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.

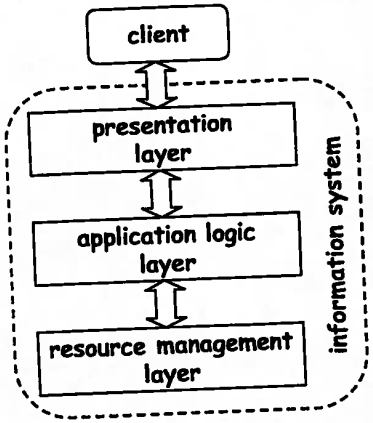

Fig. 1.1. The different layers of an information system

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 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

[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*.

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

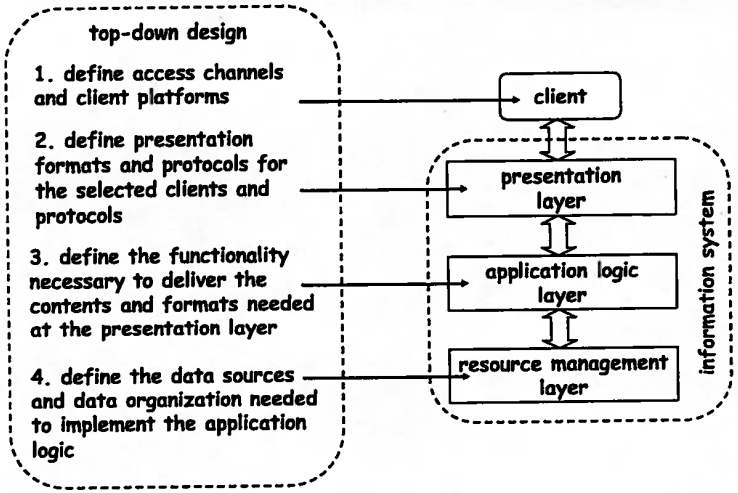

Fig. 1.2. Top-down design of an information system

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

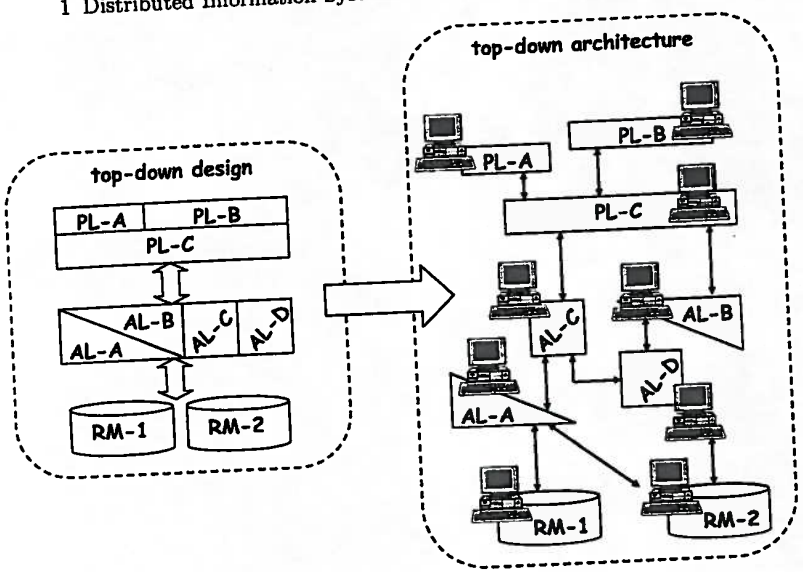

Fig. 1.3. Architecture of a top-down information system. The acronyms PL, AL, RM denote the presentation layer, application logic layer, and resource management layer. PL-A, AL-A, RM-1, and so on indicate distinct modules within each layer

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.

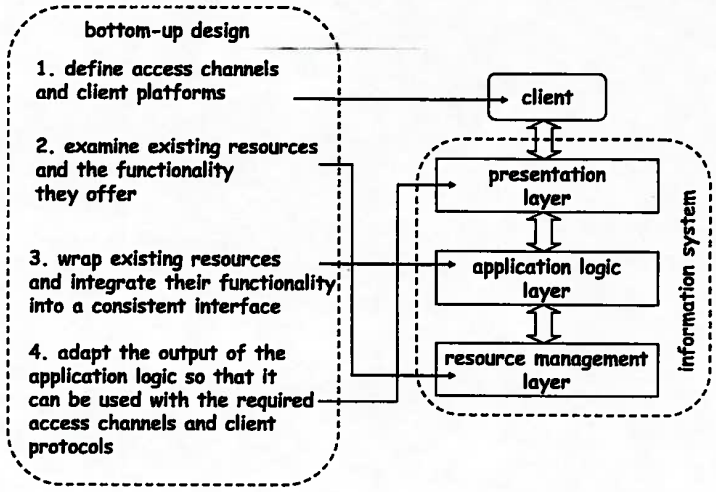

Fig. 1.4. Bottom-up design of an information system

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*.

