. Of course, this was a side effect of the technology available at the time. Nevertheless, it is important to keep this aspect in mind since other architectures incur significant deployment costs because of the complexity of the clients.

The drawback of 1-tier information systems is that they are monolithic pieces of code. They are as difficult and expensive to maintain as they are efficient. In some cases, it has become next to impossible to modify the system for lack of documentation and a clear understanding of the architecture as well as for lack of qualified programmers capable of dealing with such systems [184]. Although it would be possible nowadays to develop a 1-tier system, the software industry has been moving in the opposite direction for many years. Even modern mainframe software is no longer monolithic, especially since the mainframe has been relegated to critical aspects of the system while more mundane chores are done in clusters of personal computers (PCs) or workstations. Hence, 1-tier architectures are relevant nowadays only to those unfortunate enough to have to deal with mainframe legacy systems.

1.2.2 Two-tier Architectures

Two-tier architectures appeared when as computing hardware started to be something more than a mainframe. The real push for 2-tier systems was driven by the emergence of the PC. Instead of a mainframe and dumb terminals, there were large computers (mainframes and servers) and small computers (PCs and workstations). For designers of information systems it was no longer necessary to keep the presentation layer together with the resource management and application logic layers. The presentation layer could instead be moved to the client, i.e., to the PC (Figure 1.7).

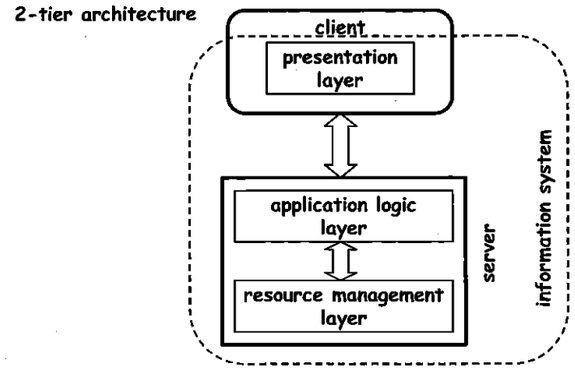

Fig. 1.7. Two-tier architectures separate the presentation layer from the other two layers. The result is a client/server system where the client has the ability to further process the information provided by the server

Moving the presentation layer to the PC achieves two important advantages. First, the presentation layer can utilize the computational power available in a PC, freeing up resources for the application logic and resource management layers. Second, it becomes possible to tailor the presentation layer for different purposes without increasing the complexity of the system. For instance, one could build one presentation layer for administration purposes and another for ordinary users. These presentation modules are independent of each other and as such can be developed and maintained separately. Of course, this is not always possible and depends on the nature of the presentation layer. Web servers, for instance, cannot be moved to the client side.

Two-tier architectures became enormously popular, particularly as *client/server* architectures [124, 162]. The *client* in client/server typically corresponds to the presentation layer and the actual client software, while the server encompasses the application logic and resource management layers. The client can take many different forms and even implement functionality that otherwise would have been in the server. Depending on how complex the client is, architectures consider *thin clients* (clients with only minimal functionality) and *fat clients* (complex clients that provide a wide range of functionality). Thin clients have the advantage of making the client easier to port, install, and maintain. They also require less processing capacity at the client machine and can therefore be used from a wider range of computers. Fat clients are much more sophisticated and offer richer functionality. The drawback is that they have a large footprint, since they are large pieces of code requiring considerable resources on the client machine. In either case, 2-tier architectures are what led many people to identify the presentation layer with the client on which it runs.

Client/server systems were involved in a positive feedback loop with many advances in computer and network hardware. As PCs and workstations became more powerful (faster CPUs, more memory and disk space, color displays, and so on), the presentation layer could be made more and more sophisticated. Increasingly sophisticated presentation layers, in turn, demanded faster and better computers and networks.

Client/server systems are also associated with many key developments in software for distributed systems. Intimately related to client/server systems is the notion of remote procedure call (RPC), discussed in Chapter 2, a programming and communication mechanism that allowed client and sever to interact by means of procedure calls. Perhaps even more importantly, client/server architectures and mechanisms such as RPC forced designers of distributed systems to think in terms of published interfaces. In fact, in order to develop clients, the server needed to have a known, stable interface. This resulted in the development of the application program interface (API), a concept that has radically changed the way information systems are designed. An API specifies how to invoke a service, the responses that can be expected, and possibly even what effects the invocation will have on the internal state of the server. Once servers had a well-known and stable APIs, it was possible to develop all sorts of clients for it. As long as the API was kept the same, developers could change and evolve the server without affecting the clients.

Through these concepts, client/server architectures became the starting point for many crucial aspects of modern information systems (Figure 1.8). The individual programs responsible for the application logic became *services* running on a *server*. The service interface defined how to interact with a given service and abstracted the details of the implementation. The collection of service interfaces made available to outside clients became the *server's* API. The emphasis on interfaces engendered the need for standardization, a process that is increasingly important today. In many respects, Web services are the latest outcome of these standardization efforts.

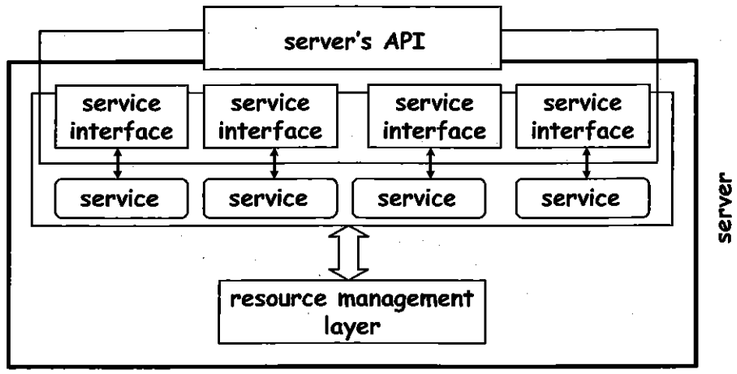

Fig. 1.8. Internal organization of the application logic layer in a 2-tier system

From a technical point of view, 2-tier systems offer significant advantages over 1-tier systems. By keeping the application logic and the resource management layers together it is still possible to execute key operations faster, since there is no need for context switches or calls between components. The same type of optimizations used in 1-tier systems are also possible in 2-tier systems. Two-tier systems also support development of information systems that are portable across different platforms, since the presentation layer is independent of the server. System designers can thus provide multiple presentation layers customized to different types of clients without worrying about the server.

The problems of client/server systems are well known, although they are not really intrinsic to the architecture itself. One obvious problem is that a single server can only support a limited number of clients, and may eventually be unable to support as many clients as needed. For example, clients need connections and authentication and require the server to maintain the context of the interaction. The server should also run the application logic and resource management layers. Typical servers do not run on mainframes but on machines that are both less expensive and less powerful, which has led to the perception

that 2-tier architectures have limited scalability. This is certainly also true of 1-tier systems, but 1-tier systems did not have to meet the performance demands of today's environments, and surviving 1-tier systems have the advantage of running today on platforms that are much more powerful than most servers in 2-tier systems.

Another typical disadvantage of 2-tier architectures is the legacy problem that arises when 2-tier systems are used for purposes other than those for which they were originally intended. These problems started when designers realized the potential of the client. Once the client became independent of the server (independent in the sense of being a separate piece of code, possibly developed by people other than those who developed the server), it could be further developed on its own. One such development was to use the client to connect to different servers, thereby integrating their services (Figure 1.9). This is in principle a good idea, but it uses the wrong architecture. For a client to connect to different servers, it needs to be able to understand the API of each server. This makes the client bigger and more complex. It also makes it dependent on two systems, thereby reducing its useful lifetime since the client must now be updated if changes are made to either server. In addition, since the two servers need not know anything about each other, the client becomes responsible for the integration. The client must combine the data from both servers, deal with the exceptions and failures of both servers, coordinate the access to both servers, and so on. In other words, an extra application layer appears but it is embedded in the client. Such an approach quickly becomes unmanageable as the client increases in both size and complexity. Furthermore, the ad hoc procedure of customizing the client to a new server must be repeated from scratch for every possible combination of servers.

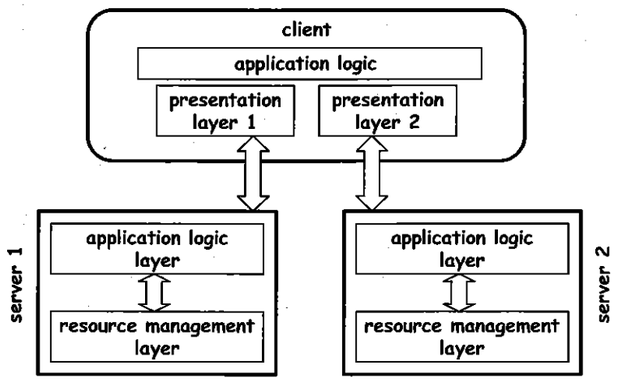

Fig. 1.9. The client is the integration engine in 2-tier architectures

Thus, in spite of their success, 2-tier architectures have acquired a reputation for being less scalable and inflexible when it comes to integrating different systems. These limitations are not intrinsic to 2-tier architectures but they are an indication of the advances in information technology that continually shift the demands on distributed systems.

1.2.3 Three-tier Architectures

The new requirements that 2-tier systems could not address were the result of the proliferation of servers with published, stable interfaces and the increase in network bandwidth provided by local area networks (LANs). The former created *islands of information* where a set of clients could communicate with a server but could not communicate with other servers. The latter made it technically possible to think about integrating different servers. What was missing was the proper architecture to do so.

Three-tier architectures are best understood when considered as a solution to this architectural problem. As we have just seen, this problem cannot be addressed at the client level. Three-tier architectures solve the problem by introducing an additional tier between the clients and the servers. This additional tier is where the integration of the underlying systems is supported and where the application logic implementing this integration resides (Figure 1.10).

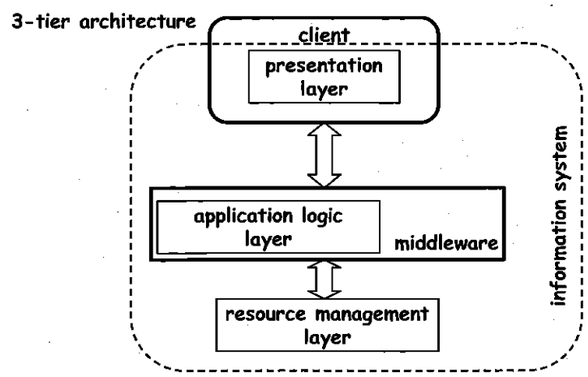

Fig. 1.10. Three-tier architectures introduce a middleware layer between the presentation and the resource management layers

Three-tier architectures are far more complex and varied than client/server systems and are therefore all the more difficult to characterize. At an abstract level, however, 3-tier architectures are usually based on a clear separation

between each of the three layers. The presentation layer resides at the client as in 2-tier architectures. The application logic resides at the middle tier. Also for this reason, the abstractions and infrastructure that support the development of the application logic are collectively known as *middleware* [26]. The resource management layer is composed of all servers that the 3-tier architecture tries to integrate. The catch in this description is that the servers at the resource management level may in turn each have their own application logic and resource management layers. From the perspective of such resource management servers, the programs running within the application logic layer of the 3-tier architecture are mere clients working in a client/server setting (Figure 1.11).

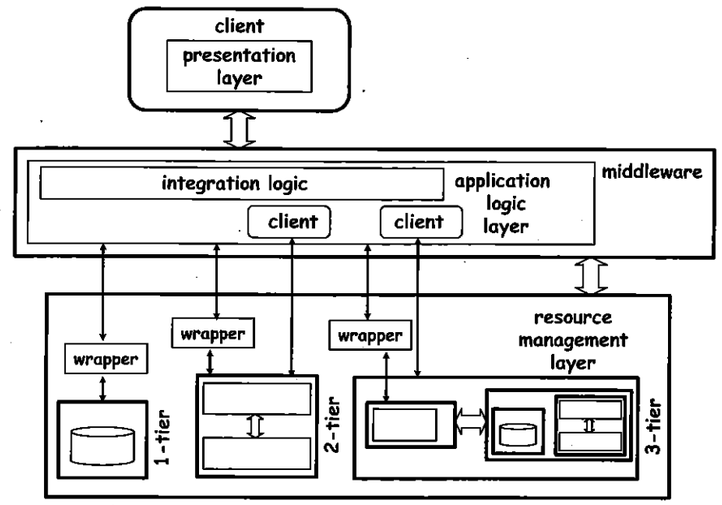

Fig. 1.11. Integration of systems with different architectures using a 3-tier approach

Although 3-tier architectures are mainly intended as integration platforms, they can also be used in exactly the same setting as 2-tier architectures. By comparing these architectures, it is possible to gain a better understanding of what 3-tier architectures imply. As already observed above, in 2-tier systems the application logic and the resource management layers are co-located, which has performance advantages and disadvantages. The advantage is that communication between these two layers is very efficient. The disadvantage is that a powerful server is needed to run both layers. If the system needs to scale, increasingly powerful servers must be obtained, which becomes very

expensive and, at a certain point, is no longer possible without resorting to a mainframe.

We can turn a 2-tier system into a 3-tier system by separating the application logic from the resource management layer [67, 89]. The advantage is that now scalability can be accomplished by running each layer in a different server. In particular, the application layer can be distributed across several nodes so that it is even possible to use clusters of small computers for that layer. Furthermore, 3-tier systems offer the opportunity to write application logic that is less tied to the underlying resource manager and, therefore, more portable and reusable. The disadvantage is that the communication between the resource manager layer and the application layer becomes much more expensive.

This comparison is far from being an academic exercise. In fact, it was a very hot debate in the database community for many years. Databases are 2-tier systems. Transaction processing monitors (TP monitors) are 3-tier systems (see Chapter 2). At a certain point, databases started to incorporate functionality available in TP monitors as part of their own application logic. This was done by allowing *stored procedures* to execute within the scope of the database in response to RPCs from clients. These stored procedures were used to implement the application logic as part of the database rather than within an intermediate layer between client and database. The result were the so-called TP-lite systems. The TP-heavy versus TP-lite debate [85] was the same as the comparison we just made between 2-tier and 3-tier architectures. It must be noted, however, that such a comparison is a bit misleading. One-tier architectures have some advantages over 2-tier systems. Two-tier systems became important when 1-tier architectures proved to be too inflexible to cope with the changes in computer hardware and networks. The same happens with 2-tier and 3-tier architectures; 2-tier architectures have some advantages over 3-tier architectures. In particular, if there is only one server, a 2-tier architecture is always more efficient than a 3-tier one. But the demand for application integration, flexible architectures, and portable application logic cannot be met with 2-tier systems. Hence the move toward 3-tier approaches.

Three-tier systems introduced important concepts that complemented and extended those already provided by 2-tier architectures. For instance, resource managers were forced to provide clear interfaces so that they could be accessed by application logic running at the middleware layer. There was also a need to make such interfaces more or less standard so that application logic code could access resource managers in a uniform manner. Such is the origin of the open database connectivity (ODBC) [135] and the java database connectivity (JDBC) [190] interfaces, which were developed so that application logic code at the middleware level could access databases in a standard manner. Thus, while 2-tier architectures forced the definition of application logic layer APIs, 3-tier architectures forced the creation of resource management APIs.

Three-tier systems are at their best when dealing with the integration of different resources. Modern middleware infrastructure provides not only the

location for developing the integration logic that constitutes the middle tier but also the functionality necessary to endow this middle tier with additional properties: transactional guarantees across different resource managers, load balancing, logging capabilities, replication, persistence, and more. By using a middleware system, the designers of the application logic can rely on the support provided by the middleware to develop sophisticated interaction models without having to implement everything from scratch. This emphasis on the properties provided at the middleware level triggered another wave of standardization efforts. For instance, a common standard emerged to be able to commit transactions across different systems (e.g., X/Open [197], Chapter 2). There were even attempts to standardize the global properties and the interfaces between middleware platforms by using an object-oriented approach (e.g., CORBA, Chapter 2).

The main advantage of 3-tier systems is that they provide an additional tier where the integration logic can reside. The resulting performance loss is more than compensated for by the flexibility achieved by this additional tier and the support that can be provided to that application logic. The performance loss when communicating with the resource management layer is also compensated for by the ability to distribute the middleware tier across many nodes, thereby significantly boosting the scalability and reliability of the system.

The disadvantages of 3-tier architectures are also due to a legacy problem. Two-tier systems run into trouble when clients wanted to connect to more than one server. Three-tier systems run into trouble when the integration must happen across the Internet or involves different 3-tier systems. In the case of integration across the Internet, most 3-tier systems were just not designed for that purpose. They can be made to communicate across the Internet but the solution is more a hack than anything else (we discuss these solutions in detail in Chapter 4). In the case of having to integrate different 3-tier systems, the problem is the lack of standards. In Chapter 5 we see how Web services try to address this problem.

1.2.4 N-tier Architectures

N-tier architectures are not a radical departure from 3-tier systems. Instead, they are the result of applying the 3-tier model in its full generality and of the increased relevance of the Internet as an access channel. N-tier architectures appear in two generic settings: linking of different systems and adding connectivity through the Internet. In the former setting, and as shown in Figure 1.11, the resource layer can include not only simple resources like a database, but also full-fledged 2-tier and 3-tier systems. In these last two cases, we say the system is an N-tier or multi-tier architecture. In the latter case, N-tier architectures arise, for example, from the need to incorporate Web servers as part of the presentation layer (Figure 1.12). The Web server is treated as an additional tier since it is significantly more complex than most presentation layers. In such systems, the client is a Web browser and the presentation layer

is distributed between the Web browser, the Web server, and the code that prepares HTML pages. Additional modules might also be necessary, such as the *HTML filter* shown in the figure, to translate between the different data formats used in each layer. The HTML filter in the figure translates the data provided by the application logic layer into HTML pages that can be sent to a browser.

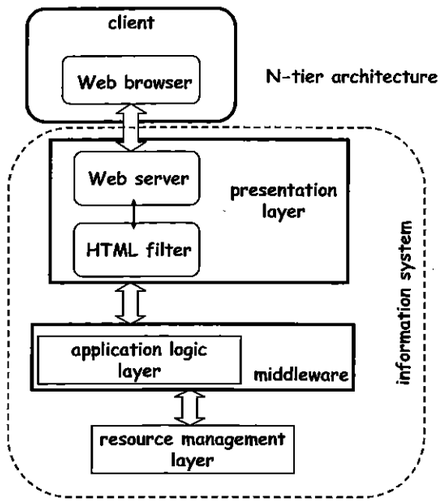

Fig. 1.12. An N-tier system created by extending a 3-tier system by adding a Web server to the presentation layer

As N-tier systems demonstrate, the architecture of most information systems today is very complex and can encompass many different tiers as successive integration efforts build systems that later become building blocks for further integration. This is in fact the main disadvantage of the N-Tier model: there is too much middleware involved, often with redundant functionality [185], and the difficulty and costs of developing, tuning, maintaining, and evolving these systems increases almost exponentially with the number of tiers. Many N-tier systems today encompass a large collection of networks, single computers, clusters, and links between different systems. As Figure 1.13 suggests, in an N-tier system it might be difficult to identify where one system ends and the next starts. Remote clients access the system via the Internet after going through a firewall. Their requests are forwarded to a cluster of machines that together comprise the Web server (clusters of machines are a very typical configuration for the layers of 3-tier and N-tier systems; they provide higher fault tolerance and higher throughput for less cost than a single machine with equivalent processing capacity). Internally, there might be additional clients spread out all over the company that also use the services of the system either through the Web server or by directly accessing the application logic implemented in the middleware. It is also very common to see the application logic distributed across a cluster of machines. There might even be several middleware platforms for different applications and functionalities coexisting in the same system. Underlying all this machinery, the often-called *back end* or *back office* constitutes the resource management layer. The back end can encompass a bewildering variety of systems, ranging from a simple file server to a database running on a mainframe and including links to additional 2-, 3-, and N-tier systems.

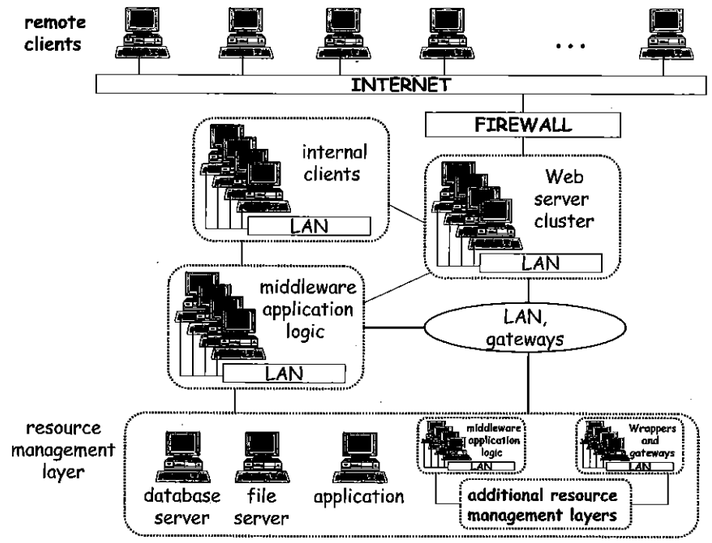

Fig. 1.13. N-tier systems typically encompass a large collection of networks, gateways, individual computers, clusters of computers, and links between systems

1.2.5 Distributing Layers and Tiers

The attentive reader may have noticed a pattern when discussing the advantages and disadvantages of each architecture. The progress from 1-tier to N-tier architectures can be seen as a constant addition of tiers. With each

tier, the architecture gains flexibility, functionality, and possibilities for distribution. The drawback is that, with each tier, the architecture introduces a performance problem by increasing the cost of communicating across the different tiers. In addition, each tier introduces more complexity in terms of management and tuning.

This pattern does not occur by chance. It is an intrinsic characteristic of the tier system. When new architectures for information systems appear, they are invariably criticized for their poor performance. This was the cause of discussions about TP-heavy versus TP-lite. This was why CORBA was criticized, and this is also why Web services are criticized. A loss in performance caused by the additional tiers must be offset by the gain in flexibility; when that happens, the new architecture prevails. The evolution from 1-tier to N-tier systems is a good reference to keep in mind when analyzing Web service technology: After all, Web services are yet another example of building a new tier on top of existing ones.

1.3 Communication in an Information System

We have so far discussed how layers and tiers are combined and distributed. The fact that we separate one tier from another assumes that there is some form of communication between all these elements. In the following, we characterize this communication.

