Chapter 1

An Introduction to Middleware

This chapter is an introduction to middleware. It starts with a motivation for middleware and an analysis of its main functions. It goes on with a description of the main classes of middleware. Then follows a presentation of a simple example, *Remote Procedure Call*, which introduces the main notions related to middleware and leads to a discussion of the main design issues. The chapter concludes with a historical note outlining the evolution of middleware.

1.1 Motivation for Middleware

Making software a *commodity* by developing an industry of reusable components was set as a goal in the early days of software engineering. Evolving access to information and to computing resources into a *utility*, like electric power or telecommunications, was also an early dream of the creators of the Internet. While significant progress has been made towards these goals, their achievement still remains a long term challenge.

On the way to meeting this challenge, designers and developers of distributed software applications are confronted with more concrete problems in their day to day practice. In a series of brief case studies, we exemplify some typical situations. While this presentation is oversimplified for brevity's sake, it tries to convey the essence of the main problems and solutions.

Example 1: reusing legacy software. Companies and organizations are now building enterprise-wide information systems by integrating previously independent applications, together with new developments. This integration process has to deal with *legacy applications*, i.e. applications that have been developed before the advent of current open standards, using proprietary tools, and running in specific environments. A legacy application can only be used through its specific interface, and cannot be modified. In many cases, the cost of rewriting a legacy application would be prohibitive, and the application needs to be integrated "as is".

The principle of the current solutions is to adopt a common, language-independent, standard for interconnecting different applications. This standard specifies interfaces and exchange protocols for communication between applications. These protocols are implemented by a software layer that acts as an exchange bus, also called a broker, between the applications. The method for integrating a legacy application is to develop a *wrapper*, i.e. a piece of software that serves as a bridge between the application's primitive interface and a new interface that conforms to the selected standard.

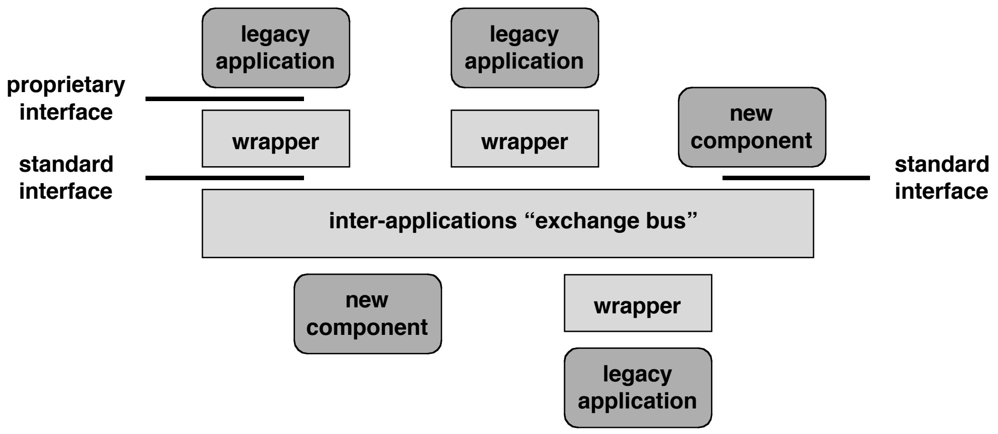

Figure 1.1. Integrating legacy applications

A "wrapped" legacy application may now be integrated with other such applications and with newly developed components, using the standard inter-applications protocols and the inter-applications broker. Examples of such brokers are CORBA, message queues, publish-subscribe systems; they are developed further in this book.

Example 2: mediation systems. An increasing number of systems are composed of a collection of various devices interconnected by a network, where each individual device performs a function that involves both local interaction with the real world and remote interaction with other devices of the system. Examples include computer networks, telecommunication systems, uninterruptible power supply units, decentralized manufacturing units.

Managing such systems involves a number of tasks such as monitoring performance, capturing usage patterns, logging alarms, collecting billing information, executing remote maintenance functions, downloading and installing new services. Performing these tasks involves remote access to the devices, data collection and aggregation, and reaction to critical events. The systems that are in charge of these tasks are called *mediation systems*.

The internal communication infrastructure of a mediation system needs to cater for data collection (data flowing from the devices towards the management components, as well as for requests and commands directed towards the devices. Communication is often triggered by an asynchronous event (e.g. the occurrence of an alarm or the crossing of a threshold by a monitored value).

An appropriate communication system for such situations is a *message bus*, i.e. a common channel to which the different communicating entities are connected (Figure 1.2). Communication is asynchronous and may involve a varying number of participants.

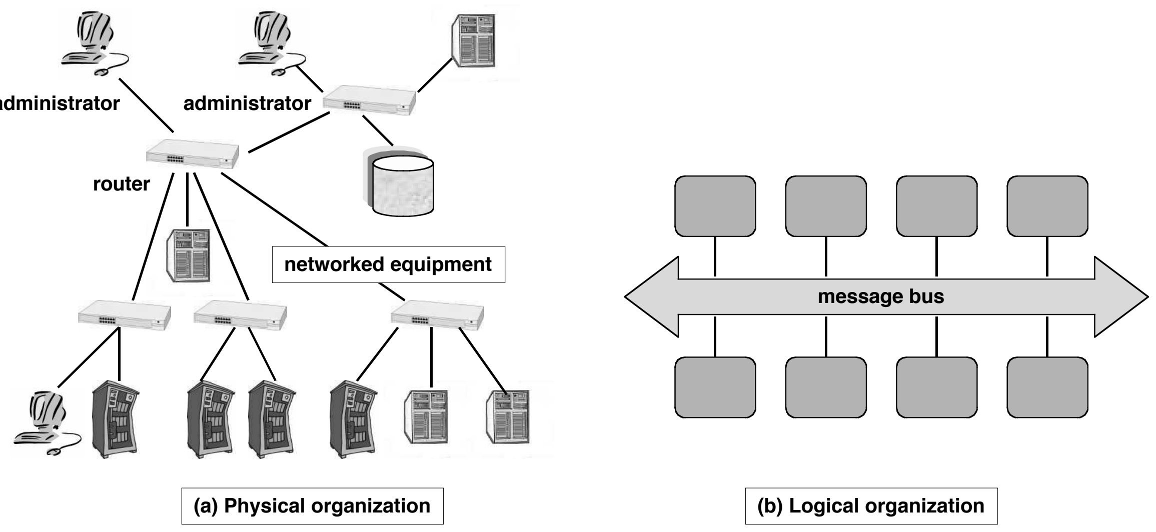

Figure 1.2. Monitoring and controlling networked equipment

Different possibilities exist to determine the recipients of a message, e.g. members of a predefined group or "subscribers" to a specific topic.