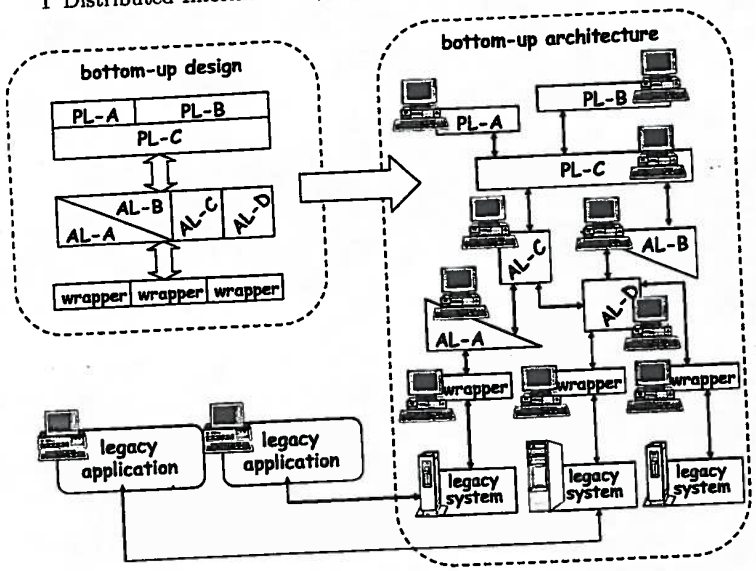

Fig. 1.5. Architecture of a bottom-up information system. The acronyms PL, AL, RM denote the presentation layer, application logic layer, and resource management layer. PL-A, AL-A, RM-1, etc. indicate distinct modules within each layer

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

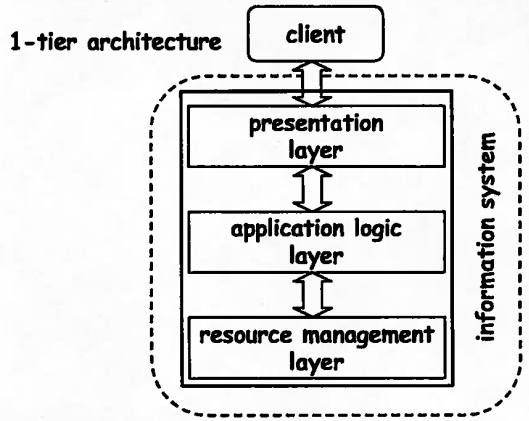

Fig. 1.6. One-tier architectures combine all layers in a single tier. In many early systems, the client in the figure was a dumb terminal

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).

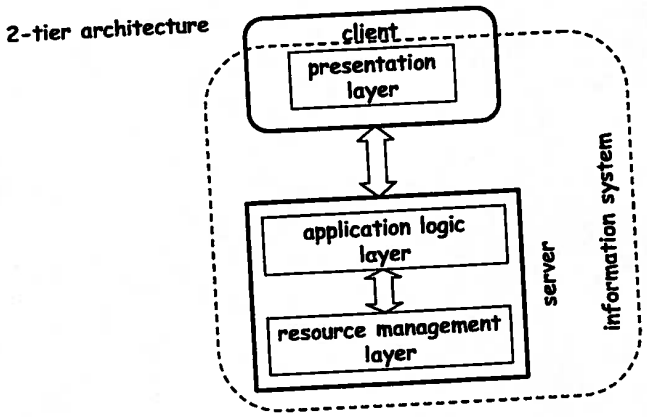

Fig. 1.7. Two-tier architectures separate the presentation layer from the other two layers. The result is a client/server system where the client has the ability to further process the information provided by the server

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.

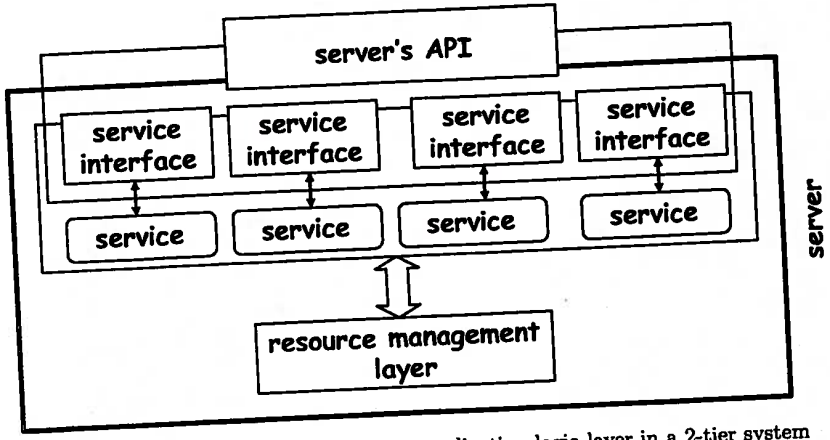

Fig. 1.8. Internal organization of the application logic layer in a 2-tier system

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.