1.3.1 Blocking and Non Blocking Interactions

The dominating characteristic of any software interaction is whether it is *synchronous* or *asynchronous*. Formally, one should actually talk about *blocking* and *non blocking* calls rather than about synchronous and asynchronous interaction. The formal definition of a synchronous system involves the existence of well-defined bounds for the time necessary to transmit messages through a communication channel [146, 125, 54]. Fortunately, for our purposes here, we can safely ignore all the formal details related to the nature of time in distributed systems. We will simply use synchronous and asynchronous systems as the accepted terms when discussing communication in an information system.

We say that an interaction is synchronous, or blocking, if the parties involved must wait for the interaction to conclude before doing anything else; otherwise, the interaction is asynchronous, or non blocking. Note that concurrency and parallelism have nothing to do with synchrony. For instance, a server can start a thread every time a client makes a request and assign that thread to that client. The server can thus deal concurrently with many clients. Synchrony, in this case, refers to how the code at the client and the code in the server thread interact. If the code at the client blocks when the call is made until a response arrives, it is a synchronous interaction. If, instead

of blocking after making the call, the client moves on to do something else, the interaction is asynchronous. Simple as these definitions are, the choice between synchronous and asynchronous interaction has important practical consequences.

1.3.2 Synchronous or Blocking Calls

In synchronous interactions, a thread of execution calling another thread must wait until the response comes back before it can proceed (Figure 1.14). Waiting for the response has the advantage of simplifying the design a great deal. It is easier for the programmer to understand as it follows naturally from the organization of procedure or method calls in a program. For instance, while a call takes place, we know that the state of the calling thread will not change before the response comes back (since the calling thread will wait for the response). There is also a strong correlation between the code that makes the call and the code that deals with the response (usually the two code blocks are next to each other). Logically it is easier to understand what happens in a synchronous system since the different components are strongly tied to each other in each interaction, which greatly simplifies debugging and performance analysis. As a result, synchronous interaction has dominated almost all forms of middleware. For instance, when the presentation layer moved to the client in 2-tier systems this was generally done through synchronous remote procedure calls. Similarly, when the application logic and the resource management layer were separated, most systems used synchronous calls for communication between both layers.

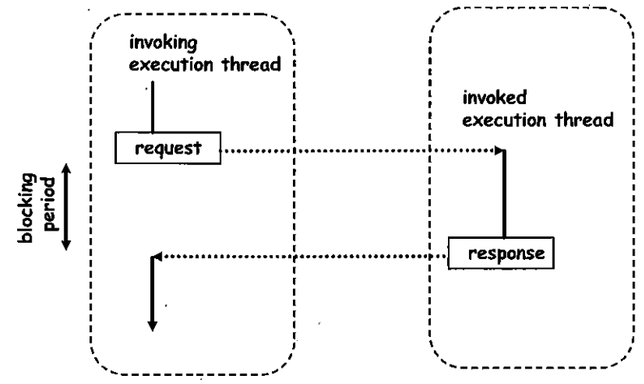

Fig. 1.14. A synchronous call requires the requester to block until the response arrives

All these advantages, however, can also be seen as disadvantages–especially if the interaction is not of the request-response type. The fact that the calling thread must wait can be a significant waste of time and resources if the call takes time to complete. Waiting is a particular source of concern from the performance point of view. For example, a waiting process may be swapped out of memory, thereby significantly increasing the time it takes to process the response when it arrives. Since this can happen at each tier, the problem is aggravated as more tiers are added to the system. Similarly, since every call results in a new connection, there is the danger of running out of connections if there are too many outstanding calls. Finally, the tight integration between the components imposed by synchronous interaction may be impossible to maintain in highly distributed, heterogeneous environments. It is also very complex to use when there are many tiers involved. In terms of fault tolerance, synchronous interactions require both the caller and the called to be online at the time the call is made and to remain operational for the entire duration of the call. This has obvious implications because of the reduced fault tolerance (for a call to succeed, both the caller and the called must work properly) and the more complex maintenance procedures (for system upgrades, everything must be taken offline since one part will not work without the other). Again, these problems become more acute as the number of tiers increases.

1.3.3 Asynchronous or Non Blocking Calls

In some cases, such as when we need to work interactively, these limitations are unavoidable. In a wide range of applications, however, it is not at all necessary to work synchronously. The alternative to synchronous interaction is asynchronous communication. One of the simplest examples of asynchronous communication is e-mail. E-mail messages are sent to a mail box where they are stored until the recipient decides to read and, eventually, answer them. The sender's process does not have to wait until a reply is received; there is not necessarily a one-to-one correspondence between messages sent and received, and indeed a response may not even be required.

Asynchronous distributed systems can be built using a similar approach. Instead of making a call and waiting for the response to arrive, a message is sent and, some time later, the program checks whether an answer has arrived. This allows the program to perform other tasks in the meanwhile and eliminates the need for any coordination between both ends of the interaction.

Historically, this model is similar to the notion of batch jobs, although the motivation behind batch jobs was different. In fact, some very primitive forms of client/server systems that preceded RPC used asynchronous communication. Later, TP monitors incorporated support for asynchronous interaction in the form of queues in order to implement batch jobs in predominantly online environments (Figure 1.15). Today, the most relevant asynchronous communication systems are *message brokers* (Chapter 2), typically used in N-tier architectures to avoid overly tight integration between multiple tiers.

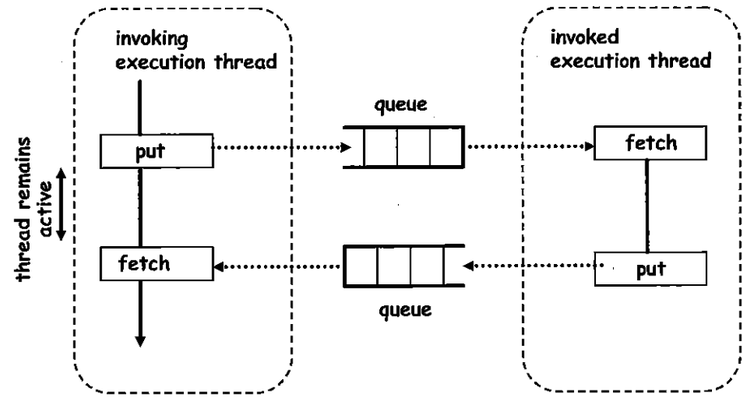

Fig. 1.15. An asynchronous call performed through queues allows the caller to continue working while the request is processed

Asynchronous communication can also be used for online interaction. In practice, asynchronous communication is used in a multitude of applications that would normally be implemented using mechanisms such as RPC, but that are prevented from doing so by design constraints (e.g., when the number of open connections would be too high if synchronous RPC were used). In such cases, the request is sent asynchronously, but the sender actively waits for and expects a response. In this mode of operation, asynchronous interaction can be effectively used to reduce problems with connection management, dependencies between components, fault tolerance, or even format representation. Asynchronous interaction is most useful, however, when the communication pattern is not of the request-response type. Examples of such applications include information dissemination (a server that periodically sends information to a collection of clients) and event notification (systems in which interaction between components does not occur through explicit calls or explicit exchanges of messages but through the publication of *events* or signals that inform those interested that a particular system state has been reached). Another example is a publish/subscribe system, where components continuously make information available by *publishing* it to the system, while other components indicate their interest on parts of the published information by *subscribing* to it. The system is then in charge of matching published information to subscriptions and delivering the information to the subscribers.

Asynchronous interaction requires messages to be stored at some intermediate place until they are retrieved by the receiver. Such intermediate storage opens up the possibility of implementing additional functionality that no longer needs to be made part of the individual components. Following this idea, many queuing systems that were used in the past simply to forward

messages between components are now being used as brokers that filter and control the message flow, implement complex distribution strategies, and manipulate the format or even contents of the messages as they transit through the queues. This is particularly useful in N-tier systems as it allows the separation of design concerns and places the logic affecting message exchanges in the queues rather than in wrappers or in the components themselves. Such separation allows changing the way messages are, e.g., filtered, translated, or distributed without having to modify the components generating and receiving the messages.

1.4 Summary

Distributed information systems have evolved in response to improvements in computer hardware and networks. This evolution can be analyzed by considering distributed information systems as a stack of three abstract layers: presentation, application logic, and resource management. When mainframes were the dominant computer architecture, the three layers were blurred into a single tier running on a centralized server. Once local area networks appeared and PCs and workstations became powerful enough, it was possible to move part of the system's functionality to the clients. The result was client/server architectures with two tiers: the presentation layer, which resided at the client, and the application logic and resource management layers, which resided at the server. Such 2-tier architectures were the first step toward modern distributed information systems. Many important concepts were developed around 2-tier systems, including RPC, service interfaces, and APIs, to name just a few.

The proliferation of information servers and the increase in network bandwidth subsequently led to 3-tier architectures, which introduce a middleware layer between the client and the server. It is in this middleware layer that the effort of integrating different information services takes place. Thanks to the middleware approach, 3-tier architectures opened the way for application integration, a higher form of distributed information system.

These ideas are crucial for understanding many of the middleware platforms that are discussed in the next two chapters. The middleware platforms we discuss there reflect this evolution and illustrate very well how system designers have tried to cope with the complexities and challenges of building systems with an increasing number of tiers. The different platforms also serve as examples of the different architectures, design alternatives, and communication trade-offs that we have discussed in this chapter.

The basic ideas described in this chapter are also very important to understand Web services and to put them in the proper context. Web services are just one more step in this evolutionary process–the latest one and probably quite significant–but a step nonetheless. In many ways, Web services are a response to problems that cannot be easily solved with 3-tier and N-tier architectures. Web services can also be seen as yet another tier on top of

existing middleware and application integration infrastructure. This new tier allows systems to interact across the Internet, with the standardization efforts around Web services trying to minimize the development cost associated to any additional tier. As indicated in the introduction of this chapter, Web services are the latest response to technology changes (such as the Internet, the Web, more available bandwidth, and the demand for electronic commerce and increased connectivity), but the problems they try to solve are still very much the same as those outlined in this chapter.

2

Middleware

Middleware facilitates and manages the interaction between applications across heterogeneous computing platforms. It is the architectural solution to the problem of integrating a collection of servers and applications under a common service interface. Simple as this description is, it still covers a wide range of situations. Obviously, integrating two databases residing on the same LAN is not the same as integrating two complete 3-tier systems residing on different branches of the same company and linked through a leased line. For the same reason, the solutions employed in the latter case cannot be the same if the systems to be integrated are owned by different companies and must communicate through the Internet.

In this and the following two chapters we examine in depth the complete range of integration possibilities. In this chapter we cover *conventional middleware* platforms as used in restricted settings such as LANs or over a collection of subsystems that are physically close to each other. In Chapter 3 we discuss integration when the systems involved are complete applications. Finally, in Chapter 4 we discuss Web technologies and their impact on application integration. In all cases we discuss middleware–often the same form of middleware, except for small extensions needed to cope with the new requirements. Accordingly, in this chapter we cover all the basic aspects of middleware and the most common middleware platforms available today. We try to follow a logical sequence that mirrors to a great extent how these different platforms were developed (Section 2.1). We start with RPC and related middleware (Section 2.2). Then, we cover modern TP monitors (Section 2.3) as transactional extensions to RPC, object brokers (Section 2.4) as the object-oriented version of RPC, object monitors (Section 2.4.6) as the result of merging TP monitors and object brokers, and message-oriented middleware (Section 2.5) as the descendant of asynchronous RPC. For each of these platforms we provide a brief historical perspective, discuss its particular approach to middleware, and compare it to other forms of middleware in order to identify its advantages and disadvantages.

2.1 Understanding Middleware

Middleware platforms fulfill several roles and appear in many guises. It can be difficult to identify the commonalities and get a comprehensive perspective of the functionality each one provides. Before discussing concrete forms of middleware, it is worthwhile to spend some time understanding the general aspects underlying all middleware platforms.

2.1.1 Middleware as a Programming Abstraction

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.

Remote procedure calls (RPCs) are a very good example of why such abstractions are helpful and of how they evolved over time. Imagine we need to write an application where part of the code is intended to run on one computer and another part must run on a different computer. A first, very basic approach, is to use sockets to open a communication channel between the two parts of the application and to use this channel to pass back and forth whatever information is needed. This is not very difficult and is a programming exercise that every student of computer science has probably completed at one time or another. Let us explore, however, what this programming model implies in real settings. First, we need to worry about the channel itself. We need to write the code that creates the channel and all the code to deal with any errors or failures that may occur on the channel. Second, we need to devise a protocol so that the two parts of the application can exchange information in an ordered manner. The protocol specifies who will be sending what, when, and what is the expected response. Third, we need to work out a format for the information being exchanged so that it can be correctly interpreted by both sides. Finally, once the issues described above have been addressed, we must develop the application that uses the communication channel. This involves including all the code necessary to deal with any errors that may occur: erroneous messages, failures of the application at the other side of the channel, recovery procedures to resume operations after failures, and so on.

Most of this work can be avoided by using middleware abstractions and tools. For example, using plain RPC, we can ignore everything related to the communication channel. RPC is a programming abstraction that hides the communication channel behind an interface that looks exactly like a normal procedure call. With RPC, the only thing we need to do is to reformulate the communication between the two parts of the application as procedure calls. The rest is done for us by the middleware implementing the RPC abstraction. The different layers hidden by the RPC abstractions are shown in Figure 2.1,

where RPC acts as the programming interface built upon the communication interface provided by the operating system (*sockets*), which in turn is built upon a stack of communication protocols.

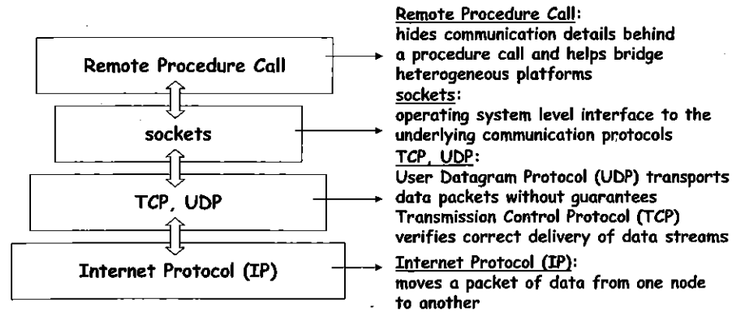

Fig. 2.1. RPC as a programming abstraction that builds upon other communication layers and hides them from the programmer

If we want the middleware to also support the handling of errors and failures, we could use transactional RPC. With transactional RPC we do not need to worry about the state of the invocation or interaction if an error occurs in the middle of it. The underlying middleware guarantees that any partial effects the call may have had are erased as part of the recovery procedure. That way, we do not need to program such clean-up procedures ourselves. Still, this might not be enough (in fact, it isn't). If the application or parts of it fail, transactional guarantees make sure that no unwanted side effects have taken place. However, the application does not necessarily remember when it failed, what things it had already done, and what was left to do. Again, this is something that we would need to program ourselves if we did not use middleware. For this purpose, many modern middleware platforms provide automatic persistence for applications so that, in the event of a failure, they do not lose their state and can quickly resume where they left off.

These are only a few examples of what middleware programming abstractions provide. In all cases, the choice for the programmer is between coding all this functionality from scratch or using a middleware platform. This choice is not always as trivial as it may seem. From the point of view of programming support, the more the middleware can do, the better. Unfortunately, the more the middleware does, the more complex and expensive it becomes. In practice, any non trivial distributed application is built using middleware, but the key to using middleware successfully is to use exactly what one needs and no more. This is not always feasible, since middleware functionality comes bundled in complete platforms. As a result, sometimes designers are better off using a

less sophisticated form of middleware that they can extend as needed rather than using a full-featured platform that may offer much more than what is actually needed. The advantages that the middleware provides in terms of programming abstractions are often offset by the cost and complexity of the infrastructure supporting these very same programming abstractions.

2.1.2 Middleware as Infrastructure

Behind the programming abstractions provided by the middleware there is a complex software infrastructure that implements those 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 (such as, e.g., Web servers). This makes middleware platforms very complex software systems.

RPC is again the best example to understand what this infrastructure entails and how it has evolved over the years toward increasing size and complexity (see the next section for more details on RPC). As a programming abstraction, RPC is very simple. Yet, in its simplest form it already requires some basic infrastructure such as an interface definition language, an interface compiler!compiler, and a number of libraries that provide the functionality required to make remote procedure calls. Part of this infrastructure is related to the development of applications using the RPC programming abstraction (the interface definition language and the interface compiler). The other part of the infrastructure is used at run time to execute calls to remote procedures. A similar division between development and run-time support can be found in all forms of middleware.

When a programming abstraction, in this case RPC, turns out to be useful and widely adopted, it starts to be extended and enhanced in different ways. Each extension entails either additional development or run-time support, and has to be general enough to be useful to many different users. These extensions make the programming abstraction more expressive, more flexible, and easier to use. Often, they also make the system more complex and the learning curve steeper. For instance, if RPC clients need to authenticate themselves before they can start to make calls, the infrastructure needs to incorporate mechanisms for dealing with authentication. Similarly, if the addresses of the procedures to invoke are determined dynamically, then a name and directory service is needed. When used, such services need to be installed and kept running in order for RPC to work, thereby adding one more component to worry about. In addition, using a naming service requires additional programming abstractions to deal with the service itself, thereby making the programming interface more complex. As another example, some designers need to have a tighter control of what goes on during an RPC interaction. For these programmers, RPC can be extended with further abstractions that allow direct access to low-level details like the transport protocol used or the parameters

of that protocol. The result is additional machinery and a longer list of programming abstractions. More sophisticated users may also need support for multi-threading, automatic logging, transactions, asynchronous RPC, and so on. Once all the required features are accounted for, the programming abstractions have become merely the visible part of a middleware platform that may provide much more than the initial programming abstractions it intended to support. In some cases, part of this functionality may assume a life of its own and become a separate middleware infrastructure. This happened, for instance, with the persistent queuing systems that all TP monitors used to incorporate. Today, such queuing systems have become message brokers.

2.1.3 Types of Middleware

In this chapter we discuss the following forms of middleware:

- **RPC-based systems**. RPC is the most basic form of middleware. It provides the infrastructure necessary to transform procedure calls into remote procedure calls in a uniform and transparent manner. Today, RPC systems are used as a foundation for almost all other forms of middleware, including Web services middleware. For example, the SOAP protocol, often used to support interactions among Web services and discussed in Chapter 6, provides a way to wrap RPC calls into XML messages exchanged through HTTP or some other transport protocol.
- **TP monitors**. TP monitors are the oldest and best-known form of middleware. They are also the most reliable, best tested, and most stable technology in the enterprise application integration arena. In a very gross simplification, TP monitors can be seen as RPC with transactional capabilities. Depending on whether they are implemented as 2-tier or 3-tier systems, TP monitors are classified into *TP-lite* and *TP-heavy* monitors. *TP-lite* systems typically provide an RPC interface to databases. *TP-heavy* monitors are the quintessential middleware platform, providing a wealth of tools and functionality that often matches and even surpasses that of operating systems.
- **Object brokers**. RPC was designed and developed at a time when the predominant programming languages were imperative languages. When object-oriented languages took over, 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. The most popular class of object brokers are those based on the Common Object Request Broker Architecture (CORBA), 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 a 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.
- **Message-oriented middleware**. The first forms of RPC middleware had to acknowledge the fact that synchronous interaction was not always needed (see Chapter 1 for a discussion of this topic). Initially this was solved by providing asynchronous RPC. Later on, TP monitors extended this support with persistent message queuing Systems. At a certain point, designers realized that these queuing systems were useful in their own right, and they became middleware platforms on their own under the general name of message-oriented middleware (MOM). Such platforms typically provide transactional access to the queues, persistent queues, and a number of primitives for reading and writing to local and remote queues.
- **Message brokers**. Message brokers are a distinct kind of message-oriented middleware that has the capability of transforming and filtering messages as they move through the queues. They can also dynamically select message recipients based on the message content. In terms of basic infrastructure, message brokers are just queuing systems. The only difference is that application logic can be attached to the queues, thereby allowing designers to implement much more sophisticated interactions in an asynchronous manner.

In this list we omit two important entries–workflow management systems and application servers. Workflow management systems are addressed in Chapter 3, and application servers are discussed in Chapter 4.

2.1.4 Middleware Convergence

It has often been argued that there is too much middleware with competing and overlapping functionality [26, 6, 185]. This refers not so much to the programming abstractions supported, but rather to the underlying infrastructure. The problem arises when, to take advantage of the different programming abstractions, different middleware platforms are used to integrate the same systems. Since each middleware platform comes with its own fixed infrastructure, using several of them amounts to dealing with several such infrastructures. The irony is that a significant percentage of the underlying infrastructure is probably identical across all platforms (RPC-based, in most cases), but is

made incompatible by the tailoring of the infrastructure in each product. As a result, there are two obvious trends that can be observed in the evolution of middleware platforms. One is the consolidation of complementary platforms. The other is the emergence of massive product suites that offer, in a single environment, many different forms of middleware.

Object monitors are an example of consolidation. They combine the features and performance of TP monitors with the object-oriented interfaces of object brokers. Another example is the extension of TP monitors or message brokers with support for workflow languages (discussed in detail in Chapter 2) in addition to third-generation programming languages such as C.

There are already many examples of massive product suites combining all the middleware platforms offered by a single vendor. The idea is to simplify multi-tier architectures by resorting to systems that were specifically designed to work together from the start (or where a significant effort has been made to simplify the integration). This is still an ongoing process, but one that is welcome given the integration problems designers face in practice. Essentially every big player in the area has taken this path. The level of integration is certainly not perfect at this stage, but it is reasonable to expect that it will improve and become seamless as new versions of such *all-purpose* middleware platforms appear.

2.2 RPC and Related Middleware

RPC is the foundation underlying the vast majority of middleware platforms available today. In the following, we describe RPC in detail.

2.2.1 Historical Background

RPC was introduced at the beginning of the 1980s by Birell and Nelson as part of their work on the Cedar programming environment [28]. The original paper presented RPC as a way to transparently call procedures located on other machines. This mechanism immediately became the basis for building 2-tier systems, which inherited much of the notation and assumptions used in RPC. In particular, RPC established the notion of client (the program that calls a remote procedure) and server (the program that implements the remote procedure being invoked). It also introduced many concepts still widely used today: *interface definition languages* (IDL), *name and directory services*, *dynamic binding*, *service interface*, and so on. These concepts appear in essentially all forms of middleware and are also a big part of Web services. For instance, about 20 years ago the ANSA Testbench platform provided programmers with a *trading service* to dynamically select a remote procedure based on criteria other than the procedure's signature [9], analogously to what modern name and directory services do.

