Middleware Architecture
with Patterns and Frameworks

Sacha Krakowiak

February 27, 2009

Contents

Preface

In a distributed computing system, *middleware* is defined as the software layer that lies between the operating system and the applications on each site of the system. Its role is to make application development easier, by providing common programming abstractions, by masking the heterogeneity and the distribution of the underlying hardware and operating systems, and by hiding low-level programming details.

Stimulated by the growth of network-based applications, middleware technologies are taking an increasing importance. They cover a wide range of software systems, including distributed objects and components, message-oriented communication, and mobile application support. Although new acronyms seem to appear almost every month in the area of middleware, these software systems are based on a few principles and paradigms that have been identified over the years, refined through experience, and embodied in working code. The aim of this book is to contribute to the elaboration and transmission of this body of knowledge for the benefit of designers and developers of future middleware systems. We believe that design patterns and software frameworks are adequate vehicles for achieving this goal.

This book approaches middleware systems from an architectural point of view. Architecture is the art of organizing parts into a whole to fulfill a function, in the presence of constraints. Architects and city planners [Alexander et al. 1977] have introduced the notion of a *design pattern* to describe, in an articulated way, the structures, the representations and the techniques used to respond to specific requirements in a given context. This notion has been adopted by software architects [Gamma et al. 1994], and is now central in software design. As experience builds up, new patterns are elaborated, and an expanding body of literature is devoted to the presentation and discussion of patterns for a variety of situations (see e.g., [Buschmann et al. 1995], [Schmidt et al. 2000] for a review of patterns applicable to distributed systems).

Design patterns allow software architects to reuse proven designs. Likewise, *software frameworks* [Johnson 1997] allow software developers to reuse working code. A software framework is a program skeleton that may be directly reused, or adapted according to well-defined rules, to solve a family of related problems. A framework usually implements a design pattern, and often uses several patterns in combination. Although the notion of a software framework is language-independent, it has mostly been used with object-oriented languages. In this context, a framework is a set of classes that may be adapted for specific environments and constraints, using well-defined rules of usage (e.g., overloading specified methods, etc.). As emphasized in [Schmidt and Buschmann 2003], patterns, frameworks, and middleware play a complementary role for improving the process of designing, building and documenting the increasingly complex applications of today.

The organization of this book is directed by architectural considerations, not by categories of middleware. For each function that a middleware system should fulfill, we summarize the main design issues and we present possible solutions in terms of design patterns. Some of these patterns are well established, and widely documented in the literature; others are more recent and are only described in research papers.

While the emphasis of this book is on design principles and paradigms of middleware systems, we think it important that these principles and paradigms be illustrated by real life examples of working code. We believe in the tenet of literate programming [Knuth 1992]: programs should be intended for reading as well as for execution. Therefore each chapter contains a discussion of a software framework related to the problem under study. These frameworks are mostly borrowed from software developed by ObjectWeb, a consortium of academic and industrial partners dedicated to the development of open source middleware. The actual source code, together with its documentation and prepared examples, is available from the ObjectWeb[1] site.

Although this book touches on various aspects of distributed systems, it is not intended as an introduction to distributed systems principles. These are covered in several textbooks such as [Coulouris et al. 2005, Tanenbaum and van Steen 2006, Veríssimo and Rodrigues 2001].

This book addresses two audiences: students of last-year undergraduate and introductory postgraduate courses, and designers and developers of middleware systems. For the students, this book may complement a course on distributed systems or a course on software engineering. The prerequisites are a basic knowledge of networking and distributed systems principles, and some practice in programming distributed applications. Familiarity with the Java language is required to follow the examples.

Acknowledgments I am indebted to my colleagues, former colleagues, and students of Project Sardes[2] for many fruitful exchanges on the subject matter of this book. Interaction with colleagues and participants of the ICAR[3] series of summer schools also provided motivation and feedback.

[1]http://www.objectweb.org/
[2]http://sardes.inrialpes.fr/
[3]http://sardes.inrialpes.fr/past-events/summer-schools/past-summer-schools.html#schools

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.