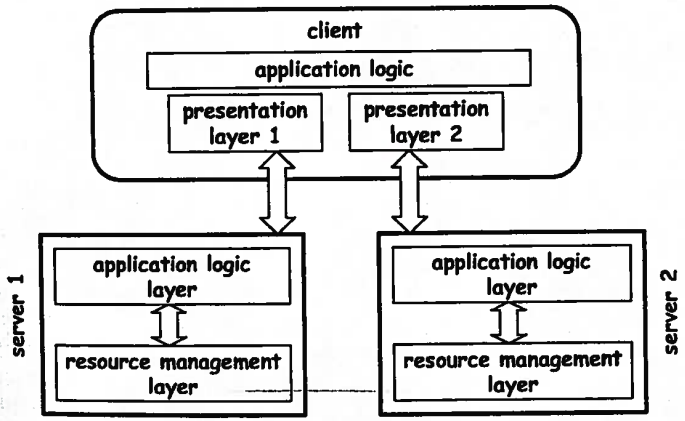

Fig. 1.9. The client is the integration engine in 2-tier architectures

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).

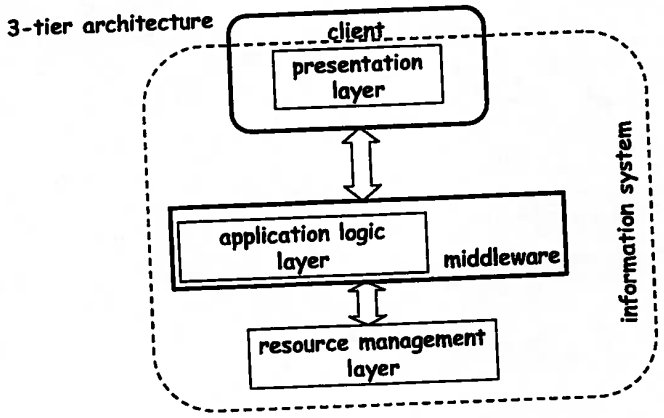

Fig. 1.10. Three-tier architectures introduce a middleware layer between the presentation and the resource management layers

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).

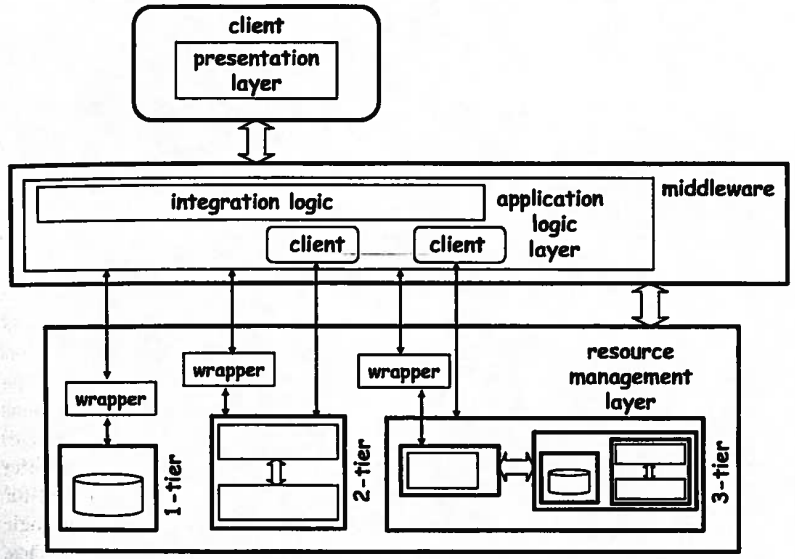

Fig. 1.11. Integration of systems with different architectures using a 3-tier approach

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.