Example 3: component-based architectures. Developing applications by composing building blocks has proved much more difficult than initially thought. Current architectures based on software components rely on a separation of functions and on well-defined, standard interfaces. A popular organization of business applications is the so-called "three tier architecture" in which an application is made up of three layers: between the presentation layer devoted to client side user interface, and the database management layer in charge of information management, sits a "business logic" layer that implements the application-specific functionality. This intermediate layer allows the application specific aspects to be developed as a set of "components", i.e. composable, independently deployable units.

This architecture relies on a support infrastructure that provides an environment for components, as well as a set of common services, such as transaction management and security. In addition, an application in a specific domain (e.g. telecommunications, finance, avionics, etc.) may benefit from a set of components developed for that domain.

This organization has the following benefits.

- Allowing the developers to concentrate on application-specific problems, through the provision of common services.

- Improving portability and interoperability by defining standard interfaces; thus a component may be reused on any system supporting the standard interface, and legacy code may be integrated by developing a wrapper that exports the standard interface.

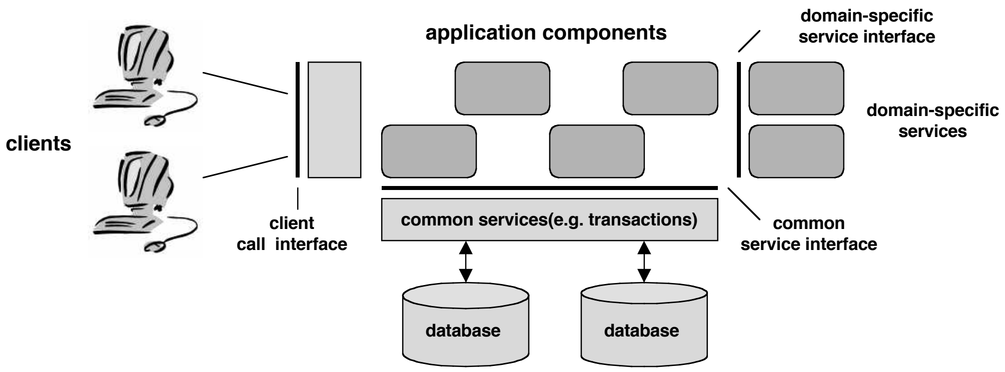

Figure 1.3. An environment for component-based applications

- Improving scalability by separating the application and database management layers, which may be separately upgraded to cater for an increase in load.

Examples of specifications for such environments are Enterprise JavaBeans (EJB) and the CORBA Component Model (CCM), which are developed further in this book.

Example 4: client adaptation through proxies. Users interact with Internet applications through a variety of devices, whose characteristics and performance figures span an increasingly wide range. Between a high performance PC, a smart phone, and a PDA, the variations in communication bandwidth, local processing power, screen capacity, ability to display color pictures, are extremely large. One cannot expect a server to provide a level of service that is adapted to each individual client's access point. On the other hand, imposing a uniform format for all clients would restrict their functionality by bringing it down to the lowest common level (e.g. text only, black and white, etc.).

The preferred solution is to interpose an adaptation layer (known as a *proxy*) between the clients and the servers. A different proxy may be designed for each class of client devices (e.g. phones, PDAs, etc.). The function of the proxy is to adapt the communication flow to and from the client to the client's capabilities and to the state of the network environment. To that end, the proxy uses its own storage and processing resources. Proxies may be hosted by dedicated equipment, as shown on Figure 1.4, or by common servers.

Examples of adaptation include compressing the data to adjust for variable network bandwidth; reducing the quality of images to adapt to restricted display capabilities; converting colors to levels of gray for black and white displays; caching data to cater for limited client storage capacity. A case study of the use of proxy-based adaptation is described in [Fox et al. 1998].

In all of the above situations, applications use intermediate software that resides on top of the operating systems and communication protocols to perform the following functions.

1. Hiding *distribution*, i.e. the fact that an application is usually made up of many interconnected parts running in distributed locations.

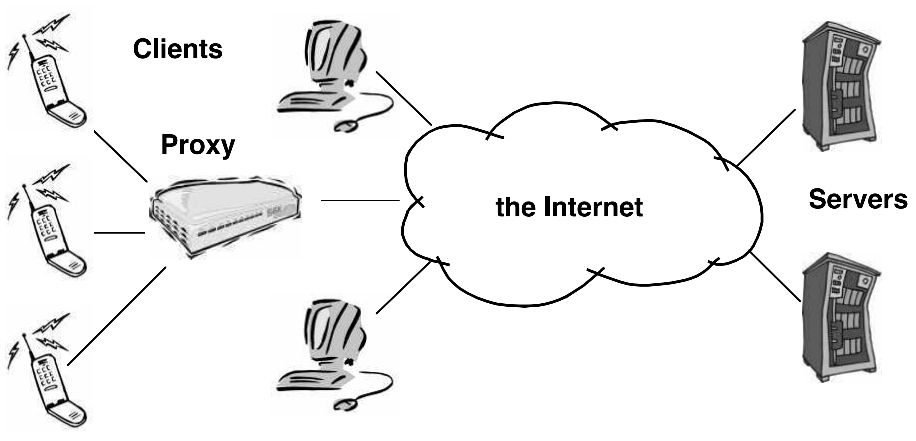

Figure 1.4. Adaptation to client resources through proxies

2. Hiding the *heterogeneity* of the various hardware components, operating systems and communication protocols that are used by the different parts of an application.

3. Providing uniform, standard, high-level *interfaces* to the application developers and integrators, so that applications can easily interoperate and be reused, ported, and composed.

4. Supplying a set of common *services* to perform various general purpose functions, in order to avoid duplicating efforts and to facilitate collaboration between applications.