The strength of RPC was that it provided a clean way to deal with distribution. Moreover, it was based on a concept that programmers at the time knew very well: the *procedure*. RPC made it possible to start building distributed applications without having to change the programming language and the programming paradigm. Initially, RPC was implemented as a collection of libraries. Later on, as more functionality was added, it became a middleware platform on its own. All that was required for a program to become a component of a distributed system was for it to be compiled and linked with the correct set of RPC libraries.

From the very beginning, RPC raised very tough questions about the design and architecture of distributed information systems [196, 54]. Many of these questions are still open. A very heated debate at the time was, for instance, whether RPC should be transparent to the programmer or not. Arguments in favor were based on the simplicity of the concept and the fact that programmers did not need to deal with distribution directly. Arguments against revolved around the fact that including a remote call in a program deeply changed the nature of that program. Forcing programmers to use special constructions for RPC was a way of making them aware of the functional (e.g., using a remote method) and non functional (e.g., performance, reliability) implications of distribution, thereby reducing the opportunities for errors. Most modern RPC systems use a transparent approach.

Today, RPC is at the heart of most distributed information systems [54], and comes in different flavors. For instance, remote method invocation (RMI) is identical to RPC but applies to object methods rather than procedures. Similarly, *stored procedures* are an instance of RPC used to interact with databases. In some cases RPC is used as a low-level primitive used by the system to implement more sophisticated forms of interaction. In other cases, programmers may use it directly to implement distributed applications. However it is used, the mechanisms underlying RPC as well as its implementation have become an intrinsic part of middleware, enterprise application integration, and even Web services.

2.2.2 How RPC Works

The development of a distributed application with RPC is based on a very well defined methodology. For simplicity, we assume here that we want to develop a server that implements a procedure to be used remotely by a single client.

The first step is to define the interface for the procedure. This is done using an interface definition language (IDL) that provides an abstract representation of the procedure in terms of what parameters it takes as input and what parameters it returns as a response. This IDL description can be considered the specification of the services provided by the server. With the IDL description in hand, we can proceed to develop the client and the server (Figure 2.2).

The second step is to compile the IDL description. Any RPC implementation and any middleware using RPC or similar concepts provides such an interface compiler. Typically, compiling the IDL interface produces:

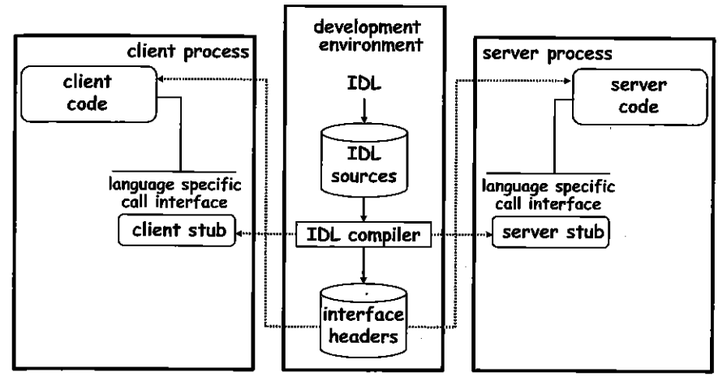

Fig. 2.2. Developing distributed applications with RPC

- **Client stubs.** Each procedure signature in the IDL file results in a client *stub*. The stub is a piece of code to be compiled and linked with the client. When the client calls a remote procedure, the call that is actually executed is a local call to the procedure provided by the stub. The stub then takes care of locating the server (i.e., *binding* the call to a server), formatting the data appropriately (which involves *marshaling* and *serializing* the data[1]), communicating with the server, getting a response, and forwarding that response as the return parameter of the procedure invoked by the client (Figure 2.3). In other words, the stub is a placeholder or *proxy* for the actual procedure implemented at the server. The stub makes the procedure appear as a normal local procedure (since it is part of the client code). The stub, however, does not implement the procedure. It implements all the mechanisms necessary to interact with the server remotely for the purposes of executing that particular procedure.
- **Server stubs.** The server stub is similar in nature to the client stub except that it implements the server side of the invocation. That is, it contains the code for receiving the invocation from the client stub, formatting the data

[1] Marshaling involves packing data into a common message format prior to transmitting the message over a communication channel, so that the message can be understood by the recipient. Serialization consists of transforming the message into a string of bytes prior to sending the message through a communication channel.

as needed (which, mirroring the operations at the client stub, involves *deserializing* and *unmarshaling* the call), invoking the actual procedure implemented in the server, and forwarding the results returned by the procedure to the client stub. As with the client stub, it must be compiled and linked with the server code.

- **Code templates and references.** In many programming languages, it is necessary to define at compile time the procedures that will be used. The IDL compiler helps with this task by producing all auxiliary files needed for development. For instance, the first versions of RPC were developed for the C programming language. In addition to the client and server stubs, the IDL compiler also generated the header files (that is, the **.h* files) needed for compilation. Modern IDL compilers even go a step beyond: they can also generate templates with the basic code for the server, such as programs containing only procedure signatures but no implementation. The programmer only needs to add the code that implements the procedure at the server and code the client as needed.

The system that results from this process is shown in Figure 2.3.

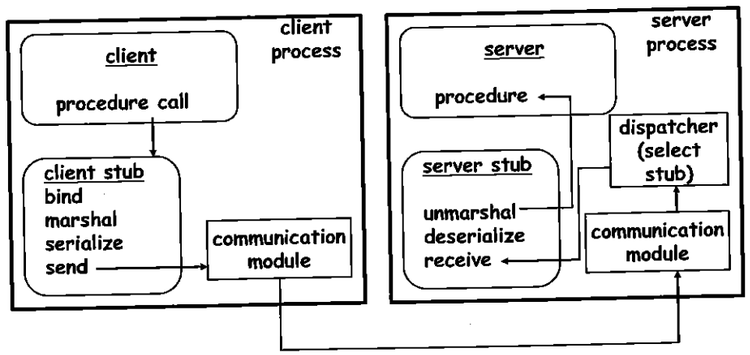

Fig. 2.3. Basic functioning of RPC

RPC stubs can handle all of the network programming details, including timeouts and retransmissions if an unreliable protocol such as the user datagram protocol (UDP) is used. If more control is needed, RPC infrastructures typically provide different RPC interfaces, ranging from very simple to very sophisticated ones, that can be invoked to configure how the interaction should occur. These interfaces are used to implement different types of stubs. The simple interfaces usually provide only three primitives for using RPC: *procedure registration* (making it known to clients that a procedure is available for remote invocation), *procedure call* (actual invocation of the remote

procedure), and *procedure call using broadcast* (same as procedure call, but using a broadcast primitive). The advanced interfaces give programmers finer control over the transport protocols and binding procedures.

2.2.3 Binding in RPC

In order for a client to make an RPC, it must first locate and bind to the server hosting the remote procedure. *Binding* is the process whereby the client creates a local association for (i.e., a *handle* to) a given server in order to invoke a remote procedure. Binding can be either *static* or *dynamic*. In static binding, the client stub is hardcoded to already contain the handle of the server where the procedure resides. The form of this handle depends on the environment: It might be an IP address and port number, an Ethernet address, an X.500 address, and so on. When the client invokes the procedure, the stub simply forwards the call to that server.

The advantages of static binding are that it is simple and efficient. No additional infrastructure is needed beyond the client and the server stubs. The disadvantages are that the client and the server become tightly coupled. That is, if the server fails, the client will not work. If the server changes location (because of upgrades, maintenance, etc.), the client must be recompiled with a new stub that points to the right location. Finally, it is not possible to use redundant servers to increase performance because the clients are bound to specific servers. Load balancing must take place at the time that clients are distributed and developed, or else the system may exhibit severe skews in load, with some servers answering most requests while others remain idle.

These disadvantages led the designers of RPC to develop dynamic binding. *Dynamic binding* enables clients to use a specialized service to locate appropriate servers (Figure 2.4). Similar to the consequence of adding new tiers to information systems (Chapter 1), dynamic binding simply adds a layer of indirection to gain flexibility at the cost of performance. This additional layer of indirection, generally called *name and directory server* (also known as a *binder* in earlier RPC implementations), is responsible for resolving server addresses based on the signatures of the procedures being invoked. Thus, when the client invokes a remote procedure, the client stub asks the directory server for a suitable server to execute that procedure. The directory server responds with the address of a server. With that address, the client stub proceeds to complete the invocation.

Dynamic binding creates many opportunities for improving RPC interaction. The directory server could, for instance, keep track of which servers have been invoked and perform load balancing so as to use resources as efficiently as possible. If the server changes location, it is enough to change the entry in the directory server for requests to be rerouted to the new location. The result is the decoupling of the client and the server, which adds a great deal of flexibility when deploying and operating the system. The cost of this flexibility is additional infrastructure such as a directory server, a protocol for interacting

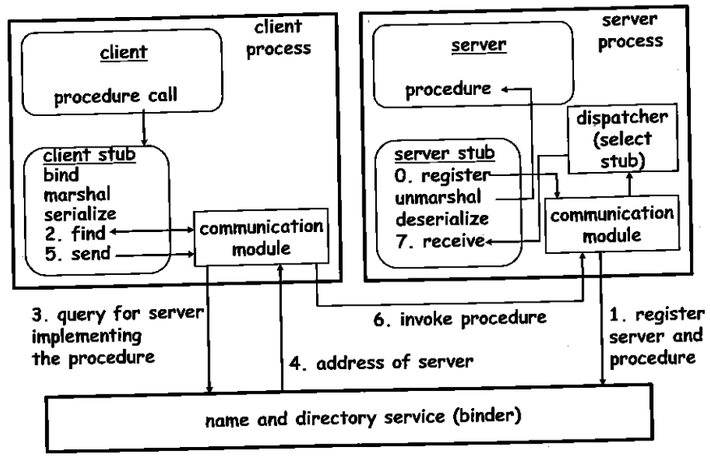

Fig. 2.4. Dynamic binding allows a server to register the procedures it implements (steps 0 and 1). A client can then ask for the address of the server implementing a given procedure (steps 2 and 3), obtain the address from the binder (step 4), and make the call to the procedure (steps 5, 6, and 7), which then proceeds as in the static case

with the directory server, and primitives for registering procedures with the server. These operations are, in principle, hidden from the programmer since they are performed by the stubs. The server stub, for instance, registers the server procedures when the server is started. Similarly, the client stub deals with the directory server when calls are made.

The directory server uses the IDL specification of the procedures to perform dynamic binding. That is, the binding is made based on the procedure's *signature*. However, there is nothing that prevents the binding from happening based on other, possibly non functional criteria. Directory servers supporting sophisticated selection mechanisms were called *traders*. Such ideas are well-explored in the context of RPC; there are even RPC platforms that support such sophisticated binding procedures [54].

RPC systems, however, predominantly worked under the assumption that the implementation of the distribution mechanism should be hidden from the programmer by the stubs. In other words, the programmer is not necessarily aware of whether binding is dynamic or static. This is decided when the system is designed and the stubs created. Such an approach is the basis for the separation between programmers of client and server functionality.

2.2.4 RPC and Heterogeneity

One of the original design goals of RPC was to serve as a mechanism for bridging heterogeneous systems. Because the bulk of the work of dealing with distribution is performed by the stubs, one can develop very complex stubs that enable client programmers to develop software on platforms other than those used by server programmers. That is, the stubs can be used to hide not only the distribution but also the heterogeneity. This was a very appealing feature of RPC. Recall that RPC was instrumental to the development of client/server systems. As we saw in Chapter 1, 2-tier or client/server architectures appeared when information systems started to move away from the mainframe approach. PCs, however, were not powerful enough to run a server; they could only support the client application. Servers required more powerful machines that used CPUs and operating systems (e.g., UNIX) different from those available for clients (e.g., DOS). Hence, when they moved to client/server architectures, information systems became not only distributed but also heterogeneous.

The problem of dealing with heterogeneous client and server systems is that, potentially, there are many different platforms on which the clients and servers could run, and there are many programming languages in which the clients and the servers could be developed. A naive approach would use a different client and server stub set for every possible combination of platforms and languages (i.e., $2 \times n \times m$ stubs are needed for n client and m server platforms). A much more efficient alternative is to use some form of intermediate representation so that clients and servers only need to know how to translate to and from this intermediate representation. The problem of dealing with heterogeneity then becomes a matter of developing client and server stubs for the corresponding platforms (which requires only $n + m$ stubs).

RPC uses IDL not only to define interfaces but also to define the mapping from concrete programming languages to the intermediate representation used in that particular RPC system. One of the reasons for using an IDL rather than simply adopting the syntax of a given programming language (although earlier IDLs showed a marked preference for programming languages such as C) is that the intermediate representation of IDL enables clients and servers to ignore differences in terms of machine architecture, programming languages, and type systems used by each one of them. As a platform independent representation, IDL serves not only for specifying interfaces but also for defining the intermediate representation for data exchanges between clients and servers. This intermediate representation specifies how parameters should be represented and organized before being sent across the network. For instance, Sun's IDL (called XDR) requires calls to be encoded into 4-byte objects, each byte being an ASCII code [54, 183]. It further defines how to use these objects to store the name of the procedure and the values of the parameters. Once marshaled into this format, the data can be serialized and sent through the

network. At the other side, the message can be deserialized and unmarshaled to obtain the parameters of the call in whatever internal format is used.

2.2.5 Extensions to RPC

Conventional RPC shares many of the characteristics of a local function call. First, it uses a synchronous, procedural approach. The execution, including the execution of subroutines, is sequential. A client can have only one outstanding call at a time and is blocked until the call returns. At the server, multiple threads can be used to handle several clients concurrently, but each thread is dedicated exclusively to one client. Very early on, designers realized that the limitations of this model prevented the implementation of all forms of interactions needed in a distributed information system. As a result, many extensions to the basic RPC mechanisms have been implemented. The motivation behind these extensions to RPC lies in the same changes that triggered the move toward 3-tier architectures, i.e., the increasing availability of servers with published, stable interfaces spurred the need for architecture and infrastructure to support server integration. In fact, extensions to RPC have resulted in different middleware platforms such as TP monitors, object monitors, queuing systems, or message brokers. Here we discuss *asynchronous RPC* (which underlies message-oriented middleware and message brokers), while in Section 2.3.2 we discuss *transactional RPC* (as the basis for TP monitors).

Asynchronous RPC was one of the first extensions to RPC to support non blocking calls . It allows a client to send a request message to a service without waiting for the response. The communication handler returns control to the client program immediately after sending request, and thus the client thread is not blocked when it makes a call. This allows it to have multiple outstanding calls while still doing other work.

This is relatively easy to implement by changing the way the client stub works. Instead of one single entry point to invoke the procedure, the stub provides two entry points–one to invoke the procedure and one to obtain the results of the invocation. When the client calls the remote procedure, the stub extracts the input parameters and immediately returns control to the client, which can then continue processing. The stub, in the meantime, makes the call to the server and waits for a response. This behavior can be easily achieved by using threads at the client without the programmer of the client having to be aware of it. Later, the client makes a second call to the stub to obtain the results. If the call has returned, the stub placed the results in some shared data structure and the client picks it up from there. If the call has not yet returned, the client receives an error that indicates that it should try again later. If there was an error with the call itself (timeout, communication error, etc.) the client receives a return value that indicates the nature of the problem. In essence, the stubs execute synchronously while giving the illusion of asynchronous execution to the clients and servers using them.

The original motivation for asynchronous interaction in RPC systems was to maintain backward compatibility, so to speak, with functionality provided by 1-tier architectures. In particular, 1-tier architectures typically supported *batch* operations, which cannot be implemented with synchronous RPC; asynchronous RPC provided a means for implementing such operations. Later, designers recognized additional advantages to asynchronous RPC, but also encountered the challenges it posed to programmers. To be truly useful, asynchronous RPC needed a much more sophisticated infrastructure than the stubs provided. For example, because less information about the connection is maintained and requests are decoupled from replies, it can be more difficult to recover from a failure involving asynchronous RPC than traditional RPC. This infrastructure was developed as part of the queuing systems used in TP monitors and later evolved into what we know as message brokers. Because RPC initially lacked the necessary additional support, the use of asynchronous RPC was not as general then as it is today. We revisit this issue in detail when discussing message-oriented middleware (Section 2.5) and message brokers (Section 3.2).

2.2.6 RPC Middleware Infrastructure: DCE

Even without considering extensions to the basic RPC mechanism, RPC already assumes a certain degree of infrastructure for both development and execution. Some of this infrastructure is minimal, particularly in basic implementations of RPC such as Sun's RPC [183]. In other cases, it is quite extensive as in the Distributed Computing Environment (DCE) provided by the Open Software Foundation (OSF) [98] (Figure 2.5). DCE is still in use today in many popular middleware and enterprise integration products. Because it represents an intermediate stage between basic client server and full 3-tier architectures, we devote some time here to understand what DCE provides.

DCE was the result of an attempt to standardize RPC–an attempt that did not succeed. DCE provides not only a complete specification of how RPC should work but also a standardized implementation. The goal of DCE was to give vendors a standard implementation that they could then use and extend as needed for their own products. By using the same basic implementation of RPC, the hope was that the resulting products would be compatible. Many see this as the reason why DCE did not become an accepted standard. When the Object Management Group proposed CORBA as a standard for distributed object platforms (discussed later in this chapter), it had learned from that error and enforced only a specification, not an implementation. However, since many implementations compliant with the CORBA standard are not compatible among themselves, the OSF may have had a point.

The DCE platform provides RPC and a number of additional services that are very useful when developing distributed information systems. These services include:

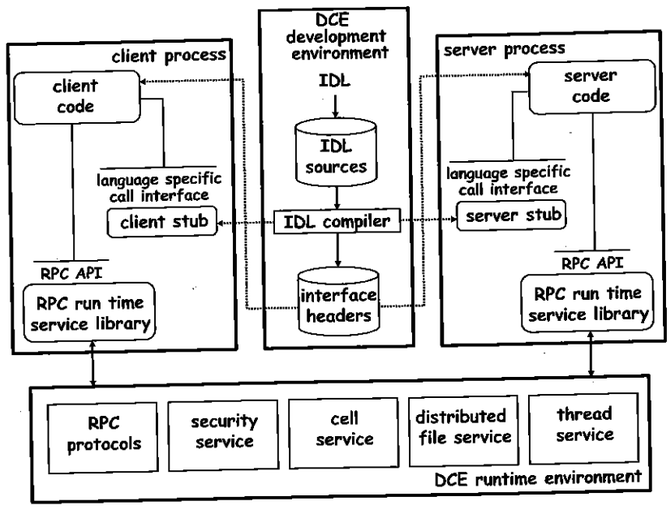

Fig. 2.5. The architecture of DCE

- **Cell directory service.** The Cell directory is a sophisticated name and directory server used to create and manage RPC domains that can coexist over the same network without interfering with each other.
- **Time service.** The Time service provides mechanisms for clock synchronization across all nodes.
- **Thread service.** As pointed out in the explanations above, multi-threading is an important part of RPC systems. DCE supports threads and multiple processors with the Thread service.
- **Distributed file service.** The file service allows programs to share data files across a DCE environment.
- **Security service.** In a shared environment, such as the ones addressed by DCE, it is not feasible to accept procedure calls from arbitrary clients. The Security service solves this problem by providing authenticated and secure communication.

The main architectural characteristics of DCE are shown in Figure 2.5. This figure is a good summary of how RPC works, as well as some of the support needed in real environments. The attentive reader will notice that the figure will appear again under different disguises as we discuss other forms of middleware. This is no accident. Most of the middleware platforms described in this and the next chapters are based on RPC and, in some cases, are direct extensions and enhancements of the RPC mechanism. In fact, many of these platforms are actually built on top of RPC platforms including those

that would appear to use different paradigms (like object brokers or message-oriented middleware).

2.3 TP Monitors

Transaction processing monitors, or *TP monitors*, are one of the oldest forms of middleware. Because they have been around for such a long time, they are very well understood and have been studied in great detail [25, 86, 209]. TP monitor products are also the most efficient form of middleware, given that their architectures and internal mechanisms have been optimized for many years. Nowadays, TP monitors are the cornerstone of many N-tier systems and their architectures and functionality are a constant reference for new forms of middleware.

2.3.1 Historical Background

TP monitors predate client/server and 3-tier architectures. One of the earliest TP monitors was IBM's Customer Information and Control Systems (CICS), developed at the end of the 1960s and still in use today. Initially, such TP monitors were designed to allow mainframes to support the efficient multiplexing of resources among as many concurrent users as possible. As part of this task, they also needed to deal with multithreading and data consistency, thereby extending core functionality with the concept of *transaction*. CICS was the first commercial product offering transaction protected distributed computing [86].

In the same way that the RPC was heavily influenced by the prevailing programming paradigm at the time it emerged, TP monitors owe much of their architecture to the characteristics of early operating systems. In many ways, a TP monitor is an alternative operating system and offers features that compete with those provided by operating systems. At the time, operating systems were rather limited in their capabilities. TP monitors were developed to bypass these limitations. For instance, CICS owed its popularity to the fact that it could create and dispatch a thread an order of magnitude faster than the operating system [86]. CICS, like many other TP monitors that came later, was not only an execution environment with functionality equivalent to that of many operating systems but also a complete development tool. Faithful to the 1-tier architecture, these initial systems were entirely monolithic and ran the complete system as a single operating system process.

With the migration to client/server and 3-tier architectures, TP monitors became more modular but not necessarily simpler. Much of their functionality was separated into components that could be combined in different manners: logging, recovery, persistence, transactional file system, security, communications, and so on. They also became more versatile by offering a wider range of functionality. For instance, CICS was initially a 1-tier system designed for

online transactions. Tuxedo, which appeared at the beginning of the 1980s, was originally a 2-tier queue-based system. Eventually, almost all commercial monitors became 3-tier systems that supported batch transactions, online transactions, and even interactive transactions. In the process, it also became clear that programming support was a critical issue. There was no point in providing sophisticated functionality if designers did not have the necessary programming abstractions to use them in practice. As a result, TP monitors started to incorporate their own languages or versions of programming languages. For example, in the early 1990s, Encina introduced *Transactional-C*, a dialect of C where transactions were first-class programming constructs that made it considerably simpler to program in TP monitor environments.