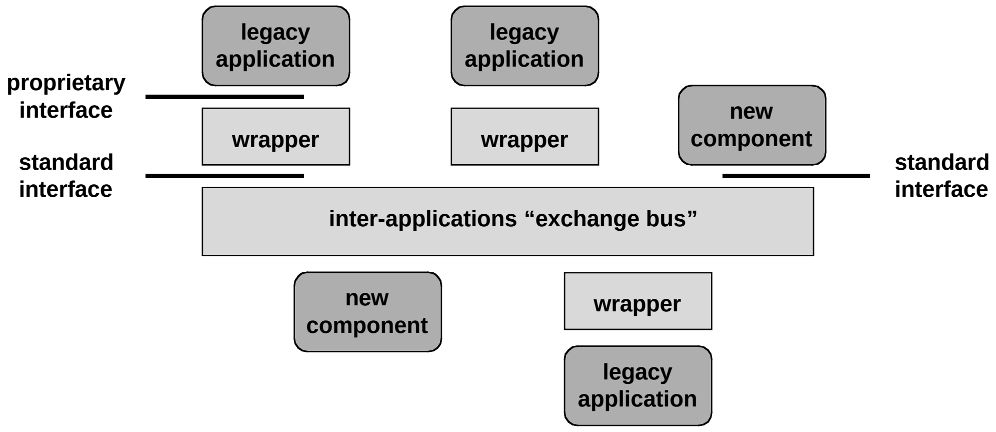


Figure 1.1. Integrating legacy applications

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.

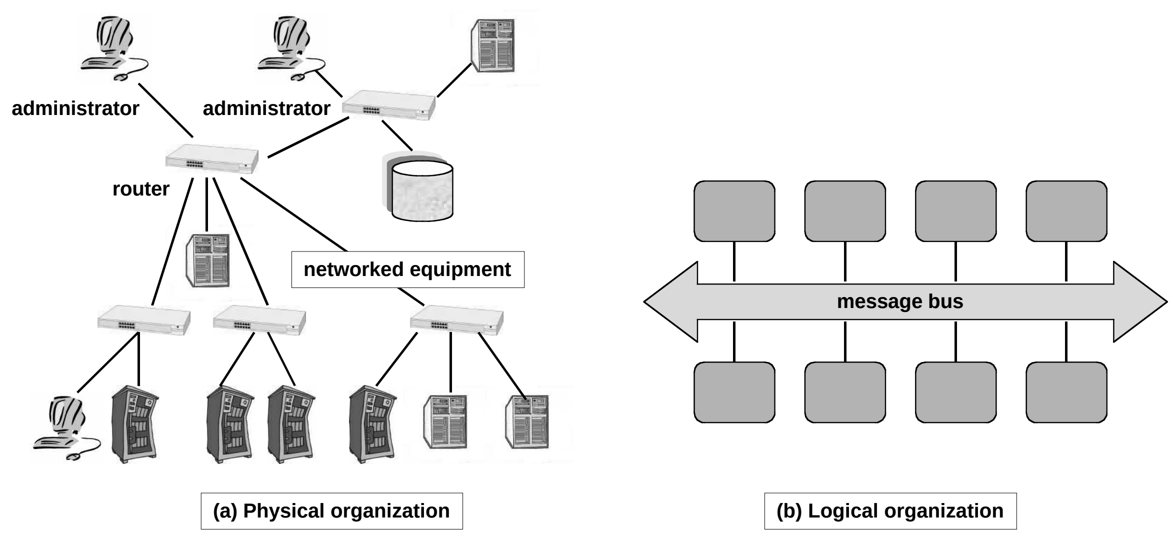


Figure 1.2. Monitoring and controlling networked equipment

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.