These intermediate software layers have come to be known under the generic name of *middleware* (Figure 1.5). A middleware system may be general purpose, or may be dedicated to a specific class of applications.

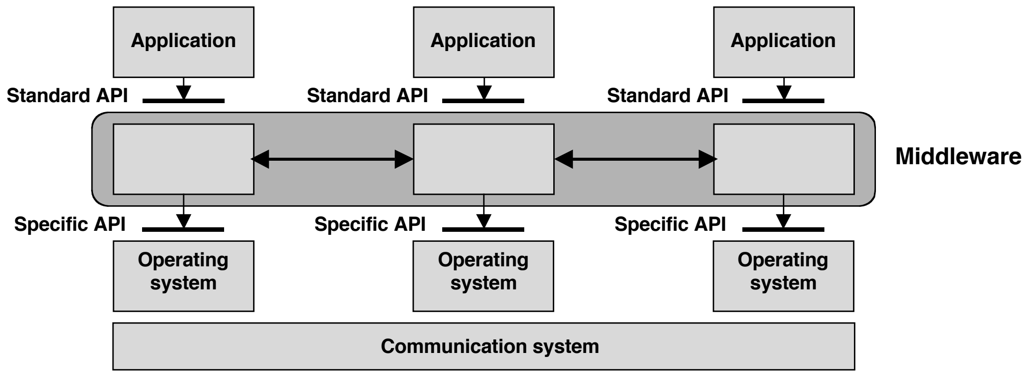

Figure 1.5. Middleware organization

Using middleware has many benefits, most of which derive from abstraction: hiding low-level details, providing language and platform independence, reusing expertise and possibly code, easing application evolution. As a consequence, one may expect a reduction in application development cost and time, better quality (since most efforts may be devoted to application specific problems), and better portability and interoperability.

A potential drawback is the possible performance penalty linked to the use of multiple software layers. Using middleware technologies may also entail a significant retraining effort for application developers.

1.2 Categories of Middleware

Middleware systems may be classified according to different criteria, including the properties of the communication infrastructure, the global architecture of the applications, the provided interfaces.

Communication properties. The communication infrastructure that underlies a middleware system is characterized by several properties that allow a first categorization.

1. *Fixed vs variable topology.* In a fixed communication system, the communicating entities reside at fixed locations, and the configuration of the network does not change (or such changes are programmed, infrequent operations). In a mobile (or nomadic) communication system, some or all communicating entities may change location, and entities may connect to or disconnect from the system, while applications are in progress.

2. *Predictable vs unpredictable characteristics.* In some communication systems, bounds can be established for performance factors such as latency or jitter. In many practical cases, however, such bounds are not known e.g. because the performance factors depend on the load on shared devices such as a router or a communication channel. A *synchronous* communication system is one in which an upper bound is known for the transmission time of a message; if such a bound cannot be predicted, the system is said to be *asynchronous*[1].

Usual combinations of these characteristics are defined as follows.

- Fixed, unpredictable. This is the most frequent case, both for local and wide area networks (e.g. the Internet). Although an average message transmission time may be estimated in many current situations, it is impossible to guarantee an upper bound.

- Fixed, predictable. This applies to environments developed for specially demanding applications such as hard real time control systems, which use a communication protocol that guarantees bounded message transfer time through resource reservation.

- Variable, unpredictable. This is the case of communication systems that include mobile (or nomadic) devices such as mobile phones or PDAs. Communication with such devices use wireless technologies, which are subject to unpredictable performance

[1]In a distributed system, the term *asynchronous* usually indicates, in addition, that an upper bound is not known for the ratio of processing speeds at different sites (a consequence of the unpredictable load on shared processors).

variations. So-called *ubiquitous* environments [Weiser 1993], in which a variety of devices may temporarily or permanently connect to a system, are also part of this category.

With current communication technologies, the category (variable, predictable) is void. The unpredictability of the communication system's characteristics imposes an additional load to middleware in order to guarantee specified performance levels. Adaptability, i.e. the ability to react to variations in communication performance, is the main quality required in this situation.

Architecture and interfaces. The overall architecture of a middleware system may be classified according to the following properties.

1. *Managed entities.* Middleware systems manage different kinds of entities, which differ by their definition, properties, and modes of communication. Typical examples of managed entities are objects, agents, and components (generic definitions of these terms are given in Chapters 5, 6, and 7, respectively, and more specific definitions are associated with particular systems).

2. *Service provision structure.* The entities managed by a middleware system may have predefined roles such as *client* (service requester) and *server* (service provider), or *publisher* (information supplier) and *subscriber* (information receiver). Alternatively, all entities may be at the same level and a given entity may indifferently assume different roles; such organizations are known as *peer to peer*.

3. *Service provision interfaces.* Communication primitives provided by a middleware system may follow the synchronous or asynchronous paradigm (unfortunately, these terms are overloaded, and their meaning here is different from that associated with the basic communication system). In synchronous communication, a client process sends a request message to a remote server and blocks while waiting for the reply. The remote server receives the request, executes the requested operation and sends a reply message back to the client. Upon receipt of the reply, the client resumes execution. In asynchronous communication, the send operation is non-blocking, and there may or not be a reply. Remote procedure calls or remote method invocations are examples of synchronous communication, while message queues and publish-subscribe systems are examples of asynchronous communication.

The various combinations of the above properties give rise to a wide variety of systems that differ through their visible structure and interfaces, and are illustrated by case studies throughout the book. In spite of this variety, we intend to show that a few common architectural principles apply to all these systems.

1.3 A Simple Instance of Middleware: Remote Procedure Call

We now present a simple middleware system, Remote Procedure Call (RPC). We do not intend to cover all the details of this mechanism, which may be found in all distributed systems textbooks, but to introduce a few patterns that will be found repeatedly in other middleware architectures, as well as some design issues.

1.3.1 Motivations and Requirements

Procedural abstraction is a key concept of programming. A procedure, in an imperative language, may be seen as a "black box" that performs a specified task by executing an encapsulated sequence of code (the procedure body). Encapsulation means that the procedure may only be called through an interface that specifies its parameters and return values as a set of typed holders (the formal parameters). When calling a procedure, a process specifies the actual parameters to be associated with the holders, performs the call, and gets the return values when the call returns (i.e. at the end of the execution of the procedure body).