This trend toward increasing complexity and functionality was reversed to a certain extent with the development of *TP-lite* monitors. The idea was to provide the core functionality of a TP monitor (i.e., transactional RPC) as an additional layer embedded in database management systems. This can be done using *stored procedures*, where application logic can be written and run within the scope of the database rather than in an intermediate layer as in conventional TP monitors. The resulting 2-tier architecture was lighter than a TP monitor, but sufficient for those applications whose only resource managers were databases. TP-lite monitors are an extension of the functionality of a database and lack most of the sophisticated functionality of a conventional TP monitor. As such, they cannot be really seen as integration tools but are, nevertheless, still heavily used–especially in conjunction with stored procedures.

For many decades, TP monitors were the absolutely dominant form of middleware. Indeed, they are one of the most successful forms of middleware, enabling many of the operations that we perform in everyday life (such as purchasing a plane ticket or performing a banking transactions) to take place with satisfactory performance and reliability. In fact, they still play this key role today; TP monitors can be found behind most application servers and Web services implementations. There are many commercial implementations of TP monitors, e.g., IBM CICS [97], Microsoft MTS [140], and BEA Tuxedo [17]. These same TP monitors can be found at the heart of modern product suites for enterprise application integration. TP monitors have also played a pivotal role in the development of middleware platforms. Almost all forms of middleware available today have a direct link to one TP monitor or another.

2.3.2 Transactional RPC and TP Monitors

The main goal of a TP monitor is to support the execution of distributed transactions. To this end, TP monitors implement an abstraction called *transactional RPC* (TRPC).

Conventional RPC was originally designed to enable a client to call a server. When there are more than two entities–and therefore more than one procedure call–involved in an interaction (e.g., a client invoking procedures of

two servers, or a client talking to a server that is talking to a database), then conventional RPC incorrectly treats the calls as independent of each other. However, the calls are not really independent, which complicates recovering from partial system failures. For example, if the call is to a server that, as part of executing the procedure, invokes some other remote procedure, then the semantics of the call are not defined. If the original call returns with an error, the client has no way of knowing where the error occurred and what changes actually took place. There is no way to distinguish between the case where the error is confined to the first server and the second completes its execution of the procedure, and the case where the error is confined to the second server. Similarly, if a client uses conventional RPC to interact with two different servers, it has no way of maintaining consistency between the two servers unless it implements all necessary mechanisms itself.

As an example, consider an application that withdraws money from one bank account and deposits it into another account. Should the client fail between the two calls, the money would be lost unless the client maintained a persistent state. An elegant solution to this problem was to make RPC *transactional*, that is, to extend the RPC protocol with the ability to wrap a series of RPC invocations into a *transaction*.

The notion of transaction was developed in the context of databases, and refers to a set of database operations that are characterized by the so called ACID properties (atomicity, consistency, isolation, and durability). In particular, the transactional abstraction implies that either all the operations in the set are successfully executed or none of them is. If the client interacting with the database fails or if an error occurs after it has executed some (but not all) of the operations within the transaction, the effect of completed operations is rolled back (undone). Databases typically provide transactional guarantees only for operations interacting with the database (Figure 2.6a). As discussed above for RPC, such properties are also desirable when dealing with data distributed across multiple, possibly heterogeneous systems (Figure 2.6b). This is what transactional RPC provides.

The semantics of transactional RPC is such that if a group of procedure invocations within a transaction are committed (i.e., successfully completed), the programmer has the guarantee that all of them have been executed. If instead the group of invocations is aborted, typically due to some failure, then none of them will have been executed (or better said, the overall effect is *as if* none of them were executed). In other words, TRPC guarantees *at most once* semantics on a group of RPC calls. Other similar extensions to RPC have been proposed, where the goal is to achieve *at least once* or *exactly once* semantics to one or groups of RPC calls.

In more detail, procedure calls enclosed within the transactional brackets (*beginning of transaction* (BOT) and *end of transaction* (EOT) instructions) are treated as one unit, and the RPC infrastructure guarantees their atomicity. This is achieved by using a *transaction management* module that coordinates interactions between the clients and the servers (Figure 2.7). This transac-

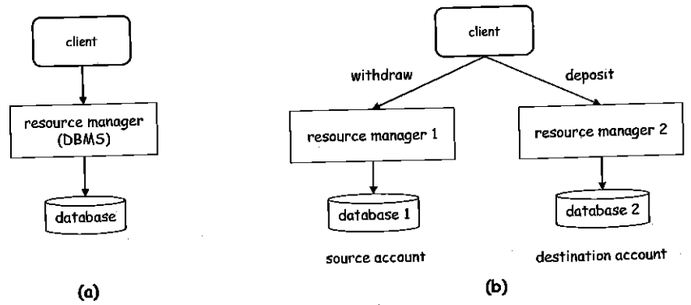

Fig. 2.6. **a** Database management systems (DBMSs) provide ACID properties for the resources they manage; **b** This is not sufficient for procedures that span multiple heterogeneous resources

tion management module is what gives TP monitors their name, but its most basic functionality can be explained as an extension of the RPC mechanism. Thus, the BOT and EOT commands are treated as calls to the corresponding stub (the client stub or the server stub since the server can also make calls to other services). Assume the transactional program resides at the client and that two RPC invocations are being made to two different servers. When the BOT is encountered, the client stub contacts the transaction manager of the TP monitor to obtain the appropriate transactional identifier and create the transactional context that will be used throughout the sequence of calls. From then on, all outgoing RPC calls carry the transactional context with them. When the client calls one of the servers, the server stub extracts the transactional context, notifies the transaction manager that it is now participating in the transaction, and forwards the call as a normal call. The same is done when the client invokes the second server. When the EOT is reached, the client stub notifies the transaction manager. The transaction manager then initiates a *two-phase commit* (2PC) protocol between the two servers involved, to determine the outcome of the transaction. Once the protocol terminates, the transaction manager informs the client stub, which then returns from the EOT to the client code indicating whether the transaction was successfully committed or not. Figure 2.7 summarizes this procedure.

The 2PC protocol [84, 117] used as part of TRPC was first used commercially within CICS, and was subsequently standardized by the Open Group within the X/Open specification [197]. Today, 2PC is the standard mechanism for guaranteeing atomicity in distributed information systems [86, 209]. In 2PC, the transaction manager executes the commit in two phases: in the first phase, it contacts each server involved in the transaction by sending a *prepare to commit* message, asking whether the server is ready to execute a commit. If the server successfully completed the procedure invoked by the TRPC, it

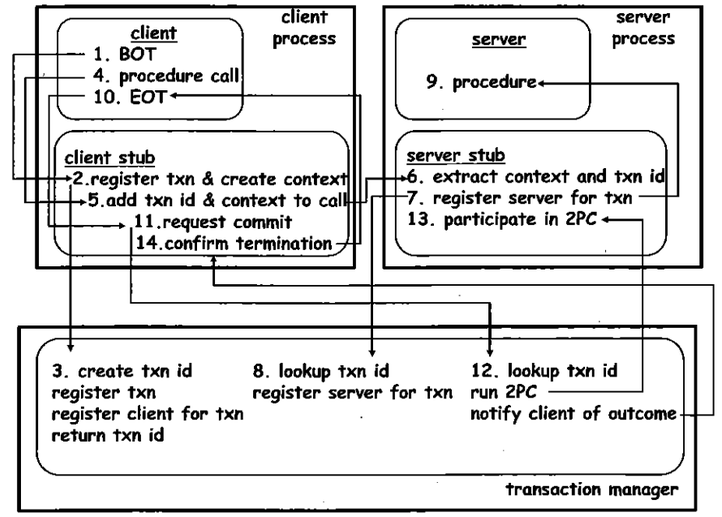

Fig. 2.7. Encapsulating an RPC call within transactional brackets

answers that it is *ready to commit*. By doing this, the server guarantees that it will be able to commit the procedure even if failures occur. If the server could not complete the TRPC, it replies *abort*. In the second phase, the transaction manager examines all the replies obtained and, if all of them are *ready to commit*, it then instructs each server to commit the changes performed as part of the invoked procedure. If at least one resource manager replied *abort* (or failed to reply within a specified time limit), then the transaction manager requests all servers to abort the transaction. Fault tolerance in 2PC is achieved through *logging* (writing the state of the protocol to persistent storage). By consulting such log entries, it is possible to reconstruct the situation before a failure occurred and recover the system. What is logged and when depends on the flavor of 2PC used (*presumed nothing*, *presumed abort*, *presumed commit*, [209]) but it is an important factor in terms of performance, since it must be done for each transaction executed and since logging implies writing to the disk (which is a time-consuming operation). 2PC may block a transaction if the coordinator fails after sending the *prepare to commit* messages but before sending a commit message to all participants. In those cases, the decision on what to do with a blocked transaction is typically left to the system administrator, who can decide to either abort or commit the transaction and then reconcile the state of the system once the coordinator recovers. In practice, for normal transactions, the probability that the coordinator fails exactly at that point in time is very small.

2.3.3 Functionality of a TP Monitor

As already pointed out, TP monitors are nowadays extremely complex systems. Their key features, however, can be explained in terms of what is needed to support TRPC. Condensed into a single sentence, a TP monitor provides the functionality necessary to develop, run, manage, and maintain transactional distributed information systems. This functionality typically includes:

- All the functionality and machinery necessary to support RPC (IDLs, name and directory servers, security and authentication, stub compilers, and so on).
- Programming abstractions for dealing with TRPC. These include RPC, BOT, and EOT, and may also include callback mechanisms and additional support for implementing complex workflow structures (such as *on commit*, *on abort*, and so on).
- A transaction manager for implementing TRPC. The transaction manager includes a wide range of functionality: logging, recovery, locking, and so forth. In modern TP monitors, this functionality is usually implemented in separate components so that there is no need to activate them if they are not needed. Some systems also provide interfaces to developers to access the functionality of these modules so that application can also take advantage of it (e.g., to log additional information about the state of the application).
- A monitor system in charge of scheduling threads, assigning priorities, load balancing, replication, starting and stopping components, and so forth. The monitor system provides performance and flexibility to the TP monitor.
- A run-time environment that acts as the computational background for all the applications that use the TP monitor. This run-time environment provides the resources and services applications may need (transactional services, security services, transactional file system, and so on.)
- Specialized components tailored for particular scenarios or systems. These components range from proprietary protocols for interacting with mainframe-based systems to persistent queuing systems for asynchronous interaction.
- A wide variety of tools for installing, managing, and monitoring the performance of all these components.

In addition, each of these elements is likely to come in different flavors and with different interfaces so that it can be used on top of a variety of platforms and in connection with a variety of legacy systems. Like RPC, the programming abstractions available can be used at different levels of complexity. In their simplest form, they merely transform RPC into TRPC. In their most complex form, they allow designers to control almost all aspects of the interaction.

2.3.4 Architecture of a TP Monitor

The typical architecture of a TP monitor is shown in Figure 2.8. The figure abstracts the most salient features of an actual TP monitor and ignores many details related to the way components are interconnected and interact with each other. Components in this architecture include:

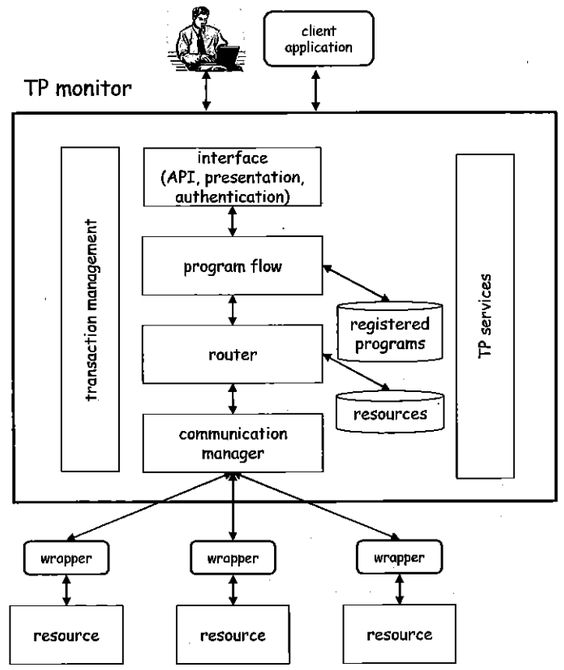

Fig. 2.8. Basic components of a TP monitor

- **Interface.** A client *interface* component provides a programmatic API as well as support for direct access via terminal and authentication.
- **Program flow.** The *program flow* component stores, loads, and executes procedures, possibly written in a language provided by the TP monitor itself. Stored programs typically involve the invocation of operations on logical resources, identified by their name.
- **Router.** The *router* maps each operation to an invocation. The invocation can involve an underlying resource manager (such as a database) or

a local service provided by the TP monitor itself. The router includes a special-purpose database that stores the definition of the mappings between logical resource names and physical devices. In case of changes to the system configuration, the system administrator needs only update this mapping; client applications do not need to be modified, because they access resources based on logical names.

- **Communication manager.** Communication with the resources, such as databases, occurs through a *communication manager* module. This component can be a messaging system, possibly endowed with transactional semantics to enable rollbacks and guaranteed delivery.
- **Wrappers.** *Wrappers* hide the heterogeneity of the different resources linked by the TP monitor. This simplifies the development of the communication module, as it does not depend upon the characteristics of the individual resources.
- **Transaction manager.** Executions of distributed transactions go through a *transaction manager* that executes the 2PC protocol, thereby guaranteeing the ACIDity of the procedures executed through the TP monitor.
- **Services.** Finally, a wide range of *services* is available to provide performance, high availability, robustness to failures, replication, and so forth. In particular, performance management is one of the most distinguishing features of a TP monitor.

A TP monitor can typically support the load of hundreds or even thousands of concurrent clients, a load that is unmanageable for most 1-tier and 2-tier systems. Such capacity is achieved by using threads rather than processes, a feature supported by even the earliest TP monitors. Processes are heavyweight entities. The overall performance of a system is typically very sensitive to the number of active processes: they consume memory, require context switching (thereby inducing page misses in caches and virtual memory), and increase the execution time of operating system functions that have to scan the list of all running processes sequentially. Instead of using processes, TP monitors use threads. A thread is a lightweight entity that shares all the instructions (code) of the program that created it, as well as all the data visible in its scope at the point of creation.

Once created, each thread gets its own thread identifier, as well as its own registers and stack for local variables, which allows the thread to execute independently. Most TP monitors provide their own thread-scheduling mechanism. In addition to threads, TP monitors provide load-balancing components to improve performance by dynamically distributing the workload across different machines. A large part of the architecture of a TP monitor is devoted to the efficient implementation of all these mechanisms.

2.4 Object Brokers

Object brokers extend the RPC paradigm to the object-oriented world and provide a number of services that simplify the development of distributed object-oriented applications. In a nutshell, object brokers are middleware infrastructures that support interoperability among objects. In the following we describe their basic principles, focusing in particular on *CORBA*, the most well-known object broker architecture.

2.4.1 Historical Background

Object brokers appeared at the beginning of the 1990s as the natural evolution of RPC to cope with object orientation, a programming paradigm that was increasingly gaining acceptance. The purposes of object brokers, especially in the beginning, were exactly the same as those of RPC: hide much of the complexity behind remote invocations by making them look like local calls from a programmer's perspective. The difference was that clients did not have to invoke a procedure, but a method of an object. Since object-oriented models include notions such as inheritance and polymorphism, the function performed by the server object actually depends on the class to which the server object belongs, and therefore different objects may respond in different ways to the invocation of the same method. This means that the middleware had to bind clients with specific objects running on a server and manage the interactions between two objects. This was indeed the main functionality of an object broker. With time, object brokers added features that went beyond basic interoperability, for example including location transparency, sophisticated dynamic binding techniques, object lifecycle management, and persistence.

Probably the best-known example of object broker is the abstraction described in the Common Object Request Broker Architecture (CORBA) specification. CORBA is an architecture and a specification for the creation and management of object-oriented applications that are distributed in a network. It was developed in the early 1990s by the Object Management Group (OMG), a consortium of more than 800 companies. CORBA [153] offers a standardized *specification* of an object broker rather than a concrete implementation, and is agnostic with respect to both the programming language used to develop object-oriented applications and also the operating systems on which the applications run.

CORBA enjoyed tremendous popularity in the mid- and late-1990s: many software vendors have implemented the CORBA specifications, and thousands of applications have been developed on top of this architecture. Indeed, it is almost impossible to speak about object brokers without speaking of CORBA, whose well-defined architecture is also suited to introducing the principles and functionality of an object broker. Object brokers and CORBA are so tightly coupled in the minds of many people in the IT world that some technical dictionaries even go as far as defining object brokers as "a component in the

CORBA programming model" [110]. However, not all object brokers were or are based on CORBA. The most significant and widely used example of non-CORBA object broker are the Distributed Component Object Model (DCOM) and its descendant, COM+ [164], which are specific to the Microsoft operating systems (indeed, they are incorporated into it).

More recently, the interest in CORBA has started to fade as new technologies have emerged. The most relevant of these are .NET from Microsoft and the Java 2 Enterprise Edition (J2EE) platforms and application servers offered by a number of other vendors. The OMG is currently responding to these challenges by enabling interoperability between CORBA and Java [156] and by defining a richer, J2EE-like component model for CORBA objects [158]. Nevertheless, CORBA has always been a hybrid standard that covered not only object-orientation but also many services already offered by other middleware platforms. It is thus very likely that CORBA specifications will eventually be subsumed by other standards.

2.4.2 CORBA: System Architecture

A CORBA-compliant system is made of three main pieces (as shown in Figure 2.9):

- **Object request broker.** The object request broker (ORB) provides basic object interoperability functions.
- **CORBA services.** A set of services, known collectively under the name of *CORBA services* [152], is accessible through a standardized API and provide functionality commonly needed by most objects, such as persistence, lifecycle management , and security.
- **CORBA facilities.** A set of facilities, collectively known under the name of *CORBA facilities* [151], provides higher-level services needed by applications rather than by individual objects. Examples include document management, internationalization, and support for mobile agents. CORBA facilities may also include services specific to a vertical market, such as education, health care, or transportation.

Together, these three components support the development of distributed object applications. The ORB is at the core of this architecture: every interaction between clients and services (user-defined, CORBA services, or CORBA facilities) goes through it.

2.4.3 How CORBA Works

In order to be accessed through an ORB, an object must first declare its interface, so that clients are aware of the methods it provides. How this is done is very similar to the mechanisms already described for RPC. Interfaces are specified in CORBA's IDL . Figure 2.10 shows an example IDL specification

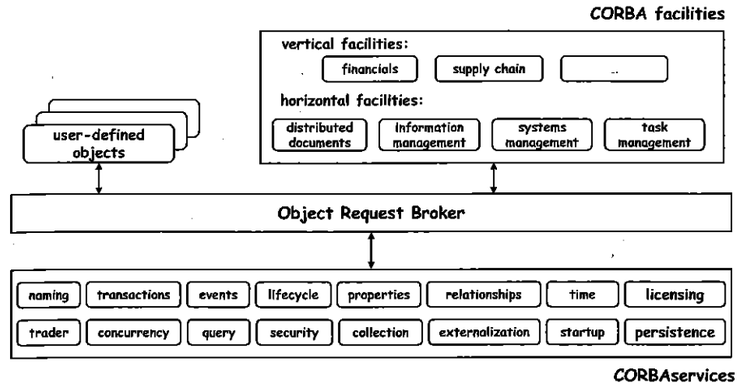

Fig. 2.9. High level view of the CORBA architecture

defining the interface of an object that provides supply chain-related operations (note the similarity to Figure 2.5 in terms of use of stubs, IDL compilers, etc.; except for the different notation, the concepts are practically identical). In addition to method declarations, and unlike the IDLs for RPC systems, CORBA's IDL supports many object-oriented concepts such as inheritance and polymorphism. As with RPC, IDL specifications can be fed to an IDL compiler that generates a *stub* and a *skeleton*. The stub is a proxy object that hides the distribution and makes method calls in the client look as if they were local instead of remote. The stub code includes the declarations of the methods provided by the object implementation and is linked with the client code to obtain the executable client application. The skeleton, on the other hand, shields the server object from all issues concerning the distribution, so that it can be developed as though function calls were coming from a local object (Figure 2.10).

Technically, to develop a client object that interacts with a given server, all a programmer needs to know is the server's IDL interface. Of course, the developer must be aware of the semantics of the interface methods as well as of other constraints (such as a specific ordering in which the methods should be invoked to achieve a certain goal). These aspects are not formalized and are assumed to be described by other means, such as through comments in the IDL specifications or by exchanging descriptive documents.

2.4.4 CORBA: Dynamic Service Selection and Invocation

The interoperability mechanism described above requires clients to be statically bound to an interface. In fact, the IDL compiler statically generates a stub that is specific to a given service interface. In addition to this form of

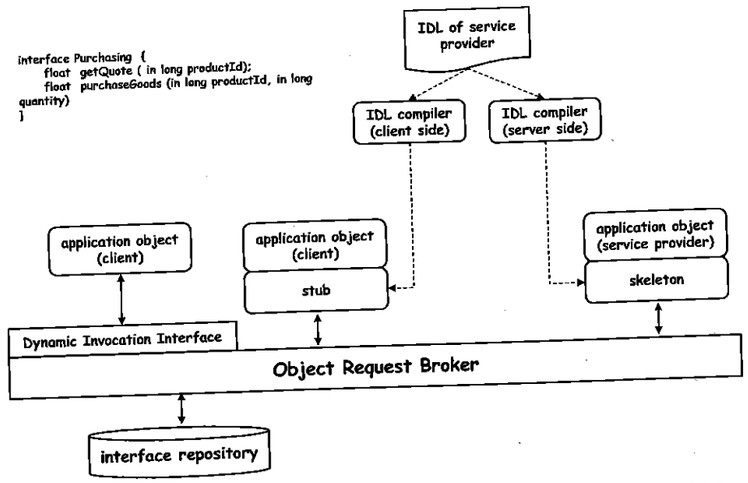

Fig. 2.10. IDL specifications are compiled into skeletons on the server side and into stubs on the client side

interface binding, CORBA allows client applications to dynamically discover new objects, retrieve their interfaces, and construct invocations of this object on the fly, even if no stub has been previously generated and linked to the client. This capability is based on two components: the *interface repository* and the *dynamic invocation interface*. The interface repository stores IDL definitions for all the objects known to the ORB. Applications can access the repository to browse, edit, or delete IDL interfaces. An ORB can have many interface repositories, and the same repository can be shared across ORBs; however, the OMG specifications require each ORB to have at least one interface repository. The dynamic invocation interface provides operations such as *get_interface* and *create_request* that can be used by clients to browse the repository and dynamically construct the method invocation based on the newly discovered interface.