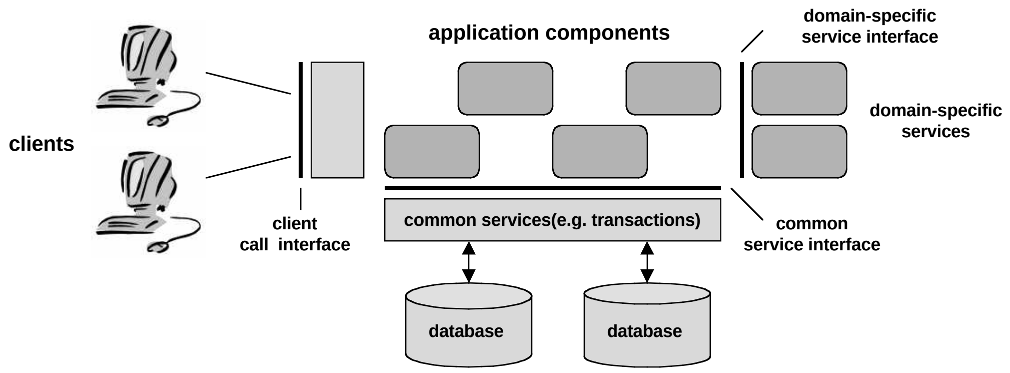


Figure 1.3. An environment for component-based applications

- 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.

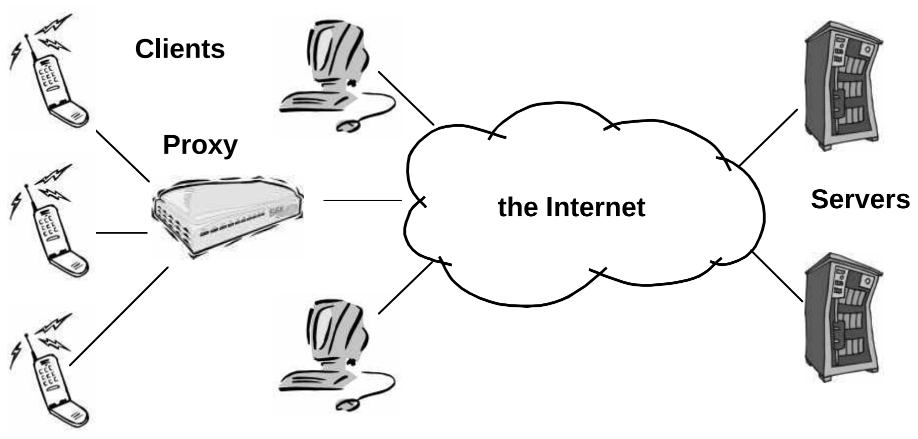


Figure 1.4. Adaptation to client resources through proxies

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.

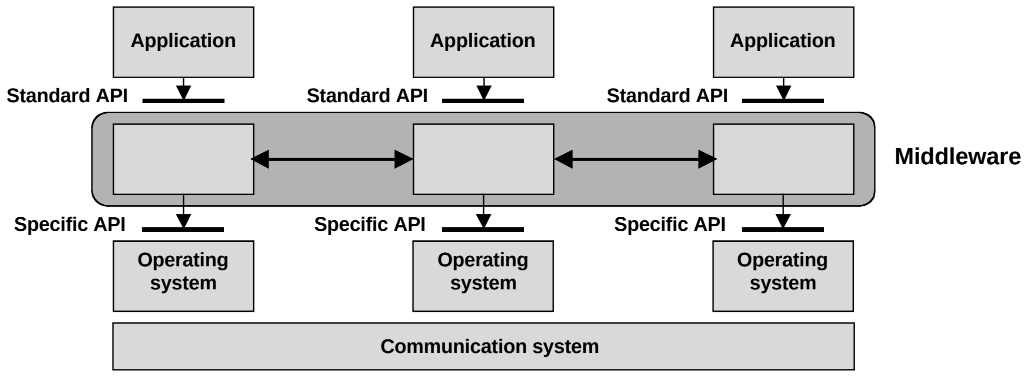


Figure 1.5. Middleware organization

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.