The requirements of remote procedure call may be stated as follows. On a site A, consider a process p that executes a local call to a procedure P (Figure 1.6 a). Design a mechanism that would allow p to perform the same call, with the execution of P taking place on a remote site B (Figure 1.6 b), while preserving the semantics (i.e. the overall effect) of the call. We call A and B the client site and the server site, respectively, because RPC follows the client-server, or synchronous request-response, communication paradigm.

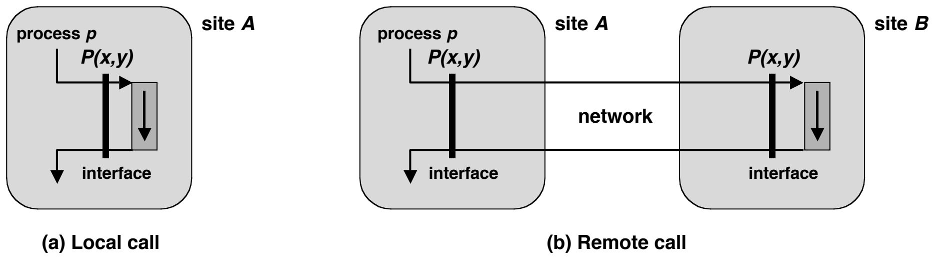

Figure 1.6. Remote procedure call: overview

By preserving the semantics between local and remote call, procedural abstraction is preserved; portability is improved because the application is independent of the underlying communication protocols. In addition, an application may easily be ported, without changes, between a local and a distributed environment.

However, preserving semantics is no easy task, for two main reasons.

- the failure modes are different in the local and distributed cases; in the latter, the client site, the server site and the network may fail independently;
- even in the absence of failures, the semantics of parameter passing is different (e.g. passing a pointer as a parameter does not work in the distributed case because the calling process and the procedure execute in distinct address spaces).

Concerning parameter passing, the usual solution is to use call by value for parameters of simple types. Fixed-size structures such as arrays or records can also be dealt with. In the general case, passing by reference is not supported, although solutions exist such as specific packing and unpacking routines for pointer-based structures. The technical aspects of parameter passing are examined in Section 1.3.2.

Concerning the behavior of RPC in the presence of failures, there are two main difficulties, at least if the underlying communication system is asynchronous (in the sense of unpredictable). First, it is usually impossible to know upper bounds on message transmission time; therefore a network failure detection method based on timeouts runs the risk of false detections. Second, it is difficult to distinguish between the loss of a message and the failure of a remote processor. As a consequence, a recovery action may lead to a wrong decision such as re-executing an already executed procedure. Fault tolerance aspects are examined in Section 1.3.2.

1.3.2 Implementation Principles

The standard implementation of RPC [Birrell and Nelson 1984] relies on two pieces of software, the client stub and the server stub (Figure 1.7). The client stub acts as a local representative of the server on the client site; the server stub has a symmetrical role. Thus both the calling process (on the client side) and the procedure body (on the server side) keep the same interface as in the centralized case. The client and server stubs rely on a communication subsystem to exchange messages. In addition, they use a naming service in order to help the client locate the server (this point is developed in section 1.3.3).

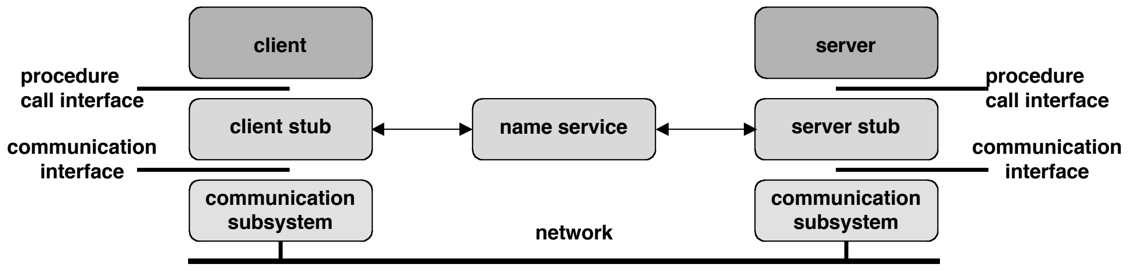

Figure 1.7. Remote procedure call: main components

The functions of the stubs are summarized below.

Process management and synchronization. On the client side, the calling process (or thread, depending on how the execution is organized) must be blocked while waiting for the procedure to return.

On the server side, the main issue is that of parallel execution. While a procedure call is a sequential operation, the server may be used by multiple clients. Multiplexing the server resources (specially if the server machine is a multiprocessor or a cluster) calls for a multithreaded organization. A daemon thread waits for incoming messages on a specified port. In the single thread solution (Figure 1.8 (a)), the daemon thread executes the procedure; there is no parallel execution on the server side. In the second scheme (Figure 1.8 (b)), a new worker thread is created in order to execute the procedure, while the daemon returns to wait on the next call; the worker thread exits upon completion. In order to avoid the overhead due to thread creation, an alternate solution is to manage a fixed-size pool of worker threads (Figure 1.8 (c)). Worker threads communicate with the daemon through a shared buffer using the producer-consumer scheme. Worker threads are waiting for new work to arrive; after executing the procedure, a worker thread returns to the pool, i.e. tries to get new work to do. If all worker threads are busy when a call arrives, the execution of the call is delayed until a thread becomes free.

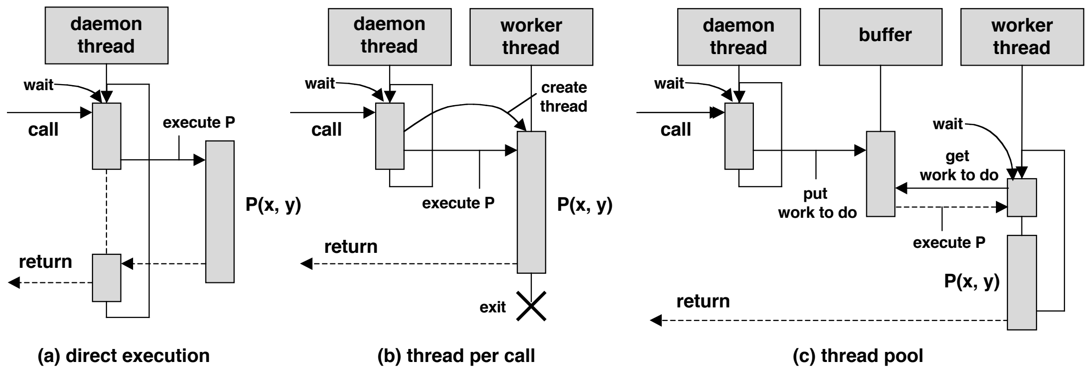

Figure 1.8. Remote procedure call: thread management on the server side

A discussion of the thread management patterns, illustrated by Java threads, may be found in [Lea 1999].