The ability to dynamically construct a method invocation based on a dynamically discovered interface solves only part of the dynamic service invocation problem. It assumes that clients have already identified the service they need. How can they do that? In CORBA, references to service objects are obtained through the Naming and Trader services. The relationship between the Naming and the Trader services is analogous to the one between telephone white and yellow pages. The Naming service allows for the retrieval of object references based on the name of the service we need (e.g., *PurchaseBooks*). The Trader service, on the other hand, allows clients to search for services based on their *properties*. Services can in fact advertise their properties with

the trader. Different services can have different properties, describing non-functional characteristics of the service. With the Trader, clients are not only able to look for objects implementing a certain interface (e.g., the Purchasing interface described above), but also for objects whose properties have specified values (e.g., companies selling history or archaeology books).

Although the capability of using CORBA to perform dynamic service selection and invocation is very intriguing, in practice it is rarely used. Constructing dynamic invocations is in fact very difficult, not so much from a technical perspective (it just requires a few more steps with respect to using statically generated stubs), but from a semantic one. One problem is that, to search for services, the client object must understand the meaning of the service properties, which in turn requires a shared ontology among clients and service providers. Furthermore, if the client has not been specifically implemented to interact with a certain service, it is difficult that it is able to figure out what the operations of the newly discovered service do, what is the exact meaning of their parameters, and in what order they should be invoked to obtain the desired functionality. We revisit these issues when discussing service discovery in Chapter 6, to compare CORBA-based service selection and invocation with the techniques adopted in the Web services domain.

2.4.5 CORBA: Encapsulation

One of the most significant features of CORBA, and of object brokers in general, is *encapsulation*. The notion of encapsulation refers to the hiding of the internal details of an object from its clients. It is a very important abstraction since it allows a service provider to change the details of the service implementations without requiring changes to the client. Encapsulation was present to a certain extent in RPC and TP monitors since they also work based on well-defined interfaces and use IDLs in a manner similar to CORBA. Yet, encapsulation fits more naturally with the object-oriented model used in CORBA.

In Chapter 1, we saw that APIs are the basic ingredient of encapsulation, but there are many other forms in which encapsulation can manifest itself. An example is CORBA's independence from programming languages and operating systems. In CORBA, and unlike in RPC systems and most TP monitors, client and server objects do not need to be implemented in the same programming language and do not need to run on top of the same operating system. Moreover, the client does not even need to know in which language the server object has been implemented or on which environment the server object is running, and vice versa. As we pointed out earlier, all a client needs to know about a server is the latter's IDL specification. Since all method calls flow through the ORB, invocation parameters are converted into a common data representation independent of the specific programming language and operating system, and are converted again by the skeleton prior to invoking

the programming language-specific methods on the server side. Therefore, developers on both the client and the server side are free to change not only the implementation logic, but also the language in which clients and servers are implemented and the operating system on top of which they run, as long as the IDL stays the same.

One could argue that this is in principle also possible with RPC and TP monitors. Note, however, that while the IDL is always the same regardless of which programming language is used for implementing the objects, IDL compilers are not all identical to each other. Each compiler compiles to a different language, and how this compilation takes place plays a crucial role in determining compatibility. For instance, the skeleton must invoke the server object's methods and such an invocation is programming language-specific. The big advantage of CORBA over previous uses of the concept of IDL is that the mappings from IDL to different programming languages are also standardized. This ensures that an object implementation can be ported across ORBs of different vendors independently of the language used. It was this standardization effort that made encapsulation much more effective in CORBA than in RPC systems and TP monitors.

Another form of encapsulation provided by object brokers is location independence. As discussed above, when clients need to invoke a service supporting a certain interface, they access the ORB to retrieve a reference to the server object. An object reference is a logical identifier for a server object, assigned as the object is created. Conceptually, the reference has no relation to the physical address of the object. From the client's perspective, it is just an opaque identifier. It is the job of the ORB to maintain the correspondence between the object reference and the actual object location. The reference remains valid until the server object is destroyed, even if the object changes physical location during its lifetime. In addition, objects not only can change physical address but they can even be running on top of a different ORB, perhaps provided by another ORB vendor. In fact, in addition to providing specifications for interoperability among objects, CORBA also supports interoperability among ORBs [153]. Every CORBA-compliant ORB is required to speak the General Inter-ORB Protocol (GIOP). GIOP can be implemented on top of different transports. CORBA requires that a CORBA-complaint ORB supports at least GIOP over TCP/IP, which is referred to as Internet Inter-ORB Protocol (IIOP). Through this protocol, an ORB can forward invocations initiated by one of its clients to another ORB, where the invoked object resides, as long as the two ORBs are aware of each other's existence.

2.4.6 TP Monitors+Object Brokers = Object Monitors

Object monitors [31] are a good example for understanding the evolution of middleware platforms and the hype around commercial products. Very often, new forms of middleware appear that are just slightly modified versions of already existing products. Object monitors are the best example of this. The very name implies what object monitors really are: an *object* broker and a TP-*monitor*. This marriage between object brokers and TP monitors was an obvious step to take at the time and, in many cases, the only feasible way to obtain commercially competitive products.

Part of the problem encountered by object brokers, particularly in the case of CORBA, is that the only real novelty they offered was object orientation as a way to standardize interfaces across different systems and programming languages. In fact, CORBA was meant to be implemented on top of conventional middleware platforms such as DCE, TP monitors, and even primitive forms of message-oriented middleware. Unfortunately, programming paradigms are a small part of the picture in a distributed information system. Many of the CORBA *services* were mere specifications that took quite a long time to be implemented in real products. The Object Transactional Service (OTS) of CORBA is probably the most illustrative case of the dilemma faced by object brokers. OTS essentially described what TP monitors had been doing very successfully for many years before CORBA appeared. Since the first commercial products available were typically systems implemented almost from scratch, when compared with already existing middleware platforms, object brokers were extremely inefficient and lacked key functionality such as transactions. This turned out to be a decisive factor limiting the adoption of object brokers.

The way out of this dilemma was to actually use TP monitors and other forms of middleware with an additional layer that would make them object-oriented (and, in some cases, CORBA compliant). When TP monitors were used, the result were object monitors. In terms of functionality, however, object monitors offered very little over what TP monitors already offered. This was the first step toward the assimilation of object broker ideas into other forms of middleware. Java, C#, and the middleware environments around them represent the culmination of this convergence process.

2.5 Message-Oriented Middleware

The previous chapters and sections have presented interoperability concepts and techniques that are mainly based on synchronous method invocation, where a client application invokes a method offered by a specific, although possibly dynamically selected, service provider. When the service provider has completed its job, it returns the reply to the client. In this section we explore abstractions supporting more dynamic and asynchronous forms of interaction, along with the corresponding middleware platforms.

2.5.1 Historical Background

Message-oriented middleware is often presented as a revolutionary technology that may change the way distributed information systems are built. The idea

is, however, not new. Originally, asynchronous interaction was used to implement batch systems. RPC implementations already offered asynchronous versions of RPC, and many TP monitors had queuing systems used to implement message-based interaction. For instance, the first versions of the TP monitor Tuxedo were based on queues. Furthermore, the notion of a persistent queue was already well understood at the beginning of the 1990s [27].

Modern message-oriented middleware is, for the most part, a direct descendant of the queuing systems found in TP monitors. In TP monitors, queuing systems were used to implement batch processing systems. But as TP monitors were confronted with the task of integrating a wider range and larger numbers of systems, it quickly became obvious that asynchronous interaction was a more useful way to do this than RPC. As computer clusters started to gain more ground as platforms for distributed information systems, the queuing systems of TP monitors started to play a bigger role when designing large information systems. Eventually they became independent systems on their own. Today, most large integration efforts are done using message-oriented middleware. As discussed in Chapter 6, messages may also become the preferred way of implementing Web services.

Some of the best-known MOM platforms include IBM WebSphere MQ (formerly known as MQ Series) [100], MSMQ by Microsoft [137], or WebMethods Enterprise by WebMethods [213]. CORBA also provides its own messaging service [155].

2.5.2 Message-Based Interoperability

The term *message-based interoperability* refers to an interaction paradigm where clients and service providers communicate by exchanging *messages*. A message is a structured data set, typically characterized by a *type* and a set of <name,value> pairs that constitute the message *parameters*. The type used to be system dependent; nowadays, most products use XML types. As an example, consider a message that requests a quotation from a vendor about the price of a set of products. The message parameters include the name of the requesting company, the item for which a quote is being requested, the quantity needed, and the date on which the items should be delivered at a specified address. In general, the language for defining message types varies with the messaging platform adopted.

```
Message quoteRequest {
 QuoteReferenceNumber: Integer
 Customer: String
 Item: String
 Quantity: Integer
 RequestedDeliveryDate: Timestamp
 DeliveryAddress: String
}
```


To show how message-based interoperability works, we assume that an application receives the customers' requests for a quote and has to transfer them to a quotation system, to retrieve the quote, and notify the requesting customer. This interaction is depicted in Figure 2.11. We refer to the quotation system as being the service provider, while the application receiving the customer's request is the client in this case.

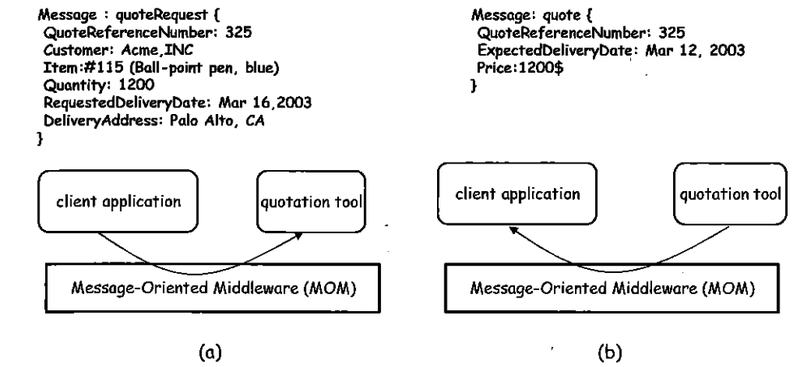

Fig. 2.11. Example of message-based interoperability: an application sends a message to a quotation tool (**a**). The tool serves the request by sending another message (b)

With message-based interoperability, once clients and service providers agree on a set of message types, they can communicate by exchanging messages. To request a service, the client application sends a message (for example, a *quoteRequest* message) to the desired provider. The service provider will receive the message, perform appropriate actions depending on the message content (for example, determine a quote), and send another message with the required information back to the client.

The class of middleware applications that support message-based interoperability is called *message-oriented middleware* (MOM). Note that although we use the terms client and service provider, this distinction is blurred in pure message-oriented interactions, at least from the perspective of the middleware. Indeed, to the MOM, all objects look alike; i.e., they send and receive messages. The difference between "clients" and "service providers" is purely conceptual and can only be determined by humans who are aware of the semantics of the messages and of the message exchange. This is different with respect to the other forms of interaction discussed earlier, where objects acting as clients invoke methods provided by other objects, acting as servers.

2.5.3 Message Queues

MOM, per se, does not provide particular benefits with respect to other forms of interactions presented earlier in the book. However, it forms the basis on which many useful concepts and features can be developed, considerably simplifying the development of interoperable applications and providing support for managing errors or system failures. Among these, one of the most important abstractions is that of *message queuing*.

In a message queuing model, messages sent by MOM clients are placed into a queue, typically identified by a name, and possibly bound to a specific intended recipient. Whenever the recipient is ready to process a new message, it invokes the suitable MOM function to retrieve the first message in the queue (Figure 2.12).

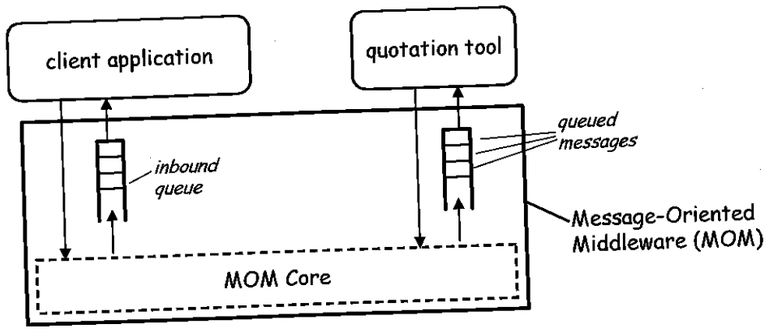

Fig. 2.12. Message queuing model

Queuing messages provide many benefits. In particular, it gives recipients control of when to process messages. Recipients do not have to be continuously listening for messages and process them right away, but can instead retrieve a new message only when they can or need to process it. An important consequence is that queuing is more robust to failures with respect to RPC or object brokers, as recipients do not need to be up and running when the message is sent. If an application is down or unable to receive messages, these will be stored in the application's queue (maintained by the MOM), and they will be delivered once the application is back online and pulls them from the queue. Of course, this also means that the messaging infrastructures must themselves be designed to be very reliable and robust with regard to failures. Queued messages may have an associated expiration date or interval. If the message is not retrieved before the specified date (or before the interval has elapsed), it is discarded.

Queues can be shared among multiple applications, as depicted in Figure 2.13. This approach is typically used when it is necessary to have multiple

applications provide the same service, so as to distribute the load among them and improve performance. The MOM system controls access to the queue, ensuring that a message is delivered to only one application. The queuing abstraction also enables many other features. For example, senders can assign *priorities* to messages, so that once the recipient is ready to process the next message, messages with higher priorities are delivered first.

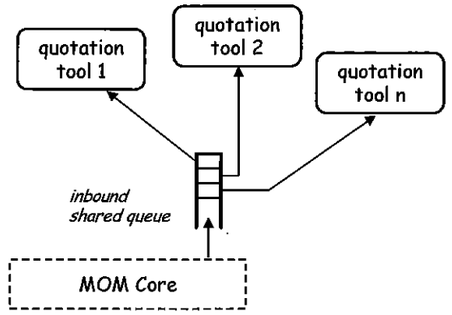

Fig. 2.13. Message queuing model with shared queues

2.5.4 Interacting with a Message Queuing System

Queuing systems provide an API that can be invoked to send messages or to wait for and receive messages. Sending a message is typically a non blocking operation, and therefore once an object has sent a message to another object, it can continue processing. Receiving a message is instead often a blocking operation, where the receiving object "listens" for messages and processes them as they arrive, typically (but not necessarily) by activating a new dedicated thread, while the "main" thread goes back to listen for the next message. Recipients can also retrieve messages in a non blocking fashion by providing a callback function that is invoked by the MOM each time a message arrives. Note therefore that this approach, unlike basic RPC, is naturally asynchronous.

Java programmers can use an industry-standard API for interacting with MOM systems: the Java Message Service (JMS) [194]. In JMS, a message is characterized by a *header*, which includes metadata such as the message type, expiration date, and priority; by an optional set of *properties* that extend the header metadata attributes, for example, to support compatibility with a specific JMS implementation; and a *body*, which includes the actual application-specific information that needs to be exchanged. In JMS, as in most MOM systems, addressing is performed through queues: senders (receivers) first *bind* to a queue, i.e., identify the queue to which they want to

send messages (receive messages from), based on the queue name. Then, they can start sending (retrieving) messages to (from) the queue.

As already indicated, JMS is simply an API and not a platform. In fact, several, but not all, MOM products are JMS compliant. JMS can be implemented as a stand-alone system or as a module within an application server (Chapter 4). For instance, the Java Open Reliable Asynchronous Messaging (JORAM, an open source implementation) [109], is an example of a system that can be used as both stand-alone or as part of an application server. Another open source implementation is *JBossMQ* [108]. There are also several commercial implementations such as FioranoMQ [71].

2.5.5 Transactional Queues

Another very important MOM feature, aimed at providing robustness in the face of errors and failures, is *transactional queuing* (sometimes called reliable messaging). Under the transactional queuing abstraction, the MOM ensures that once a message has been sent, it will be eventually delivered once and only once to the recipient application, even if the MOM system itself goes down between the time the message is notified and the time it is delivered. Messages are saved in a persistent storage and are therefore made available once the MOM system is restarted.

In addition to providing guaranteed delivery, transactional queuing provides support for coping with failures. In fact, recipients can bundle a set of message retrievals and notifications within an *atomic* unit of execution. As described earlier, an atomic unit identifies a set of operations that have the all-or-nothing property: either all of them are successfully executed, or none are. In case a failure occurs before all operations in the atomic unit have been executed, all completed operations are rolled back (undone). In message queuing, rolling back a message retrieval operation corresponds to placing the message back in the queue, so that it can be then consumed again by the same application or, in case of shared queues, by other applications. From the perspective of the sender, transactional queues mean that messages sent within a sender's atomic unit are maintained in a persistent storage by the MOM and are made visible for delivery only when the execution of the atomic unit is completed. Therefore, rolling back message notifications simply involves deleting the messages from the persistent storage.

For example, assume that a set of quotation applications $A = \{a_1, a_2, ..., a_n\}$ retrieves *quoteRequest* messages from a queue, and that the message retrieval operation is within an atomic unit that also involves getting pricing and availability information from the product suppliers, preparing a *quote* message, and sending the message back to the requester. If an application a_j retrieves message $quoteRequest_k$ and is unable to obtain the quote, either because the application a_j itself fails, or because for any reason it is unable to obtain pricing and availability information from suppliers, then message $quoteRequest_k$ is put back into the queue, so that other applications in set A can later retrieve message $quoteRequest_k$ again in an attempt to fulfill the request.

We observe here that although transactional queuing is a very useful mechanism to overcome short-lived failures, additional higher-level abstractions are in general needed to manage exceptional situations: indeed, putting a message back in the queue does guarantee that the message is not discarded and that it will be eventually processed again, enabling service providers to overcome problems that have a short time span. However, if the inability to process the message correctly is due to a cause other than a temporary failure, placing the message back in the queue neither addresses the problem nor provides feedback to the requester about the inability to complete the service. For example, if application a_j is unable to determine a quote because no supplier can deliver the requested goods by the specified delivery date, then rolling back the operations and putting the message back in the queue is not going to help in any way.

2.6 Summary

There are two important aspects to middleware that are sometimes blurred, but that should be differentiated. On the one hand, middleware provides programming abstractions for designing distributed applications. These abstractions can range from more sophisticated communication models (e.g., remote procedure calls instead of sockets), to transactions, queues for asynchronous interactions, or automatic properties conferred to the code (e.g., security or persistence). On the other hand, middleware implements the functionality provided by the programming abstractions. This might include transactional support, specialized communication primitives, name and directory services, persistence, and so on. Here we are no longer discussing the programming abstractions, but rather the mechanisms necessary to implement such programming abstractions. The type of middleware, its capabilities, and how it is used depend very much upon these two aspects.

In practice, however, middleware platforms differ much more in terms of the programming abstractions they provide than in their underlying infrastructure. In this chapter we took an evolutionary perspective when discussing the different forms of middleware available today. We started with RPC and progressed to TP monitors, object brokers, and message-oriented middleware. In this progression it is important to keep in mind the motivation behind each step. RPC provides a very basic form of distribution that initially was devised to avoid having to deal with the low-level communication interfaces available in UNIX (sockets). Soon, the designers of RPC realized that additional infrastructure was necessary to support a higher level of programming abstraction. TP monitors took advantage of RPC to extend to distributed environments the functionality they offered in large computers. By doing so, they became the predominant form of middleware, a role that they still enjoy

today. Changes in programming paradigms and the increasing opportunities for distribution triggered the emergence of object brokers. At the same time, object brokers became the second attempt to standardize middleware platforms (the first was DCE, which tried to standardize the implementation for RPC). For practical reasons, few object brokers made it on their own. Their ideas and specifications were eventually taken over by TP monitors and have been by now almost completely superseded by the .NET and J2EE initiatives. Finally, to cope with the increasing use of cluster-based architectures and the greater demand for enterprise application integration solutions, the queuing systems found in TP monitors became stand-alone systems that soon evolved toward MOM systems and later into what we know today as message brokers. As we will see in Chapter 3, message brokers play a crucial role among the platforms used for enterprise application integration.

3

Enterprise Application Integration

Middleware and enterprise application integration (EAI) are not completely orthogonal concepts. They are, however, distinct enough to warrant separate treatment. As we saw in Chapter 2, middleware constitutes the basic infrastructure behind any distributed information system. Initially, middleware was used to construct new systems and to link to mainframe-based systems (2-tier architectures). Later, it was used to distribute the application logic and to integrate the many servers created by 3-tier architectures.

When the systems involved were compatible and comparable in their functionality and did not involve many platforms, middleware could be used without further ado to integrate the servers. Unfortunately, for more ambitious projects, plain middleware was not enough. The main limitation was that any concrete middleware platform makes implicit assumptions about the nature of the underlying systems. When these systems are very different in nature and functionality, using conventional middleware to integrate them becomes rather cumbersome, and in some cases simply infeasible.

EAI can be seen as a step forward in the evolution of middleware, extending its capabilities to cope with application integration, as opposed to the development of new application logic. Such extensions involve some significant changes in the way the middleware is used, from the programming model to the marked shift toward asynchronous interaction. In this chapter we examine how these extensions came about and how they facilitate large scale integration. In the first part, we discuss in detail the problem of application integration (Section 3.1). Then we address the use of message brokers as the most versatile platform for integration (Section 3.2), and conclude with a review of workflow management systems as the tools used to make the integration logic explicit and more manageable (Section 3.3).

3.1 From Middleware to Application Integration

The difference between conventional middleware and EAI can sometimes be rather subtle. To understand the differences in terms of requirements and actual systems, it helps to differentiate between application development and application integration.

3.1.1 From a Mainframe to a Set of Servers

In Chapter 1, we discussed in detail how client/server systems came to be. As the available bandwidth increased and PCs and workstations became increasingly powerful, the client/server paradigm gained momentum. Functionality that was previously only available in a single location (the mainframe) began to be distributed across a few servers. At the same time, companies became more decentralized and more geographically dispersed and also started to increasingly rely upon computers. As a result, information servers established their presence everywhere within a company. Because of the limitations of client/server architectures, these servers were effectively information islands; clients could communicate with servers, but the information servers did not communicate with each other. At a certain point, this severely limited the ability to develop new services and introduced significant inefficiencies in the overall functioning of the enterprise.