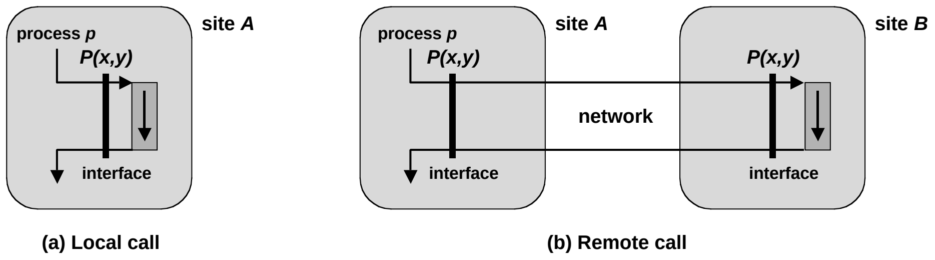


Figure 1.6. Remote procedure call: overview

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).

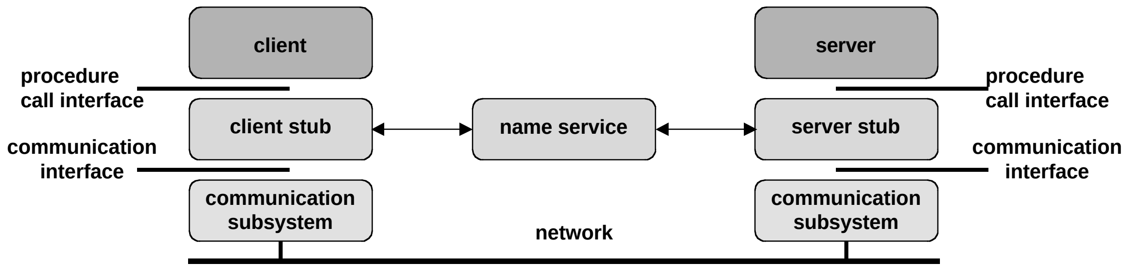


Figure 1.7. Remote procedure call: main components

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.