All these synchronization and thread management operations are performed by the stubs and are invisible to the client and server main programs.

Parameter marshalling and unmarshalling. Parameters and results need to be transmitted over the network. Therefore they need to be put in a serialized form, suitable for transmission. In order to ensure portability, this form should be standard and independent of the underlying communication protocols as well as of the local data representation conventions (e.g. byte ordering) on the client and server machines. Converting data from a local representation to the standard serialized form is called *marshalling*; the reverse conversion is called *unmarshalling*.

A marshaller is a set of routines, one for each data type (e.g. `writeInt`, `writeString`, etc.), that write data of the specified type to a sequential data stream. An unmarshaller performs the reverse function and provides routines (e.g. `readInt`, `readString`, etc.) that extract data of a specified type from a sequential data stream. These routines are called by the stubs when conversion is needed. The interface and organization of marshallers and unmarshallers depend on the language used, which specifies the data types, and on the standard representation format.

Reacting to failures. As already mentioned, failures may occur on the client site, on the server site, and in the communication network. Taking potential failures into account is a three step process: formulating failure hypotheses; detecting failures; reacting to failure detection.

The usual failure hypotheses are fail-stop for nodes (i.e. either the node operates correctly, or it stops), and message loss for communication (i.e. either a message arrives uncorrupted, or it does not arrive at all, assuming that message corruption is dealt with at the lower levels of the communication system). Failure detection mechanisms are based on timeouts. When a message is sent, a timeout is set at an estimated upper bound of the expected time for receipt of a reply. If the timeout is triggered, a recovery action is taken.

Such timeouts are set both on the client site (after sending the call message) and on the server site (after sending the reply message). In both cases, the message is resent after timeout. The problem is that an upper bound cannot be safely estimated; a call message may be resent in a situation where the call has already been executed; the call might then be executed several times.

The net result is that it is usually impossible to guarantee the so-called "exactly once" semantics, meaning that, after all failures have been repaired, the call has been executed exactly one time. Most systems guarantee an "at most once" semantics (the call is either executed, or not at all, but partial or multiple executions are precluded). The "at least once" semantics, in which the call is executed one or more times, is acceptable when the call is idempotent, i.e. the effect of two calls in succession is identical to that of a single call.

The overall organization of RPC, not showing the aspects related to fault tolerance, is described on Figure 1.9

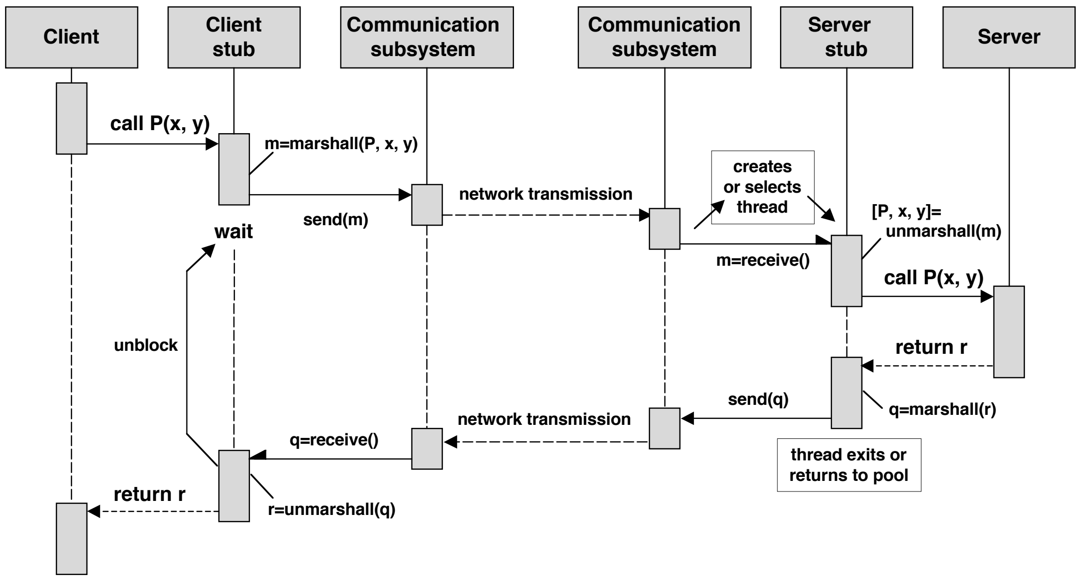

Figure 1.9. Remote procedure call: overall flow of control

1.3.3 Developing Applications with RPC

In order to actually develop an application using RPC, a number of practical issues need to be settled: how are the client and the server linked together? how are the client and server stubs constructed? how are the programs installed and started? These issues are considered in turn below.

Client-Server Binding. The client needs to know the server address and port number to which the call should be directed. This information may be statically known, and hardwired in the code. However, in order to preserve abstraction, to ensure a flexible management of resources, and to increase availability, it is preferable to allow for a late binding of the remote execution site. Therefore, the client must locate the server site prior to the call.

This is the function of a naming service, which is essentially a registry that associates procedure names (and possibly version numbers) with server addresses and port numbers. A server registers a procedure name together with its IP address and the port number at which the daemon process is waiting for calls on that procedure. A client consults the naming service to get the IP address and port number for a given procedure. The naming service is usually implemented as a server of its own, whose address and port number are known to all participant nodes.

Figure 1.10 shows the overall pattern of interaction involving the naming service.