When 3-tier architectures and middleware emerged, they addressed two issues. First, by separating the application logic layer from the resource management layer, the resulting architecture became more flexible. This approach gained even more relevance when systems began to be built on top of computer clusters instead of powerful servers, a trend that was markedly accentuated by the Web. Second, they served as a mechanism for integrating different servers. In this regard, middleware can be seen as the infrastructure supporting the middle tier in 3-tier systems. As such, it is the natural location for the integration logic that brings different servers together. For instance, the accepted way to integrate different databases was to use a TP monitor.

3.1.2 From a Set of Servers to a Multitude of Services

The use of middleware led to a further proliferation of services. In fact, 3-tier architectures facilitate the integration of different resource managers and, in general, the integration of services. The functionality resulting from this integration can be then exposed as yet another service, which can in turn be integrated to form higher-level services. This process can go on ad infinitum, leading to a proliferation of services. The big advantage is that each new layer of services provides a higher level of abstraction that can be used to hide complex application and integration logic. The disadvantage is that now integration is not only integration of resource managers or servers, but also the integration of services. Unfortunately, while for servers there has been a

significant effort to standardize the interfaces of particular types of servers (e.g., databases), the same cannot be said of generic services. As long as the integration of services takes place within a single middleware platform, no significant problems should appear beyond the intrinsic complexity of the system being built. Once the problem became the integration of services provided by different middleware platforms, there was almost no infrastructure available that could help to reduce the heterogeneity and standardize the interfaces as well as the interactions between the systems.

Thus, while 3-tier architectures provided the means to bridge the islands of information created by the proliferation of client/server systems, there was no general way to bridge 3-tier architectures. Enterprise application integration appeared in response to this need. Middleware was originally intended as a way to integrate servers that reside in the resource management layer. EAI is a generalization of this idea that also includes as building blocks the application logic layers of different middleware systems.

3.1.3 An Example of Application Integration

Behind any system integration effort there is a need to automate and streamline procedures. During the late 1980s and the 1990s, enterprises increasingly relied on software applications to support many business functions, ranging from "simple" database applications to sophisticated call center management software or customer relationship management (CRM) applications. The deployment of information systems allowed the automation of the different steps of standard business procedures. The problem of EAI appears when all these different steps are to be combined into a coherent and seamless process.

To understand what this entails, consider the problem of automating a *supply chain*. A supply chain is the set of operations carried out to fulfill a customer's request for products and services. Simplifying and abstracting the procedure, a basic supply chain comprises the following steps: *quotation*, *order processing*, and *order fulfillment*. Quotation involves processing a *request for quotes* (RFQ) from a customer. In an RFQ, the customer queries the company about the price, availability, and expected delivery dates of particular goods. Based on this information, the customer may eventually place a purchase order with the company. Order processing involves the analysis of the purchase order placed by the customer. This includes verifying that the purchase order corresponds to a previously given quote and that it can be fulfilled under the conditions requested by the customer. It also involves placing the order internally to schedule the manufacturing of the goods, purchase the necessary components, and so on. Order fulfillment includes several steps: *procurement*, *shipment*, and *financial aspects*. Procurement is the actual acquisition of components and manufacturing of the requested product. Shipment is the delivery of the product to the customer. The financial aspects include invoicing the customer, paying suppliers, and so forth.

An actual supply chain is much more complex and involves many more steps. We can nevertheless already understand the difficulties of EAI using this basic example. Each one of the steps mentioned is likely to be implemented and supported using a different information system. For instance, companies maintain extensive customer, product, and supplier databases. Responding to an RFQ may involve checking the availability of the product, their production schedule, and even checking with suppliers for delivery dates and prices for the required components. In many cases, each of these databases is a separate system that may even reside in a different geographic location. Processing the purchase order may involve interacting with a warehouse control system that indicates the current stock levels of the requested product and where it can be obtained. As part of the order fulfillment step, the purchase order may be forwarded to a manufacturing system. In this case there might be many additional steps to purchase components from suppliers, arrange for delivery dates, schedule the production and testing, and so on, all of them involving more interactions implemented using different information systems. Finally, shipment and billing also require more or less complex interactions with invoice databases, purchase order archives, and the like. Of the systems involved, some are home-grown, others are based on off-the-shelf packages, and yet others are the result of previous integration efforts. Moreover, each has different characteristics:

- Each system may run on a different operating system (e.g., Windows, Linux, Solaris, HP-UX, or AIX).
- Each system may support different interfaces and functionality. For example, some will be transactional, while others will not understand transactions; some will use a standard IDL to publish interfaces, others would use a proprietary syntax, etc.
- Each system may use a different data format and produce information that cannot be easily cast into parameters of a procedure call (e.g., a complex multimedia document).
- Each system may have different security requirements (for example, some systems may require authentication based on X.509 certificates, while others may need a simple username/password authentication).
- Each system may use a different infrastructure as well as different interaction models and protocols (e.g., a DCE installation, a TP monitor, a CORBA-based system, etc.).

Automating the supply chain implies bringing all these disparate systems together. To make matters worse, EAI is also complicated by non technical challenges: the systems to be integrated are typically owned and operated by different departments within a company. Each department is autonomously managed, and uses its systems to perform a variety of department-specific functions whose needs and goals are not necessarily aligned with those of the integrating application.

In spite of all these difficulties, it is often critical to automate the supply chain. When it is not automated, all the operations that involve going from one step to the next in the chain are carried out manually (Figure 3.1). When that is the case, the whole process involves a lot of repetitive human labor, exchange of paper documents, inefficiencies, and errors. Orders are difficult to monitor and track. It is difficult, if not impossible, to have an overall view of operations and to give information about the status of an order. Any tracking or monitoring can only be done through the cumbersome procedure of following the paper trail left behind by the process. Such inefficiencies strongly affect the quality of the supply chain and severely harm the ability to sustain growth. In the following we describe the middleware technologies that facilitate the integration of such coarse-grained, heterogeneous components.

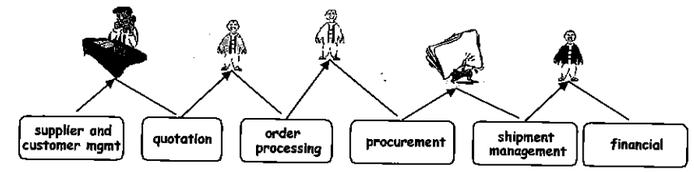

Fig. 3.1. Manual implementation of a supply chain where human users act as relays between the different steps by extracting data from one system, reformatting it, and feeding it into the next

3.2 EAI Middleware: Message Brokers

Traditional RPC-based and MOM systems create point-to-point links between applications, and are thus rather static and inflexible with regard to the selection of the queues to which messages are delivered. Message brokers address this limitation by acting as a broker among system entities, thereby creating a (logical or physical) "hub and spoke" communication infrastructure for integrating applications. Message brokers provide flexibility in routing, as well as other features that support the integration of enterprise applications. This functionality, together with asynchronous messaging, is exactly what is needed in generic EAI settings, and message brokers are thus emerging as the dominant EAI tool used today.

3.2.1 Historical Background

Message brokers are direct descendants of the platforms for message oriented middleware discussed in Chapter 2. They are derived from the new requirements posed by EAI, in terms of supporting the integration of heterogeneous,

coarse-grained enterprise applications such as enterprise resource planning (ERP) and CRM systems. Indeed, as soon as the problems behind EAI (exemplified in the previous section) were recognized, the limitations of using MOM systems to support EAI become manifest. Specifically, MOM did not provide support for defining sophisticated logic for routing messages across different systems and did not help developers to cope with the heterogeneity.

In response to these needs, message brokers extend MOM with the capability of attaching logic to the messages and of processing messages directly at the middleware level. In message-oriented middleware the task of the middleware is to move messages from one point to another with certain guarantees. A message broker not only transports the messages, but is also in charge of routing, filtering, and even processing the messages as they move across the system. In addition, most message brokers provide *adapters* that mask heterogeneity and make it possible to access all systems with the same programming model and data exchange format. The combination of these two factors was seen as key to supporting EAI.

The first examples of modern message brokers were developed in the early and mid-1990s, as soon as the need for EAI was recognized. In the beginning, the area was dominated by startup companies such as ActiveSoftware, founded in 1995 and later acquired by WebMethods. With time, software heavyweights such as IBM entered this profitable area, typically by enhancing existing MOM infrastructure. The final "blessing" came from the Java Message Service, a Java API that provides a standard way to interact with a message broker, as least for the basic message broker functionality [194]. Examples of leading commercial implementations of EAI platforms today are Tibco ActiveEnterprise [199], BEA WebLogic Integration [18], WebMethods Enterprise [213], and WebSphere MQ [100].

3.2.2 The Need for Message Brokers

To better understand the limitations of RPC-based and basic MOM systems, let us again consider the execution of a supply chain operation where a company receives a *purchase order* (PO) from a customer and needs to fulfill it. Many different systems will need to process the PO, including, for instance, inventory management applications (to check availability), ERP systems to manage payments, and the shipping application that interfaces with the shipping department to arrange for delivery of the goods. With RPC and message queuing systems, the application that receives and dispatches the PO needs to get a reference (either statically or dynamically) to these three PO processing applications and either invoke one of their methods or send them a message (Figure 3.2). Now, assume that because of a change in the business or IT environment, company policies require additional applications to be notified of the PO. An example of such an application could be a *month-end closing* system that performs book-keeping operations and provides monthly summaries of revenues and profits. In this case, the dispatching application needs

to be modified to cope with the change and include the code for notifying this additional system.

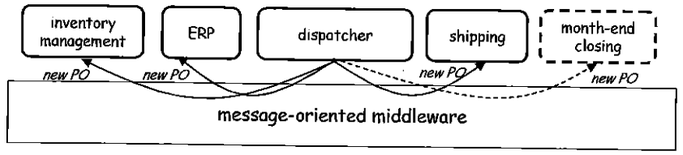

Fig. 3.2. With RPC or message-based interoperability, applications need to be changed if they need to interoperate with a new system (*dashed*)

As another even more "dynamic" example, consider an application that monitors the stock market and interoperates with other applications interested in stock price changes, such as systems that manage investments for stock brokers. In such a scenario, the number of applications interested in stock price modifications changes continuously, as a result of changes in the number of brokers or in the brokers' investing strategies. Therefore, using a basic MOM system to achieve interoperation between the stock monitor and the brokers' applications would be extremely complex, involving the generation of many messages whose number and destination queues are a priori unknown and continuously changing. The problem becomes even more complex if not one but multiple applications can generate the *new PO* or *stock price change* messages, as all these applications then need to be changed.

3.2.3 Extending Basic MOM

The cause of the problem in the above examples is that, in a basic MOM system, the responsibility for defining the receiver of a message lies with the sender. As we have seen, this sort of point-to-point addressing scheme becomes increasingly complex to manage as the number of senders and recipients grows and as the environment becomes more dynamic.

Message brokers are enhanced MOM systems that attempt to overcome this limitation by factoring the message routing logic out of the senders and placing it into the middleware (Figure 3.3). In fact, with a message broker, users can define application logic that identifies, for each message, the queues to which it should be delivered. In this way, senders are not required to specify the intended recipients of a message. Instead, it is up to the message broker to identify the recipients by executing user-defined rules.

The advantage of this approach is that regardless of how many applications can dispatch *new PO* or *stock price change* messages, there is now a single place where we need to make changes when the routing logic for these messages needs to be modified. Later in this section we will see that, using message brokers, there are ways to avoid even this maintenance effort.

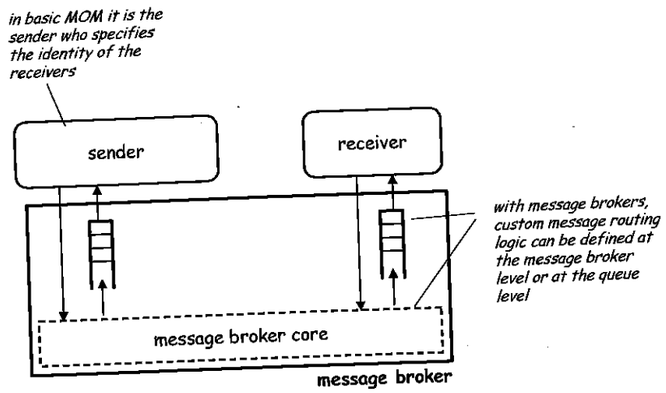

Fig. 3.3. Message brokers enable users to define custom message routing logic

Routing logic can be based on the sender's identity, on the message type, or on the message content. It is typically defined in a rule-based language, where each rule includes a Boolean condition to be evaluated over the message data and an action defining the queues to which messages satisfying the condition should be delivered. This logic can be defined at the message broker level or at the queue level. If defined at the message broker level, it applies to all messages that are then routed accordingly. If it is associated with a specific queue, it defines the kind of messages that the queue is interested in receiving.

The above discussion shows that message brokers can *decouple* senders and receivers. Senders do not specify and are not aware of which applications will receive the messages they send, and, conversely, receivers may or may not be aware of which applications are capable of sending messages to them.

Note that MOM systems that support shared queues also provide a limited form of decoupling, in that applications send messages to queues rather than to specific recipients. However, with shared queues, each message is delivered to at most one application. Applications taking messages from a shared queue are typically of the same type (or different threads of the same process), and shared queues have the purpose of load balancing. Indeed, shared queues may be combined with message brokering to provide both decoupling and load balancing: messages can be delivered to several queues, depending on the routing logic (decoupling). Multiple threads or applications can then share the load of retrieving and processing messages from a queue (load balancing).

Since in MOM systems (and in message brokers) communication between applications goes through a middle layer, it is possible to implement even more application-specific functionality in this layer, going beyond routing rules. For example, another reason for associating logic with queues is to enable the

definition of *content transformation* rules. Refer again to the PO processing example: in that case, routing is only part of the problem. Another issue to be handled is that different applications support different data formats. For example, an application may assume that the weight of goods to be shipped is expressed in pounds, while another may require the weight to be expressed in kilograms. By defining content transformation rules and by associating them with the queue, it is possible to factorize these mappings in the message broker, as opposed to having each application perform them. Every application pulling messages from the queue will now receive the weight expressed in kilograms, as desired. By assigning different transformation rules to each queue, we can accommodate the needs of different applications without modifying these applications.

In general, there is no limit to the amount of application logic that can be "embedded" into the broker or into the queue. However, placing application logic into the broker is not always a good idea. In fact, although on the one hand we make applications more generic and robust to changes, we embed the integration logic into the message brokers' queues. Distributing such logic into the queues makes it difficult to debug and maintain, as no tools are provided to support this effort. Another problem is that of performance: although message brokers tend to be quite efficient, if they have to execute many application-specific rules each time a message is delivered, the overall latency and throughput is degraded. Finally, another limitation of message brokers is their inability to handle large messages. This is perhaps due to the fact that they have been designed to support OLTP-like interactions, which are short-lived and have a light payload. Whenever they are used to route large messages, their performance is greatly affected.

3.2.4 The Publish/Subscribe Interaction Model

Thanks to the possibility of defining application-specific routing logic, message brokers can support a variety of different message-based interaction models. Among those, perhaps the most well-known and widely adopted one is the *publish/subscribe* paradigm. In this paradigm, as in message-based interaction, applications communicate by exchanging messages, again characterized by a type and a set of parameters. However, applications that send messages do not specify the recipients of the message. Instead, they simply *publish* the message to the middleware system that handles the interaction. For this reason, applications that send messages are called *publishers*. If an application is interested in receiving messages of a given type, then it must *subscribe* with the publish/subscribe middleware, thus registering its interest. Whenever a publisher sends a message of a given type, the middleware retrieves the list of all applications that subscribed to messages of that type, and delivers a copy of the message to each of them.

Figure 3.4 exemplifies publish/subscribe interaction in the PO processing example described above. As the figure shows, the PO processing application

simply needs to send a message to the message broker, publishing a notification that a new PO has been received. All applications interested in new PO notifications will then receive the message, provided that they subscribed to the *new PO* message type prior to the time the message was sent.

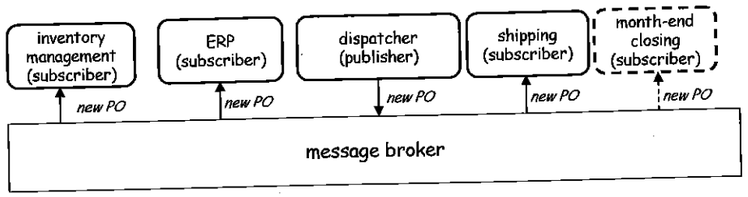

Fig. 3.4. Publish/subscribe models make interoperability more flexible and robust to changes

In a publish/subscribe model, subscribers have two main ways to define the messages they are interested in receiving. The first is to specify a message *type*, such as *new PO*. In simple cases, the type namespace is flat and is defined by a string. More sophisticated systems may allow the types' namespace to be structured into a type/subtype hierarchy of arbitrary depth. For example, if we assume that the type hierarchy is encoded by means of dot-separated strings of the form *type.subtype.subsubtype...*, then a legal message type name could be *Supply Chain.new PO*. With structured types, subscribers not only can register their interest in messages having a specific type and subtype, but can also subscribe to messages whose type T has another type A as ancestor in the type hierarchy. For example, a subscription to *Supply Chain.new PO* specifies interest in receiving all messages related to new purchase orders, while *Supply Chain.*∗ is a more general subscription that declares interest in all supply chain-related messages, including but not limited to new purchase orders.

The second form of subscription is parameter-based: subscribers specify the messages they want to receive by means of a boolean condition on the message parameters. For example, the condition *type="new PO" AND customer="ACME Co." AND quantity>1200* specifies a subscription to all *new PO* messages related to orders by *ACME Co.* whose volume is above 1200 units.

Virtually every message broker today supports the publish/subscribe interaction paradigm. There have even been attempts to standardize the programming abstractions and the interfaces (APIs) between applications and publish/subscribe middleware infrastructure. JMS [194] is one such effort that is part of Java-related standards from Sun. The JMS specifications include a publish/subscribe API in addition to a point-to-point one. In JMS, publications and subscriptions are based on the notion of *topic*. A topic is analogous

to a message broker's queue: it is identified by a string (such as "new PO"), and it is the entity to which clients bind to send and receive messages. The difference between topics and queues is that multiple recipients can subscribe to the same topic and receive the same message.

3.2.5 Distributed Administration of a Message Broker

Message broker systems include support for an *administrator*, a distinguished user that has the authority to define (1) the types of messages that can be sent and received, and (2) which users are authorized to send and/or receive a message and to customize routing logic. Administrators are also present in MOMs, but they are more relevant in message brokers due to the decoupling between senders and receivers that, in general, causes the senders to be unaware of which applications will receive the message. Publish/subscribe systems may, however, allow publishers to define limitations on the set of users that can receive a certain message.

Message broker architectures are naturally extensible to meet the needs of communication-intensive applications that span different administrative domains (possibly corresponding to different departments or companies). In fact, it is possible to compose several message brokers, as shown in Figure 3.5. In this architecture, a message broker (say, message broker MB-A) can be a client of another message broker MB-B, and vice versa. If a client of MB-B wants to receive messages sent by clients of MB-A, then it subscribes with MB-B. In turn, MB-B subscribes for the same message type with MB-A. When one of MB-A clients publishes the message of interest, then MB-A will deliver it to MB-B, in accordance with the subscription. As soon as MB-B receives the message, it will deliver it to all of its clients that subscribed for such a message. Note that from the perspective of MB-B, MB-A is conceptually just like any other subscriber. The only difference is that it belongs to a different administrative domain, and therefore the administrator of broker MB-A will set up subscription and publication permissions for this client accordingly. System administration and security are kept simple and modular with this approach, since each message broker administrator only needs to determine which messages can be sent to or received from another domain.

3.2.6 EAI with a Message Broker

Figure 3.6 shows the basic principles on which most EAI platforms are based. There are two fundamental components:

- **Adapters.** Adapters map heterogeneous data formats, interfaces, and protocols into a common model and format. The purpose of adapters is to hide heterogeneity and present a uniform view of the underlying heterogeneous world. A different adapter is needed for each type of application that needs to be integrated

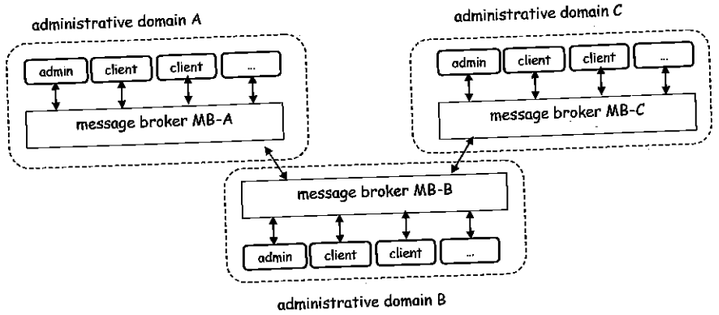

Fig. 3.5. Clients can interoperate through multiple message brokers distributed across different administrative domains

- **Message broker.** A message broker (or a MOM) facilitates the interaction among adapters and therefore, ultimately, among the back-end systems that need to be integrated.

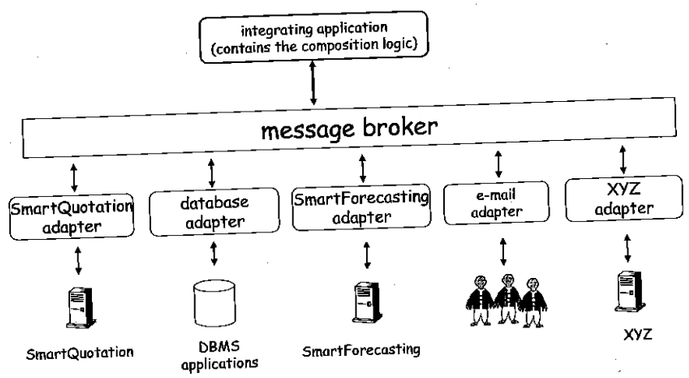

Fig. 3.6. High-level architecture of typical EAI systems

If an EAI platform is in place and the adapters for all the systems to be integrated have been deployed, then EAI-based application integration amounts to:

1. Developing an application (often in Java or C) that implements the integration logic. This application interacts with the message broker (and consequently with the back-end systems accessible through the message broker and the adapters) by publishing and receiving messages.
2. Configuring the adapters so that they subscribe to the appropriate messages and perform the appropriate action on the back-end system.