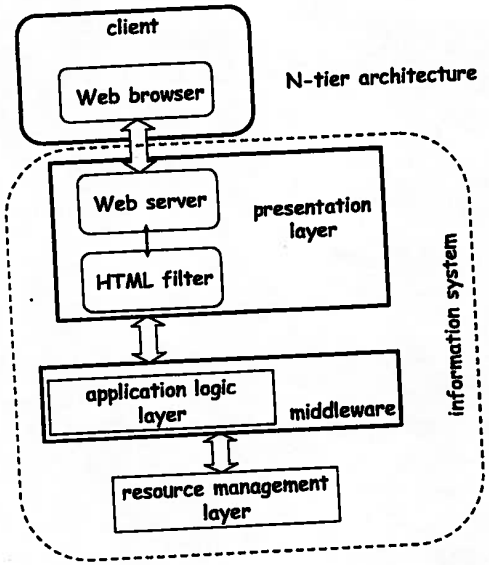

Fig. 1.12. An N-tier system created by extending a 3-tier system by adding a Web server to the presentation layer

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.

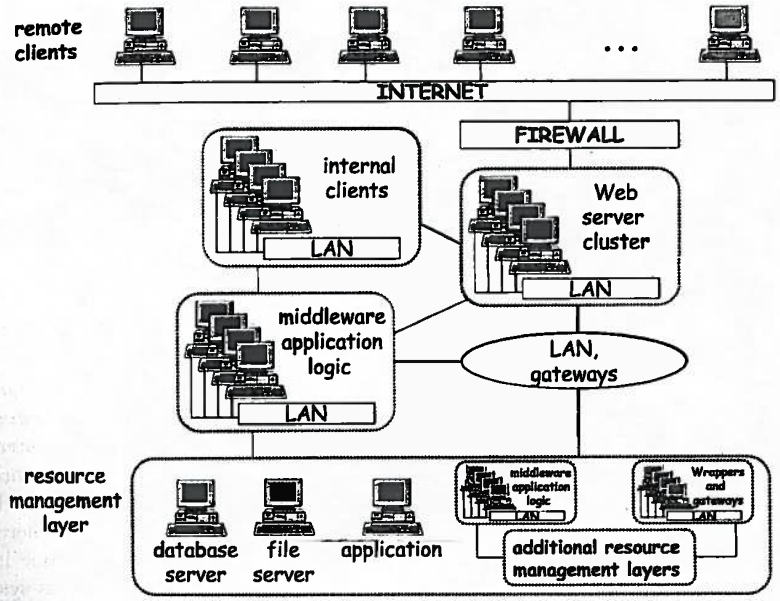

Fig. 1.13. N-tier systems typically encompass a large collection of networks, gateways, individual computers, clusters of computers, and links between systems

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.

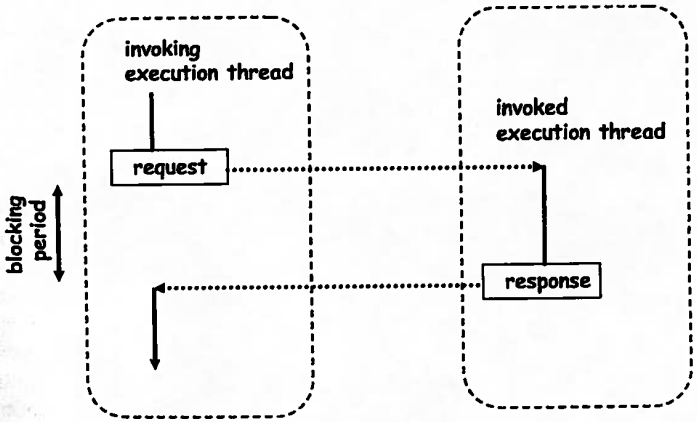

Fig. 1.14. A synchronous call requires the requester to block until the response arrives

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.

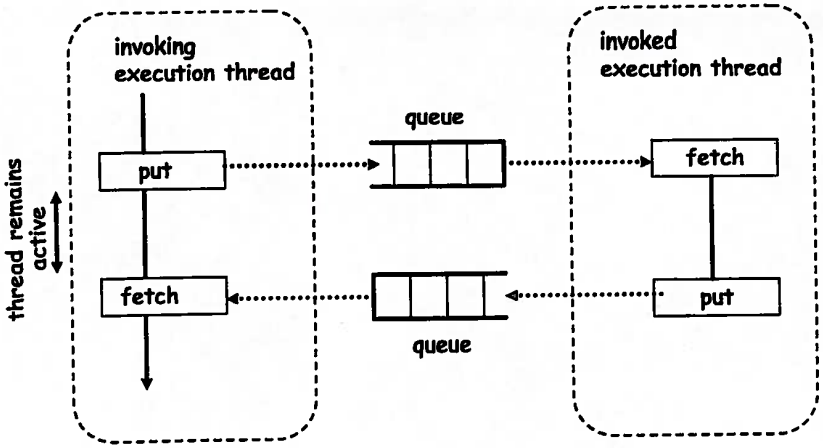

Fig. 1.15. An asynchronous call performed through queues allows the caller to continue working while the request is processed

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.