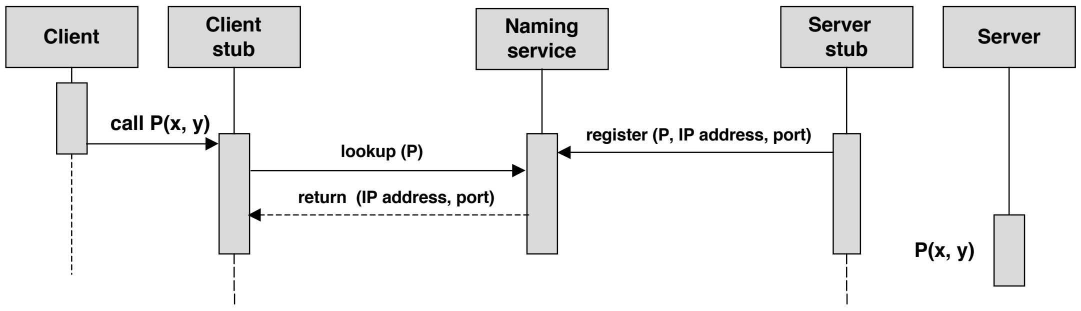

Figure 1.10. Remote procedure call: locating the server

The problem of binding the server to the client is easily solved by including the address and port number of the client in the call message.

Stub generation. As seen in Section 1.3.2, the stubs fulfill a set of well-defined functions, part of which is generic (e.g. process management) and part of which depends on the specific call (e.g. parameter marshalling and unmarshalling). Considering this fixed pattern in their structure, stubs are obvious candidates for automatic generation.

The call-specific parameters needed for stub generation are specified in a special notation known as an Interface Definition Language (IDL). An interface description written in IDL contains all the information that defines the interface of the procedure call: it acts as a contract between the caller and the callee. For each parameter, the description specifies its type and mode of passing (e.g. by value, by copy-restore, etc.). Additional information such as version number and mode of activation may also be specified.

Several IDLs have been defined (e.g. Sun XDR, OSF DCE). The stub generator is associated with a specific common data representation format associated with the IDL;

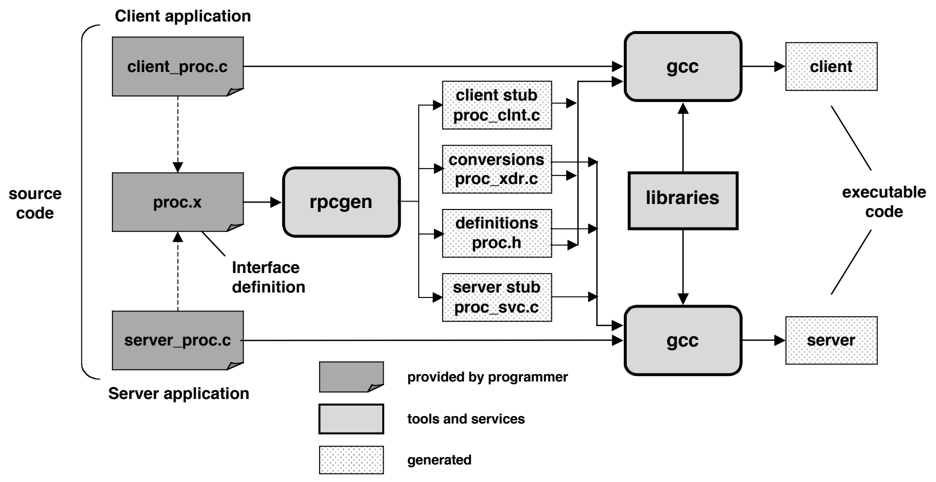

Figure 1.11. Remote procedure call: stub generation

it inserts the conversion routines provided by the corresponding marshallers and unmarshallers. The overall development cycle of an application using RPC is shown on Figure 1.11 (the notation is that of Sun RPC).

Application Deployment. Deployment is the process of installing the program pieces that make up a distributed application on their respective sites and of starting their execution when needed. In the case of RPC, the installation is usually done by executing prepared scripts that call the tools described in Figure 1.11, possibly using a distributed file system to retrieve the source files. As regards activation, the constraints are that the server needs to be activated before the first call of a client, and that the naming service must be activated before the server. These activations may again be performed by a script; the server may also be directly activated by a client (in that case, the client needs to be allowed to run a remote script on the server's site).

1.3.4 Summary and Conclusions

A number of useful lessons can be learned from this simple case.

1. Complete transparency (i.e. the property of preserving the behavior of an application while moving from a centralized to a distributed environment) is not achievable. While transparency was an ideal goal in the early days of middleware, good engineering practice leads to recognize its limits and to accept that distributed applications should be considered as such, i.e. distribution aware, at least for those aspects that require it, like fault tolerance or performance. This point is discussed in [Waldo et al. 1997].

2. Several useful patterns have emerged. Using local representatives to organize the communication between remote entities is one of the most common patterns of middleware (the Proxy design pattern). Another universal pattern is client-server

matching through a naming service acting as a registry (the Broker architectural pattern). Other patterns, perhaps less apparent, are the organization of server activity through thread creation or pooling, and the reaction to failures through the detection-reaction scheme.

3. Developing a distributed application, even with an execution scheme as conceptually simple as RPC, involves an important engineering infrastructure: IDL and stub generators, common data representation and (un)marshallers, fault tolerance mechanisms, naming service, deployment tools. Designing this infrastructure in order to simplify the application developers' task is a recurring theme of this book.

Regarding RPC as a tool for structuring distributed applications, we may note a number of limitations.

- The structure of the application is static; there is no provision for dynamic creation of servers or for restructuring an application.
- Communication is restricted to a synchronous scheme. There is no provision for asynchronous, event driven, communication.
- The data managed by the client and server programs are not persistent, i.e. they do not survive the processes that created them. Of course, these data may be saved in files, but the save and restore operations must be done explicitly: there is no provision for automatic persistence.

In the rest of this book, we introduce other schemes that do not suffer from these limitations.

1.4 Issues and Challenges in Middleware Design

In this section, we first discuss the main issues of middleware organization, which define the structure of the rest of the book. We then identify a few general requirements that favor a principled design approach. We conclude by listing a few challenges for the designers of future middleware systems.

1.4.1 Design Issues

The function of middleware is to mediate interaction between the parts of an application, or between applications. Therefore *architectural* issues play a central role in middleware design. Architecture is concerned with the organization, overall structure, and communication patterns, both for applications and for middleware itself. Architectural issues are the subject of Chapter 2. In addition to a discussion of basic organization principles, this chapter presents a set of basic patterns, which are recurring in the design of all categories of middleware.