We next present a simple example based on a quotation scenario that shows how applications can be built on top of EAI platforms. Assume that the application logic for processing requests for quotes involves:

1. Receiving the quote from the customer
2. Accessing the quotation system to obtain a quote
3. Inserting quote information into a forecasting system (i.e., a system that, based on previous quotes, predicts the order volume)
4. Sending the quote back to the customer

Steps 1 and 4 are not discussed here; they involve interactions with the customer and occur either via traditional methods (fax or e-mail), or via the Internet, possibly in the form of interactions among Web services. We focus instead on steps 2 and 3, which involve the integration of enterprise applications.

Figure 3.7 shows the main components of the solution and illustrates the flow of messages among those components. The *RFQ processing* application executes the business logic required for handling requests for quotes, performing the steps described above. In particular, it interacts with the two back-end systems (*SmartQuotation* and *SmartForecasting*) via the EAI platform, by sending messages. It is therefore a client of the EAI message broker, and it is specifically developed for the EAI platform being used. To obtain a quote from the SmartQuotation system, the RFQ processing application publishes a *quoteRequest* message (or, in JMS terminology, sends a message on the *quoteRequest* topic) and expects to receive a *quote* message as reply by the quotation system. To enter a forecast, it publishes a message of type *newQuote*.

The *SmartQuotation Adapter* is an EAI client that enables access to the *SmartQuotation* system via messages. In this example, we assume that the adapter has been configured to:

- Subscribe to messages of type *quoteRequest*.
- Invoke the *getQuote* function provided by the quotation system each time a *quoteRequest* message is received. The parameters required for the function invocation are extracted from the message.
- Publish a *quote* message as the function execution terminates and returns a reply. The message parameters are taken from the return value of the *getQuote()* function.

The adapter to the forecasting system needs to be configured in an analogous way. In particular, it subscribes to *newQuote* messages, and invokes the

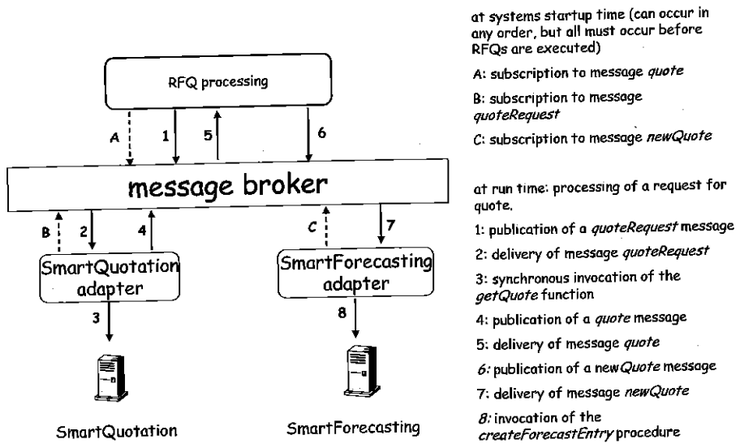

Fig. 3.7. Message exchanges and procedure invocations during the processing of a request for quote

procedure *createForecastEntry* on the *SmartForecasting* system, which will cause the new forecast to be created.

Adapters are typically configured through graphical user interfaces (GUIs) that allow users to define the messages that the adapter can publish or to which it should subscribe, as well as how to map messages to method invocation on the back-end system. If adapters for the back-end systems are not provided "out of the box" by the EAI vendor, then they must be developed. EAI platforms typically include tools that help users in this effort, such as "skeletons" of adapters that can be reused and configured for bridging the message broker with a back-end system.

Once all the adapters have been configured, the process logic can be implemented by writing an RFQ processing application that:

- Subscribes to messages of type *quote*.
- Publishes *quoteRequest* messages as a quote is requested.
- Extracts relevant parameters as a *quote* message is received, and creates and publishes a *newQuote* message.

Observe that, in an alternative approach, the forecasting tool could subscribe to *quote* messages and invoke the *createForecastEntry* when such messages are received, as opposed to having to go through the RFQ processing application. However, "hiding" the process logic in the adapter configurations can make the process difficult to design and maintain, as users would have

to examine each adapter in detail to understand and reconstruct if and how they interact. This problem quickly becomes unmanageable as the number of adapters grows.

3.2.7 A Critical View of Message Brokers as EAI Platforms

Message brokers are not the only way to tackle the problem of enterprise application integration. As discussed in Chapter 2, TP monitors also provide a limited form of application integration functionality. However, message brokers have been in many cases designed explicitly for this purpose, and as such provide more application integration functionalities in terms of the number and type of available adapters, of their customization and extensibility, and of adapter development tools. As a result, integration through a message broker entails a number of benefits that, of course, depend of the concrete scenario and the functionality to be implemented but that are typically characterized as:

- **Lower development cost.** Integration is simpler and can be done more quickly. The integration is simpler because systems are loosely coupled and the biggest part of the integration effort can be localized in the adapters. This is in contrast with the tightly integrated systems resulting from using TP monitors or object brokers.
- **Lower opportunity costs.** Since integration is done more quickly, the automation (and the corresponding cost savings) can be achieved sooner. This directly affects the ability to implement new services and expand the capabilities of the overall infrastructure of the enterprise. It is also possible to react more rapidly to demands for new services, and more services can thus be offered than if alternative solutions (i.e., RPC-based middleware platforms) were used.
- **Lower maintenance effort.** The use of adapters has the effect of extracting the interaction with external systems as an orthogonal concern and localizing the related logic in a single module: the adapter. This offers significant advantages from the software engineering point of view. For instance, if new versions of existing systems are introduced, the EAI vendor will very likely provide new adapters for them. The application that executes the integration logic will thus not have to be changed substantially since the interaction between the integration logic and the underlying applications happens through the adapters.

Despite these advantages, message brokers are not a panacea for all application integration problems. Indeed, there are many drawbacks to implementing EAI solutions using message brokers. The main issue is that software licenses are extremely expensive. An EAI message bus, along with its development and management tools, may cost several hundreds of thousands of dollars, or even millions of dollars. In addition, each adapter costs additional

significant amounts of money. Besides licensing costs, companies need to invest in the training of IT personnel and acquire the resources necessary for the installation and operation of the tool. Finally, while the development of applications that compose existing systems is certainly made easier if an EAI tool is available, it is still a significant effort. In fact, developers must still code the application integration logic, configure the adapters, and even develop adapters for those systems not supported by the message broker-based EAI platform.

These reasons often discourage small and medium enterprises from adopting EAI platforms, and even from performing the integration at all. The situation is different for large companies: first, they are typically faced with very complex integration problems, both because they require the integration of many systems, and because the autonomy of the departments and business units is often a cause of heterogeneity in the choice of the back-end systems. Such problems are quite challenging without automated support. In addition, large companies can afford the investment required by a message broker-based EAI platform, both in terms of software licenses and in terms of operation and maintenance costs, and can leverage these investments across many different integration projects. In the near future, however, the situation may change rapidly, particularly as a result of the adoption of the Web service standards that are discussed in Chapters 6 to 8. Such standards may greatly facilitate the integration efforts and considerably simplify the design and use of the integration middleware.

4

Web Technologies

In Chapter 3, we have studied the need for integrating enterprise applications in order to achieve business process automation. The need to integrate, however, is not limited to the systems within a single company. The same advantages that can be derived from automating a company's business processes can be obtained from automating business processes encompassing several companies. Hence, it should not come as a surprise that there is as much interest in inter-enterprise application integration as there is in intra-enterprise application integration. This chapter serves as a transition between the chapters that precede, which focus on intra-enterprise application integration and on middleware services, and the chapters that follow, which describe Web services for inter-enterprise application integration. The purpose of this chapter is to introduce the basic Web technologies that are used to implement the "Web" portion of Web services.

The Web emerged as a technology for sharing information on the Internet. However, it quickly became the medium for connecting remote clients with applications across the Internet, and more recently (with the advent of Web services) a medium for integrating applications across the Internet. Some of the same technologies that enabled information sharing and integration of remote clients also form the basis for inter-enterprise application integration. That is why, in this chapter, we start by examining the core Web technologies (Section 4.1). We then follow that with a discussion of Web technologies for creating remote clients (Section 4.2). The initial set of technologies for moving the client to remote locations emerged as mechanisms for wrapping local information systems. However, as the popularity of using the Web for building client/server systems increased, traditional middleware platforms were forced to provide the ability to "Web-enable" their applications. In Section 4.3, we discuss one such middleware platform–J2EE-based application servers. Finally, in Section 4.4, we initiate a discussion of some of the Web technologies that were aimed at inter-enterprise application integration. These technologies are precursors to Web services, which are the focus of discussion for the rest of this book.

4.1 Exchanging Information over the Internet

The Internet is a global system of computer networks. In 1969, the Advanced Research Projects Agency (ARPA) connected the computer systems of Stanford Research Institute, UCLA, UC Santa Barbara, and the University of Utah together, across the United States, in a small network called the ARPANET.

ARPANET allowed the connection of autonomous computing systems, which gave rise to the first standards organizations for governing this network. These standards groups developed protocols such as Transmission Control Protocol (TCP), which handles the conversion between messages and streams of packets, and the Internet Protocol (IP), which handles the addressing of packets across networks. TCP/IP is the defining technology of the Internet, because it enables packets to be sent across multiple networks using multiple standards (e.g., Ethernet, FDDI, X.25, etc.).

4.1.1 Before the Web

Two of the earliest standards for exchanging information through the Internet were the telnet protocol [166] and electronic mail's Simple Mail Transfer Protocol (SMTP) [215]. Each of these protocols specifies a different way to directly connect accounts on different systems, regardless of the underlying operating systems and computers involved. The SMTP protocol was eventually extended with Multi-purpose Internet Mail Extensions (MIME), which support the exchange of richer data files, such audio, video, and images data. Soon after telnet and electronic mail emerged, the File Transfer Protocol (FTP) was published (in 1973) [167]. FTP supports file transfers between Internet sites, and allows a system to publish a set of files by hosting an FTP server. One of the innovations of FTP was that although it required authentication, it also permitted anonymous users to transfer files. This meant that users were no longer required to have accounts on all systems that were to be connected. FTP led to the development of the first Web-like distributed information systems, such as Archie [69] (in the late 1980s), which can be thought of as using FTP to create a distributed file system, and Gopher, an early application protocol that provided a simple client/server system and graphical user interface (GUI) for publishing and accessing text files over the Internet [129].

4.1.2 The Web

The core Web technologies as we know them today, i.e., HTTP, HTML, Web servers, and Web browsers, are an evolution of these early technologies. The HyperText Transfer Protocol (HTTP), is a generic, stateless protocol that governs the transfer of files across a network [70]. It was originally developed at the European Laboratory for Particle Physics (CERN). HTTP is generic in that it also supports access to other protocols such as FTP or SMTP. The

same team at CERN that developed HTTP and other related concepts also came up with the name World Wide Web (also known as the Web) [23, 24]. Later, further development was taken over by the World Wide Web Consortium (W3C) with the goal of promoting standards for the Web.

HTTP was designed to support hypertext (the ability to interconnect documents by inserting links between them as part of the document contents). In particular, HTTP supports the HyperText Markup Language (HTML), which defines a standard set of special textual indicators (markups) that specify how a Web page's words and images should be displayed by the Web browser.

Information is exchanged over HTTP in the form of documents, which are identified using Uniform Resource Identifiers (URIs). The exchanged documents can be static (where the resource itself is returned) or dynamic, meaning that the content of the document is generated at access time. Every resource accessible over the Web has a Uniform Resource Locator (URL) that identifies the location of the resource (a file if the resource is a document or a program if the resource is a program that will produce the document) and describes how to access it. Thus, a URL provides the name of a protocol to be used to access the resource, an address (which can be either a domain name or an IP address) of a machine where the resource is located, and a hierarchical description of the location of the resource on that machine (similar to a directory path).

The HTTP mechanism is based on the client/server model, typically using TCP/IP sockets. An HTTP client (e.g., *a Web browser*) opens a connection to an HTTP server (*a Web server*) and sends a request message consisting of a *request method*, URI, and protocol version, followed by a "MIME-like message" (HTTP is not strictly complaint with MIME [70]). The server then returns a response message consisting of a status line (indicating the message's protocol version and a success or error code), followed by a MIME-like message (usually containing the requested document) and closes the connection. Prior to HTTP version 1.1, a separate TCP connection needed to be established to fetch the individual elements of an HTML page–even when they were all located on the same server (meaning that a separate TCP connection had to be established for every access). With version 1.1, HTTP requires servers to support persistent connections, in order to minimize the overhead associated with opening and closing connections.

The request method indicates the actual operation to perform on the server side. Typical methods include *OPTIONS* (send information about the communication options supported by that particular server), *GET* (retrieve whatever document is specified in the request or, if the request specifies a program, retrieve the document produced by the program), *POST* (append or attach the information included in the request to the resource specified in the request), *PUT* (store the information included in the request in the location specified as part of the request), and *DELETE* (delete the resource indicated in the request).

As shown in Figure 4.1, one or more intermediaries, such as a proxy (forwarding agent), gateway (receiving agent), or tunnel (relay point), may lie between the client and the server. RFC 2616 defines a proxy as *an intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients*; a gateway as *a server which acts as an intermediary for some other origin server for the requested resource*; and a tunnel as *an intermediary program which is acting as a blind relay between two connections* [70]. That is to say, a proxy acts on behalf of a client, a gateway acts on behalf of a server, and a tunnel simply connects two networks (e.g., bridges the security mechanisms that might insulate a network, as discussed later in Section 4.4.3). Both the proxy and the gateway potentially process the URLs and content of the messages that they handle; the tunnel does not. Proxies, gateways, and tunnels are all essential concepts that allow HTTP applications to face the integration challenges posed by the Web environment, such as firewalls, load balancing, data translation, etc.

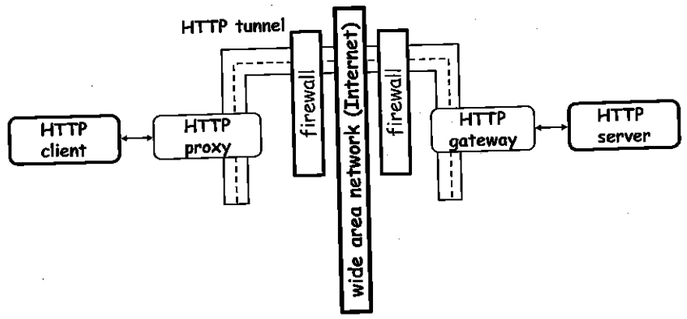

Fig. 4.1. Interaction between a HTTP client and a HTTP server could pass through several intermediaries

4.1.3 Limitations of HTTP

Despite its advantages, HTTP is subject to a number of limitations, many of which have been addressed by corresponding extensions to the HTTP protocol. First, ordinary HTTP does not encrypt data before sending it; if someone were to use a network sniffer to intercept messages between the HTTP server and client, they would be able to "read" those messages. Netscape therefore developed the Secure Sockets Layer (SSL), a protocol that uses public key encryption to protect data transferred over TCP/IP. HyperText Transfer Protocol over Secure Sockets Layer (HTTPS), also known as HTTP over SSL, allows the Web server and client to use SSL to authenticate to each other

and establish an encrypted connection between themselves [148]. As shown in Figure 4.2, HTTPS runs on top of SSL, and requires both the server and the browser to be SSL-enabled.

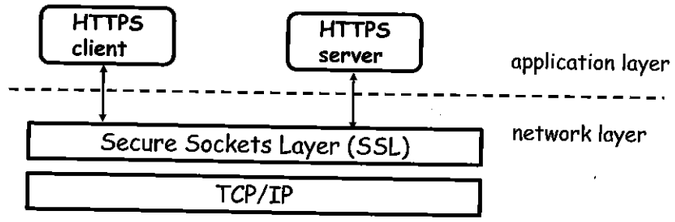

Fig. 4.2. HTTPS servers and clients encrypt the pages that they exchange

Second, HTTP is stateless, meaning that the protocol does not provide any support for storing information across HTTP transactions. Even if they are sent from the same client to the same server, there is no indication of any relationship between two different requests. The application developer is thus responsible for storing and managing the state information. One way for an HTTP server to store information on an HTTP client's machine is to use *HTTP cookies* [149], which are small data structures that a Web server requests the HTTP client to store on the local machine (the actual location of this storage depends on the Web browser being used). The cookies are then used to maintain state information by creating a trace on the local machine of any information needed across different invocations. For example, an e-commerce storefront site might use a cookie to keep track of which items a customer has looked at in a session.

4.2 Web Technologies for Supporting Remote Clients

The original intent of the core Web technologies was to enable linking and sharing documents. But, it was quickly realized that by wrapping local information systems to expose their presentation layer using HTML documents, one could leverage the core Web technologies to have clients that are distributed across the Internet.

4.2.1 Need for Supporting Remote Clients

Conventional 3-tier architectures (and conventional middleware in general) are designed to operate within a single company. This means that the clients of the system will be operated mostly by employees of the company and that data exchanges occur within the safe boundaries of the company. However,

there is in principle no reason why the system could not be opened to other users (for instance, to customers) if the need arises.

The automatic teller machines (ATMs) used in banks are an excellent example of the advantages of doing so. An ATM is basically a PC with a network connection to the information services of the bank [25]. By having ATMs widely available throughout the city, a bank achieves several goals. First, it gives customers easier access to their accounts without the bank incurring the cost of opening, maintaining, and staffing new branches. Second, a significant part of the manual work involved in dealing with customers disappears because instead of having a clerk behind a window, it lets the customer perform the banking operation directly. This practice results in significant savings for the bank and more efficient interactions with customers. Technologically speaking, ATMs are client/server systems. Their practical impact has been very significant.

The extension of 3-tier architectures to enable even further integration possibilities can be analyzed in light of the ATM example. ATMs provide a great service. However, there are limitations for how many ATMs can be installed and maintained in a cost effective manner. No matter how hard a bank tries, customers must travel from their homes or work place to the nearest ATM to complete their banking operations. This would not be necessary if a "personal" ATM machine could be installed at everyone's home and office. That way, customers would have access to their bank accounts, any time they wanted, without having to visit an ATM. Giving the ATM to the customer, so to speak, helps extend its functionality. In addition to depositing or withdrawing money, one could also use the personal ATM to invest in stock, transfer funds to other accounts, make payments, and carry out many other financial operations. Once the user owns the client (rather than sharing it with other users), there are no constraints on the time the customer might take to complete an operation or on the complexity of the information the bank can display. The possibilities are almost endless, to the point that customers may never need to visit the bank in person to perform a financial operation.

The resulting architecture is depicted in Figure 4.3. Such interactions are called *business-to-consumer* (B2C), indicating that the business allows consumers to access their information services directly. Although technically feasible and not overly complicated, implementing this in practice is not as easy as simply running the client on a remote computer. Users wanting to take advantage of the opportunity would need to have a specialized client for every company they wanted to interact with, which is not a practical solution. Moreover, if the middleware-based systems are already quite complex to maintain and administer, this complexity would grow enormously if the clients were not local but rather distributed all over the Internet, many of them running completely outside the control of the middleware platform or its administrators.

One of the biggest contributions of the Web has been precisely to provide a universal client for such extensions. Nowadays, architectures like that in Figure 4.3 are implemented not by moving the client to the remote computer

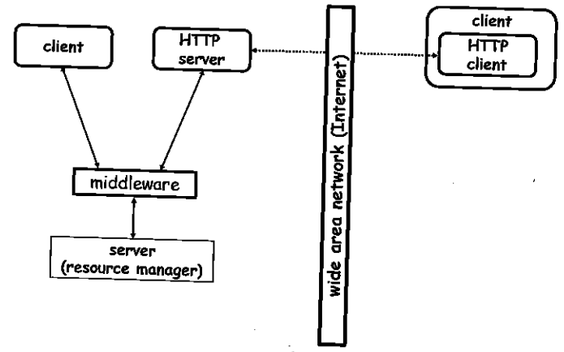

Fig. 4.3. Extending a 3-tier architecture by removing the client to a remote location across the Internet

but by letting the remote computer use a Web browser as a client. Using the Web to implement remote clients effectively places a virtual ATM on any computer with a Web browser. Since Web browsers are standard tools, no application-specific clients need to be installed. Companies can take advantage of the existence of such a standard client by using widely available tools for building a browser-based access channel for their application. In practice this has given rise to new Web technologies for wrapping local information systems to support the new access channel.

4.2.2 Applets

One of the first problems faced when using the Web infrastructure as the medium for implementing remote clients is that Web browsers were originally intended only to display static documents returned by HTTP calls, and it is thus difficult to build sophisticated application-specific clients for Web browsers. One answer to this problem was to introduce *applets*. Applets are Java programs that can be embedded in an HTML document. When the document is downloaded, the program is executed by the Java virtual machine (JVM) present in the browser. Hence, the way to turn the browser into a client is to send the client code as an applet. This, of course, suffers from the limitation of having to download the code every time the client is used. Nevertheless, for applications based on thin clients, it is a very common solution.

Figure 4.3 should thus be understood as the client being implemented as an applet and running on a Web browser, as shown in Figure 4.4.

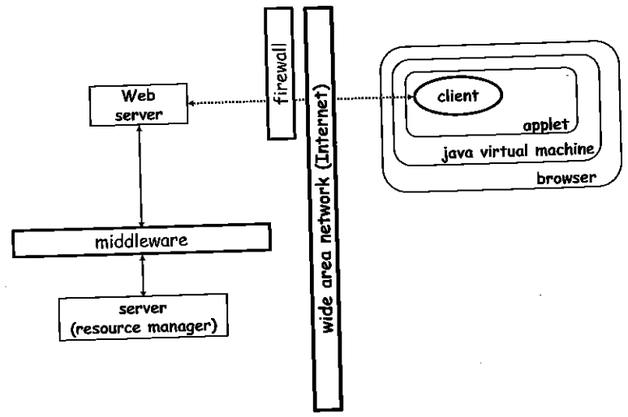

Fig. 4.4. Applets as a way to implement remote clients

Applets have the significant advantage of turning a Web browser into an application-specific client without complex configuration or installation procedures. However, because applets exist only for the lifetime of a particular browser instance, they are transient, and are thus inadequate for supporting complex client code or frequent interactions. An alternative is to use a specialized client that is not based on a browser but contains the necessary code to interact with a Web server through HTTP. Such an approach does not have any of the limitations of applets but requires a specialized client. In this case, the advantages and disadvantages are identical to those of client/server systems, discussed in Chapter 1.