Besides architectural aspects, the main problems of middleware design are those pertaining to various aspects of distributed systems. A brief summary follows, which defines the plan of the rest of the book.

Naming and binding are central in middleware design, since middleware may be defined as software that binds pieces of application software together. Naming and binding are the subject of Chapter 3.

Any middleware system relies on a communication layer that allows its different pieces to interoperate. In addition, communication is a function provided by middleware itself to applications, in which the communicating entities may take on different roles such as client-server or peer to peer. Communication is the subject of Chapter 4.

Middleware allows different interaction modes (synchronous invocations, asynchronous message passing, coordination through shared objects) embodied in different patterns. The main paradigms of middleware organization using distributed objects, mainly in client-server mode, are examined in Chapter 5. The paradigms based on asynchronous events and coordination are the subject of Chapter 6.

Software architecture deals with the structural description of a system in terms of elementary parts. The notions related to composition and components are now becoming a key issue for middleware, both for its own organization and for that of the applications it supports. Software composition is the subject of Chapter 7.

Data management brings up the issues of persistence (long term data conservation and access procedures) and transactions (accessing data while preserving consistency in the face of concurrent access and possible failures). Persistence is the subject of Chapter 8, and transactions are examined in Chapter 9.

Administration is a part of the application's life cycle that is taking an increasing importance. It involves such functions as configuration and deployment, monitoring, and reconfiguration. Administration middleware is the subject of Chapter 10.

Quality of Service includes various properties of an application that are not explicitly formulated in its functional interfaces, but that are essential for its users. Three specific aspects of QoS are reliability and availability, performance (specially for time-critical applications), and security. They are respectively covered in Chapters 11, 12, and 13.

1.4.2 Architectural Guidelines

In this section, we briefly discuss a few considerations derived from experience. They are generally applicable to software systems, but are specially relevant to middleware, due to its dual function of mediator and common services provider.

Models and Specifications

A *model* is a simplified representation of (part of) the real world. A given real object may be represented by different models, according to the domain of interest and to the accuracy and degree of detail of the representation. Models are used to better understand the object being represented, by explicitly formulating relevant hypotheses, and by deriving useful properties. Since no model is a perfect representation of the reality, care must be taken when transposing the results derived from a model to the real world. A discussion of the use (and usefulness) of models for distributed systems may be found in [Schneider 1993].

Models (with various degrees of formality) are used for various aspects of middleware, including naming, composition, fault tolerance, and security.

Models help in the formulation of rigorous specifications. When applied to the behavior of a system, specifications fall into two categories:

- *Safety* (informally: no undesirable event or condition will ever occur).
- *Liveness* (informally: a desirable event or condition will eventually occur).

For example, in a communication system, a safety property is that a delivered message is not corrupted (i.e., it is identical to the message sent), while an example of liveness is that a message sent will eventually be delivered to its destination. Liveness is often more difficult to ensure than safety. Specifications of concurrent and distributed systems are discussed in [Weihl 1993].

Separation of Concerns

In software engineering, *separation of concerns* refers to the ability to isolate independent, or loosely related, aspects of a design and to deal with each of them separately. The expected benefits are to allow the designer and developer to concentrate on one problem at a time, to eliminate artificial interactions between orthogonal concerns, and to allow independent variation of the requirements and constraints associated with each separate aspect. Separation of concerns has deep implications both on the architecture of middleware and on the definition of roles for the division of the design and implementation tasks.

Separation of concerns may be viewed as a “meta-principle” that can take a number of specific forms, of which four examples follow.

- The principle of encapsulation (2.1.3) separates the concerns of the user of a software component from those of its implementor, through a common interface definition.
- The principle of abstraction allows a complex system to be decomposed into levels (2.2.1), each level providing a view that hides irrelevant details which are dealt with at lower levels.
- Separation between policy and mechanism [Levin et al. 1975] is a widely used principle, specially in the area of resource management and protection. This separation brings flexibility for the policy designer, while avoiding overspecification of the mechanisms. It should be possible to change a policy without having to reimplement the mechanisms.
- The principle of orthogonal persistence (8.2) separates the issue of defining the lifetime of data from other aspects such as the type of the data or the properties of their access programs.

These points are further developed in Chapter 2.

In a more restricted sense, separation of concerns attempts to deal with aspects whose implementation, in the current state of the art, is scattered among various parts of a software system, and tightly interwoven with other aspects. The goal is to allow a separate expression of aspects that are considered independent and, ultimately, to be able to automate the task of producing the code dealing with each aspect. Examples of such aspects are those related to "extra-functional" properties (see 2.1.2) such as availability, security, persistence, and those dealing with common functions such as logging, debugging, observation, transaction management, which are typically implemented by pieces of code scattered in many parts of an application.

Separation of concerns also helps identifying specialized roles in the design and development process, both for applications and for the middleware itself. By focusing on a particular aspect, the person or team that embodies a role can better apply his expertise and improve the efficiency of his work. Examples of roles associated with various aspects of component software may be found in Chapter 7.

Evolution and Adaptation

Software systems operate in a changing environment. Sources of change include evolving requirements resulting from the perception of new needs, and varying execution conditions due to the diversity of communication devices and systems, which induce unpredictable variations of quality of service. Therefore both applications and middleware need to be designed for change. Responding to evolving requirements is done by program evolution. Responding to changing execution conditions is done by dynamic adaptation.

In order to allow evolution, the internal structure of the system must be made accessible. There is an apparent contradiction between this requirement and the principle of encapsulation, which tends to hide implementation details.

There are several ways to deal with this problem. Pragmatic techniques, often based on interception (2.3.4), are widely used for commercial middleware. A more systematic approach is to use reflection. A *reflective* system [Smith 1982, Maes 1987] is one that provides a representation of itself, in order to enable inspection (answering questions about the system) and adaptation (modifying the behavior of the system). To ensure consistency, the representation must be *causally connected* to the system, i.e. any change of the system must be reflected in its representation, and vice versa. Meta-object protocols provide such an explicit representation of the basic mechanisms of a system and a protocol to examine and to modify this representation. Aspect-oriented programming, a technology designed to ensure separation of concerns, is also useful for implementing dynamic evolution capabilities. These techniques and their use in middleware systems are reviewed in Chapter 2.

1.4.3 Challenges

The designers of future middleware systems face several challenges.

- Performance. Middleware systems rely on interception and indirection mechanisms, which induce performance penalties. Adaptable middleware introduces additional indirections, which make the situation even worse. This problem may be alleviated by various optimization methods, which aim at eliminating the unnecessary overheads by such techniques as inlining, i.e. injecting the middleware code directly into the application. Flexibility must be preserved, by allowing the effect of the optimizations to be reversed if needed.

- Large scale. As applications become more and more interconnected and interdependent, the number of objects, users and devices tends to increase. This poses the problem of the scalability of the communication and object management algorithms, and increases the complexity of administration (for example, does it even make sense to try to define and capture the "state" of a very large system?). Large scale also complexifies the task of preserving the various forms of Quality of Service.

- Ubiquity. Ubiquitous (or pervasive) computing is a vision of the near future, in which an increasing number of devices embedded in various physical objects will be participating in a global information network. Mobility and dynamic reconfiguration will be dominant features, requiring permanent adaptation of the applications. Most of the architectural concepts applicable to systems for ubiquitous computing are still to be elaborated.

- Management. Managing large applications that are heterogeneous, widely distributed and in permanent evolution raises many questions, such as consistent observation, security, tradeoffs between autonomy and interdependence for the different subsystems, definition and implementation of resource management policies.

1.5 Historical Note

The term "middleware" seems to have appeared around 1990, but middleware systems existed long before that date. Messaging systems were available as products in the late 1970s. The classical reference on Remote Procedure Call implementation is [Birrell and Nelson 1984], but RPC-like, language-specific constructs were already in use by then (the original idea of RPC appeared in [White 1976][2] and an early implementation was proposed in [Brinch Hansen 1978]).

Starting in the mid-1980s, a number of research projects developed middleware support for distributed objects, and elaborated the main concepts that influenced later standards and products. Early efforts are Cronus [Schantz et al. 1986] and Eden [Almes et al. 1985]. Later projects include Amoeba [Mullender et al. 1990], ANSAware [ANSA], Arjuna [Parrington et al. 1995], Argus [Liskov 1988], Chorus/COOL [Lea et al. 1993], Clouds [Dasgupta et al. 1989], Comandos [Cahill et al. 1994], Emerald [Jul et al. 1988], Gothic [Banâtre and Banâtre 1991], Guide [Balter et al. 1991], Network Objects [Birrell et al. 1995], SOS [Shapiro et al. 1989], and Spring [Mitchell et al. 1994].

The Open Software Foundation (OSF), later to become the Open Group [Open Group], was created in 1988 in an attempt to unify the various versions of the Unix operating system. While this goal was never reached, the OSF specified a software suite, the Distributed Computing Environment (DCE) [Lendenmann 1996], which included such middleware components as an RPC service, a distributed file system, a distributed time service, and a security service.

The Object Management Group (OMG) [OMG] was created in 1989 in order to define standards for distributed object middleware. Its first effort led to the CORBA 1.0

[2]an extended version of Internet RFC 707.

specification in 1991 (the latest version, as of 2003, is CORBA 3). Later developments include standards for modeling (UML, MOF) and components (CCM). The Object Database Management Group (ODMG) [ODMG] defines standards applicable to object databases, bridging the gap between object-oriented programming languages and persistent data management.

The Reference Model for Open Distributed Processing (RM-ODP) [ODP 1995a], [ODP 1995b] was jointly defined by two standards bodies, ISO and ITU-T. Its contribution is a set of concepts that define a generic framework for open distributed computing, rather than a specific standard.

The definition of the Java programming language by Sun Microsystems in 1995 led the way to several middleware developments such as Java Remote Method Invocation (RMI) [Wollrath et al. 1996] and the Enterprise JavaBeans (EJB) [Monson-Haefel 2002]. These and others are integrated in a common platform, J2EE [J2EE].

Microsoft developed the Distributed Component Object Model (DCOM) [Grimes 1997], a middleware based on composable distributed objects, and an improved version, COM+ [Platt 1999]. Its next offering is .NET [.NET], a software platform for distributed applications development and Web services provision.

The first scientific conference entirely dedicated to middleware took place in 1998 [Middleware 1998]. Current research issues include adaptive and reflective middleware, and middleware for mobile and ubiquitous systems.

References

[Almes et al. 1985] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D. (1985). The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59.

[ANSA] ANSA. Advanced networked systems architecture. http://www.ansa.co.uk/.

[Balter et al. 1991] Balter, R., Bernadat, J., Decouchant, D., Duda, A., Freyssinet, A., Krakowiak, S., Meysembourg, M., Le Dot, P., Nguyen Van, H., Paire, E., Riveill, M., Roisin, C., Rousset de Pina, X., Scioville, R., and Vandôme, G. (1991). Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–67.

[Banâtre and Banâtre 1991] Banâtre, J.-P. and Banâtre, M., editors (1991). *Les systèmes distribués : expérience du projet Gothic*. InterÉditions.

[Birrell and Nelson 1984] Birrell, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59.

[Birrell et al. 1995] Birrell, A. D., Nelson, G., Owicki, S., and Wobber, E. (1995). Network objects. *Software–Practice and Experience*, 25(S4):87–130.

[Brinch Hansen 1978] Brinch Hansen, P. (1978). Distributed Processes: a concurrent programming concept. *Communications of the ACM*, 21(11):934–941.

[Cahill et al. 1994] Cahill, V., Balter, R., Harris, N., and Rousset de Pina, X., editors (1994). *The COMANDOS Distributed Application Platform.* ESPRIT Research Reports. Springer-Verlag. 312 pp.

[Dasgupta et al. 1989] Dasgupta, P., Chen, R. C., Menon, S., Pearson, M. P., Ananthanarayanan, R., Ramachandran, U., Ahamad, M., LeBlanc, R. J., Appelbe, W. F., Bernabéu-Aubán, J. M., Hutto, P. W., Khalidi, M. Y. A., and Wilkenloh, C. J. (1989). The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1):11–46.