4.2.3 Common Gateway Interface

Thus far, we have only discussed the case where the Web server returns a static document (possibly containing embedded applet code to be executed on the client). But if we view Web servers as interfaces to distributed information systems, they must be able to serve up content from dynamic sources (e.g., publish information retrieved from a database). The issue then is, for example, how a Web server can respond to a request (triggered by addressing a URL) by invoking an application that will automatically generate a document to be returned.

One of the first approaches to solve this problem was to use the Common Gateway Interface (CGI) [147], a standard mechanism that enables HTTP

servers to interface with external applications, which can serve as "gateways" to the local information system. CGI assigns programs to URLs, so that when the URL is invoked the program is executed. Arguments or additional information needed by the program (such as parameters or path information) are sent as part of the invoked URL. CGI programs can be written in a variety of programming languages, and are typically placed in a special directory so that the Web server can identify them as programs (as opposed to static content). Thus, when a request for a URL goes to that directory, the Web server knows it must start a program rather than return an HTML page. From the URL received, the Web server extracts any information it may need to pass on to the program (i.e., parameters for a query or arguments for the program) and starts the program as a separate process[1]. These programs can then be used to interact with the underlying middleware (see Figure 4.5).

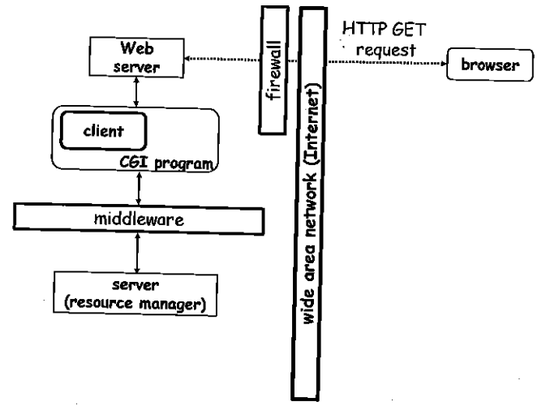

Fig. 4.5. CGI programs as a way to with an application on the server side through a URL

For example, CGI programs often serve as an interface between a database and a Web server, allowing users to submit complex queries over the database through predefined URLs. The parameters of the query are embedded into the URL. When the Web server receives the request for the URL, it will run a program that will act as a client of the database and submit the query.

[1] Fast-CGI reduces some of the overhead associated with the creation of new processes by allowing calls to be made to an already running process rather than creating a new process for every invocation of the URL.

The program executes the query, then packs the query results into a HTML document to be returned to the remote browser.

4.2.4 Servlets

From a performance point of view, CGI programs involve a certain overhead. First, a separate process is usually created for each instance. The creation of the process takes time and requires a context switch in the operating system to pass control to the CGI program. The result is an increase in the overall response time of the system. In addition, multiple requests result in multiple processes being created, all of them competing for resources such as database connections or memory (multiple requests to the same CGI program result in the program being loaded in memory once for every request), thereby limiting the scalability of the system.

To avoid this overhead, Java *servlets* [188] can be used instead of CGI programs. The idea is exactly the same as in CGI programs but the implementation differs (Figure 4.6). The execution of a servlet is triggered in the same way as a CGI script: by addressing a URL. The result is also the same: a document is returned. Unlike CGIs, however, servlets are invoked directly by embedding servlet-specific information within an HTTP request. Servlets run as threads of the Java server process rather than as independent processes. Moreover, they run as part of the Web server. This eliminates the overhead of having to create a process for each invocation and the cost of context switching. It also reduces memory requirements, as a single program image is used even if multiple requests to the same servlet occur simultaneously. Since there is a persistent context for the execution of the servlets (the Java server process), it is possible to use optimizations that would be rather cumbersome to implement when using CGI programs. For instance, results of previous requests can be cached so that identical requests from different clients can be answered without actually having to execute the operation. Session tracking, sharing of database connections, and other typical optimizations that help with scalability are also easier to implement with servlets than with CGI programs. Not discussed in this chapter but closely related to CGI and Servlets are server page technologies such as Active Server Pages (ASP) [134], JavaServer Pages (JSP) [189], and ASP.NET, which embed code to be interpreted on the server side into HTML pages.

4.3 Application Servers

The increasing use of the Web as a channel to access information systems forced middleware platforms to provide support for Web access. This support is typically provided in the form of *application servers*.

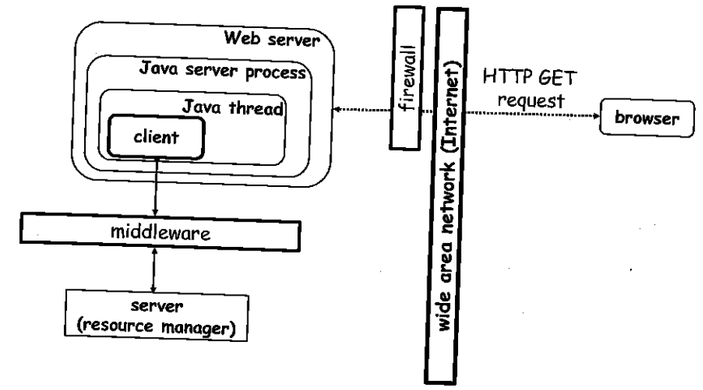

Fig. 4.6. Servlets as a way to interaction with an application on the server side through a URL

4.3.1 Middleware for Web Applications

Application servers are equivalent to the middleware platforms discussed in earlier chapters. The main difference is the incorporation of the Web as a key access channel to the services implemented using the middleware.

Incorporating the Web as an access channel has several important implications. The most significant one is that the presentation layer 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 are thus a big part of what an application server needs in order to extend a middleware platform to one with Web access capabilities. This is typically done by merging the presentation layer related to the Web with the application layer of the middleware platform (Figure 4.7). The reason to integrate the Web presentation layer and the application layer is to allow the efficient delivery of content through the Web as well as to simplify the management of Web applications. Connectivity to the resource management layer is achieved through standard connection architectures and APIs, such as JDBC [190] and ODBC [135] (as it is done in middleware platforms and EAI architectures).

4.3.2 J2EE as the Core of an Application Server

The core functionality of an application server can be described by examining one of the two competing frameworks for Web-based middleware: SUN's J2EE and Microsoft's .NET. In terms of functionality, J2EE- and .NET-based application servers are very similar. Hence, for the purposes of this book, we just

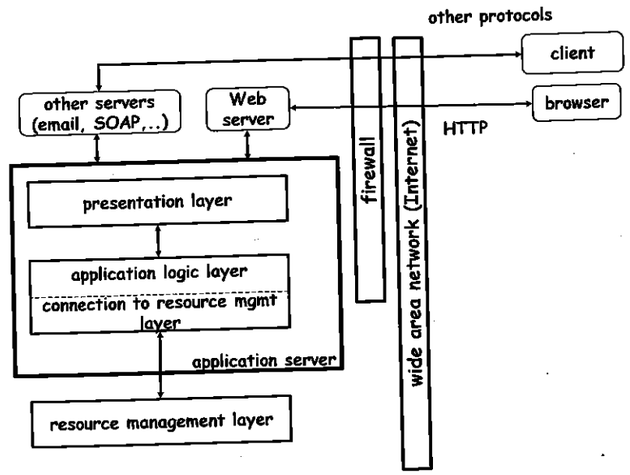

Fig. 4.7. Application servers cover both the application logic and presentation layer

use one of them (J2EE) to illustrate what application servers are all about. The ideas presented should also apply to .NET without major changes.

Figure 4.8 summarizes the main components of the J2EE specification. In addition, J2EE includes other API specifications whose implementation provides functionality commonly needed by application integration projects, such as object brokering and transactional support.

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., WebLogic or WebSphere) has been progressively used to label every middleware component offered by a company. For example, IBM messaging and workflow platforms are now marketed under the name WebSphere MQ, while BEA uses the name WebLogic to label many of its middleware products, such as WebLogic Integration. Sometimes even data warehousing and business intelligence are considered to be part of the application server functionality [161]. Business and marketing reasons therefore contribute to blurring the borders between application servers and other middleware.

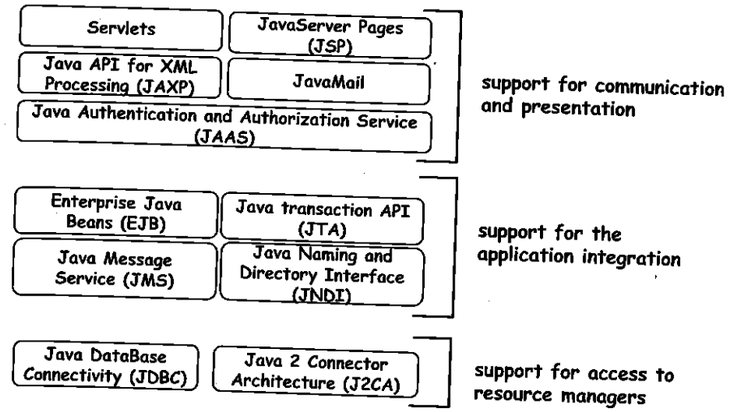

Fig. 4.8. Main API mandated by the J2EE specifications

In the following discussion, we describe the support that application servers provide for the application logic and the presentation layer. Readers interested in more details on application servers than those provided in this chapter can draw from many different resources. For J2EE, the best source of information is the Java Web site [191]. In particular, a short introduction to J2EE is available at [192], while a longer and more detailed version is provided by [29]. Information on commercial application servers is as usual available from the vendors' Web sites. Leading application servers are BEA WebLogic [18], IBM WebSphere [99], Microsoft .NET [165], Sun One [195], and Oracle 9i AS [160]. A more complete list of vendors as well as a comparison of the features provided by J2EE-based application servers is available from [73].

4.3.3 Application Server Support for the Application Layer

At the application layer, application servers conceptually resemble conventional middleware. The functionality provided is similar to that of CORBA, TP monitors, and message brokers. This is why application servers are not limited to Web-based integration, but can also be used for EAI. The goal of application server vendors is in fact to provide a unique environment for hosting all kinds of application logic, whether Web-based or otherwise. As part of this effort, application servers try to provide typical middleware functionality (e.g., transactions, security, persistence) in an automatic manner when the application is deployed in a given server. In this way, application developers do not need to handle this functionality themselves, as these properties are acquired on the fly by deploying a program within an application server.

In J2EE, the support for application logic concentrates on three main specifications: EJB, JNDI, and JMS (see Figure 4.9). We have already discussed the abstractions behind JMS in Chapter 3. We now briefly describe EJB and JNDI.

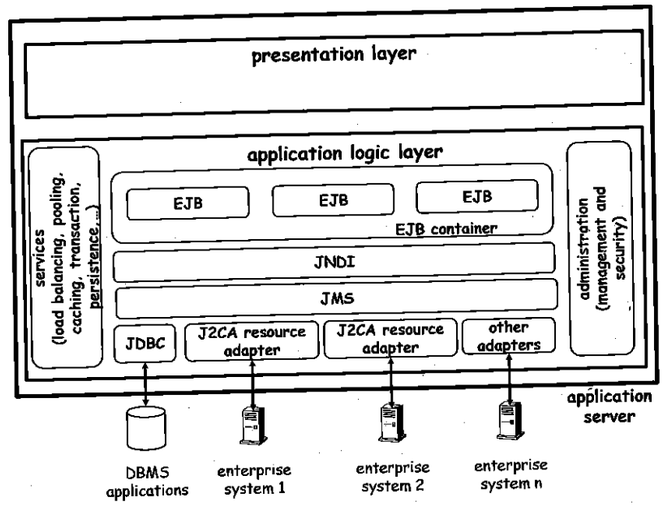

Fig. 4.9. Application servers support the application logic layer

The Enterprise JavaBeans (EJB) specification is at the heart of J2EE, in that the EJB is where the bulk of the application logic resides. An EJB is a server-side component that delivers application-specific functionality such as responding to a request for a quote or processing a purchase order. The EJB specification defines three different types of beans, based on how they interact with other components and on how they manage state and persistence:

- **Session beans** handle a session with a client. They can be *stateful* or *stateless*. Stateful beans maintain the state of a conversation with a client. The typical example of a stateful bean is an online shopping cart. Stateless beans do not maintain any state, and therefore the same stateless bean can be reused for different clients.
- **Entity beans** live beyond the boundaries of a session with the client. They have a state, stored in a database or in another persistent storage. EJB persistence can be managed by the bean itself (meaning that the developer must write SQL or other statements to save the state in a database) or it can be managed by the EJB platform, meaning that developers do not have to write any additional code, and the system will take care of the persistence of objects.
- **Message-driven beans** are the latest addition to the EJB family. They cater to asynchronous interaction with clients, unlike session or entity beans, which instead interoperate in an RPC-like fashion. Asynchronous interaction is achieved through messaging, with message driven beans acting as clients to a JMS message bus.

The *EJB container* provides the environment in which the beans run. All interactions between the EJB and other objects go through the container. Thanks to this mediation role, the container can provide a number of services. For example, it supports transactions, freeing developers from having to define transaction boundaries and implement the related code. Transactions can be managed by the container, according to properties assigned to the EJB methods at deployment time. That is, it is possible to "label" an EJB method according to whether it should or should not be run within transactional boundaries and, if it should be run, whether the method should be executed within the transaction of the EJB client (if any) or whether a new transaction should be created. Other container services include persistence (for entity beans) and security.

Binding to EJBs is done through a Java Naming and Directory Interface (JNDI) directory. JNDI defines an interface for directory services, without mandating any implementation. Using JNDI, clients can bind to servers based on the object name. In the case of EJBs, binding to a server actually involves binding to an object that provides the interface for interacting with a server.

In addition to supporting the development and execution of application logic, J2EE (and an application server in general) addresses the problem of connecting to the resource layer. The approach is once again based on standard APIs and architectures. In particular, J2EE leverages two standards in this space: the above-mentioned JDBC and the J2EE Connector Architecture (J2CA). JDBC is an API that enables developers to access almost any tabular data source by executing SQL commands from a Java program. JDBC methods can be called from an EJB or directly from a servlet, bypassing the access to the application logic layer. J2CA is a generalization of this approach, in that it defines how to build *resource adapters*, i.e., Java components that interface EJB and other Java applications with a resource manager. Each resource adapter is characterized by *contracts* with the application and with the J2EE platform. The application contract essentially defines the API that Java applications can use to access the resource manager. The contract with the platform describes the properties of the resource adapter so that the J2EE platform can support connection pooling, XA transactions, and security.

Besides implementing J2EE (or other specifications in the non-Java world), application servers also offer services that simplify the administration and management of the applications and provide for performance and high availability. For example, they can cache objects that are frequently needed, dis-

tribute the load among pools of objects, or continuously check that an application is running and restart it upon a failure. They also provide object administration and security, defining which user has access to which applications and enforcing access restrictions.

The previous discussion has shown that EJB platforms (and application servers in general) provide many different services that relieve programmers from having to perform a number of tasks. What could previously be done only manually can now be done automatically by the EJB platform. Analogous features are provided (or are being designed) for other distributed object models, such as COM+, or by CORBA (refer, for example, to the discussion on the CORBA Component Model [158]). This is important not only because it speeds up application development, but also because it facilitates their management, thereby reducing the so-called "Total Cost of Ownership." For these reasons, EJBs and other analogous approaches are enjoying a wide popularity today.

Just as compilers needed some time to improve to a point where they could match or even surpass human beings in how they provide their services, EJB platforms are currently improving how they automatically manage persistence, transactions, and other functionality [114]. Today, there is still a trade-off between ease of development and performance. This is true not only for EJB, but also holds for application servers in general. Application servers cannot match, for example, the performance of TP monitors. TP monitors are an excellent platform for supporting high load applications whose characteristics are rather static, as the configuration and deployment phase can be delicate and time consuming. Application servers, on the other hand, try to make systems easier to develop and easier to evolve.

4.3.4 Application Server Support for the Presentation Layer

The support for the presentation layer and for the document as the basic unit of transfer is what differentiates application servers from conventional middleware. CGI programs take a somewhat black box approach in implementing the presentation layer of a Web application, in the sense that they try to link to the middleware platform without requiring changes to it. The resulting architectures have the advantage that they can be used on top of systems that were not prepared to handle wide area integration. For example, legacy systems for which new clients are written can become Web-enabled simply by wrapping that client with a CGI program. In spite of their obvious practical uses, true integration cannot really be achieved unless the middleware platform cooperates. That is, middleware should not be treated as a black box but modified to provide the necessary support to make its services accessible through the Web.

CGI programs offer a crude way to achieve this. Application servers implement similar mechanisms that are slowly evolving toward more sophisticated implementations that make the transition between documents and arguments

more efficient, flexible, and manageable. They provide a variety of presentation features to support the delivery of dynamically generated, personalized content (i.e., documents) to different types of clients. A modern application server supports the following types of clients (as depicted in Figure 4.10):

- **Web browsers**, including both those requesting plain HTML pages and those downloading and executing applets.
- **Applications**, such as those encountered in conventional middleware.
- **Devices**, such as mobile phones or PDAs.
- **E-mail programs.**
- **Web services clients**, i.e., applications that interact with the server through standard Web services protocols.

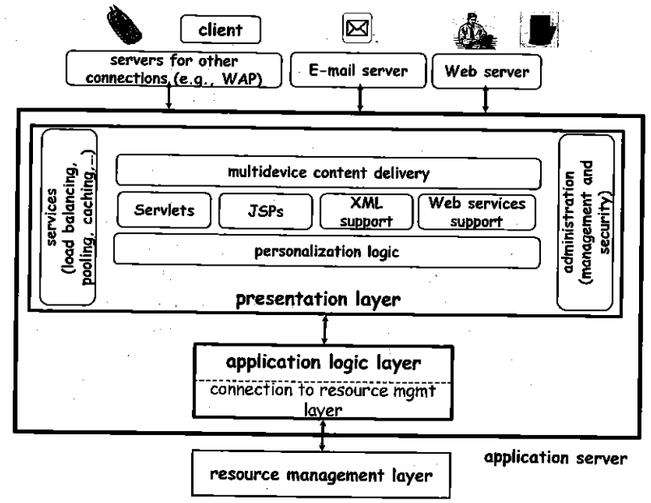

Fig. 4.10. Application server support for the presentation layer

Web browsers are by far the most common type of clients. They interact with the application servers (via the Web server) through HTTP or HTTPS and receive statically or dynamically generated HTML pages. Static pages are in plain HTML while dynamic pages include servlets, JSPs, and ActiveX controls. In both the .NET and the J2EE world, these dynamic components are integrated with the application logic layer. For example, application server development tools simplify the linkage of JSP fragments with the EJBs in the application layer (through the EJB container) providing the data. The JSP

then arranges the data into a format suitable for presentation to the client (typically a Web browser). If the client is an applet rather than a Web browser, the communication with the application server may occur through protocols other than HTTP. For example, applets may interact with an application server through RMI (Remote Method Invocation) or CORBA/ IIOP.

Web browsers are not the only way users can access the information provided by a Web site. Other types of clients exist and are expected to be increasingly used in the near future. Examples of such clients are mobile phones and PDAs. These devices may use presentation languages different from HTML and protocols different from HTTP or IIOP. For instance, mobile phones have adopted the Wireless Application Protocol (WAP) to interact and the Wireless Markup Language (WML) as a presentation language. Application servers aim at transparently supporting different protocols, by wrapping the document into whatever protocol is needed by the client. In this way, developers do not have to write different code for different clients. More advanced tools even enable the dynamic generation of documents in different markup languages and the automatic conversion between those languages.

We must observe, however, that while these characteristics are indeed appealing and may be useful in some practical situations, the requirements of each device type are often so peculiar that they may require an entirely different organization of the presentation logic. For example, preparing content so that it can be consumed by a mobile phone is very different from developing Web pages that will be viewed on a desktop workstation. The problem is not so much in different markup languages or protocols, but rather in organizing the information so that it can be easily displayed using a screen that is typically small and with few colors. Therefore, in many cases a different presentation (and even application) logic must be written for each device.

E-mail programs, just like devices, are characterized by their own protocol (SMTP) and their own text markup format, which is essentially plain text, although it is possible to send email with structured text (HTML) and richer content (attachments). Application servers support the packaging of information and its delivery on top of SMTP.

Applications interact with the application server in much the same way that applets do, and therefore are not discussed here. Technically, the only difference is that the applet's code is downloaded from the server on the fly and, coming from an external entity, is by default not trusted and runs with security constraints.

The fifth type of client is the Web services client. It requires yet another protocol (SOAP) as well as other languages and infrastructures. Application server support for interaction with Web services clients includes facilities for creating, parsing, and validating XML documents, as well as for packaging and unpackaging messages to be delivered through SOAP. This type of client will be discussed in the following chapters.

Besides supporting different types of clients, application servers also aim at supporting different types of users. In fact, one of the features provided by

such systems is *personalization*, i.e., the ability to provide different content and different renderings of the content based on the user to which this content should be delivered. This is often achieved through the definition of a set of condition/action rules: the condition identifies whether the request (and the requestor, if information about the user is available) conforms to a certain profile. For example, the condition can test whether the request comes from a US Internet address or whether the requestor is a "premium" customer. The action part defines the content that should be delivered to requests satisfying the condition, and how this content is structured. For example, it can cause US news to be shown, and at the top of the page rather than at the bottom.

In general, personalization features can be quite sophisticated, going beyond the simple example described above. Indeed, there are many platforms that have exactly this as their main focus and selling point. Examples of Web application development environments with emphasis on personalization include ATG Dynamo [12], WebRatio [214], and Broadvision [37]. Some of these platforms later evolved to provide support for application logic (such as ATG Dynamo), while others focused on Web application development (and run on top of other application servers, e.g., WebRatio). At the same time, vendors that initially focused on the application logic layer are progressively adding personalization features, thereby supporting the architecture depicted in Figure 4.10. Note that Web servers (and the other communication servers) may or may not be included in a vendor's offering. Most application servers can work with many different Web servers, such as Microsoft Internet Information Server, Apache, or Netscape Enterprise Server.

As the figure shows, at this level the application server also provides a number of services that help manage performance, availability, and security, as well as an administration console to manage servlets, JSPs, and other components that populate the presentation layer.