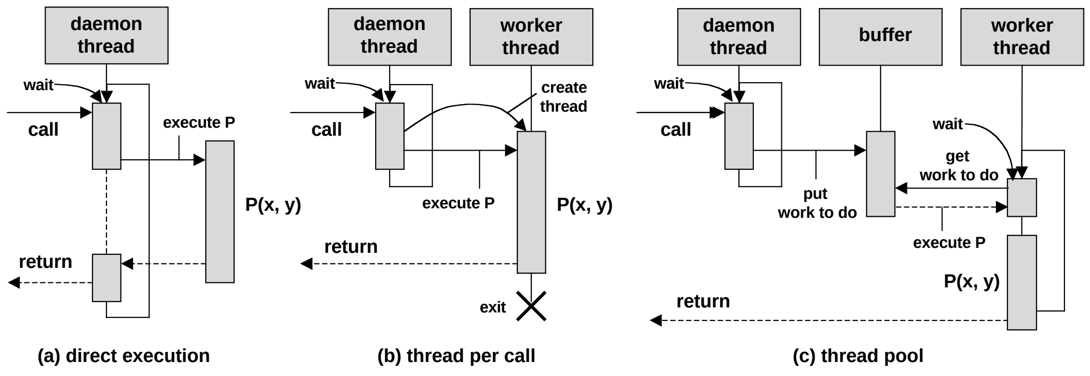


Figure 1.8. Remote procedure call: thread management on the server side

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

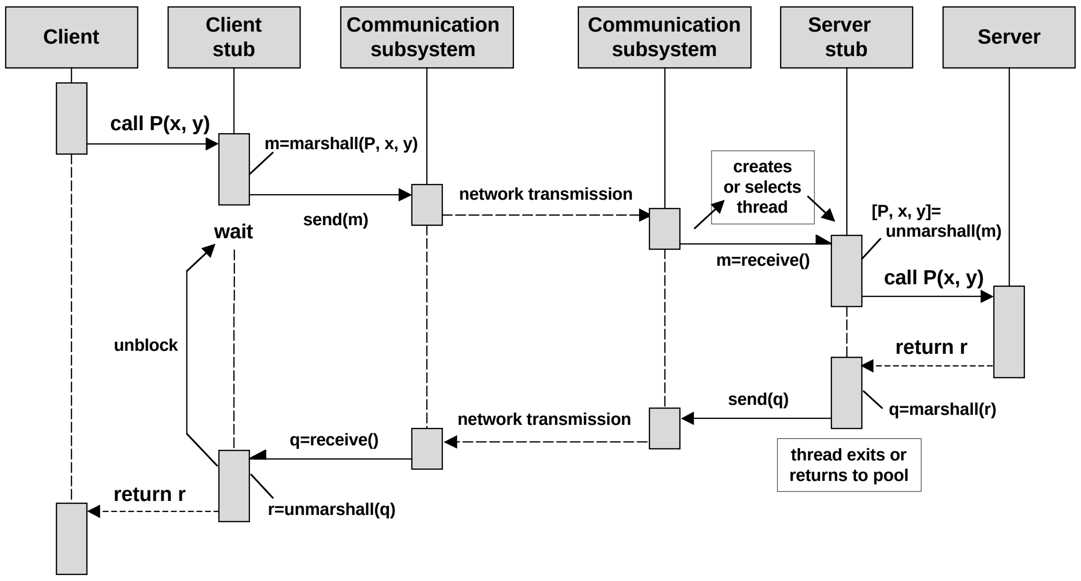


Figure 1.9. Remote procedure call: overall flow of control

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.

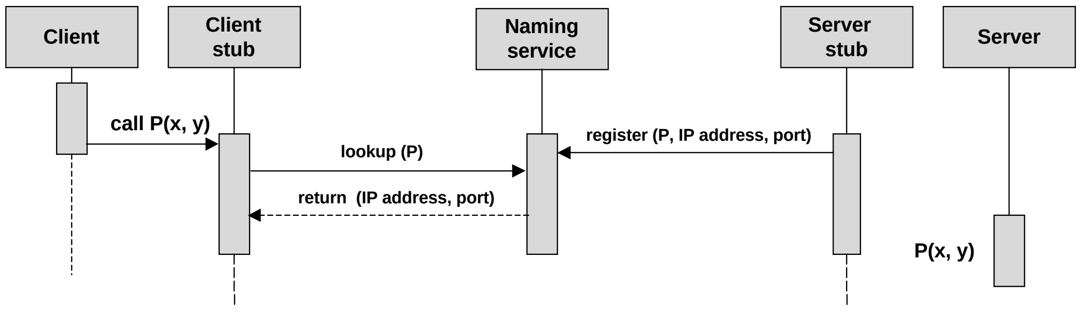


Figure 1.10. Remote procedure call: locating the server

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;

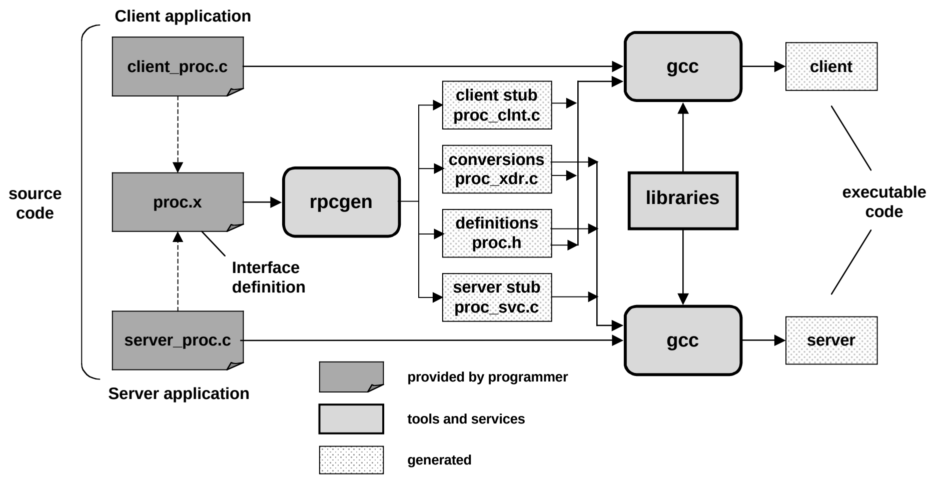


Figure 1.11. Remote procedure call: stub generation

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 1993b].

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.

Chapter 2

Middleware Principles and Basic Patterns

In this chapter, we present the main design principles of middleware systems, together with a few basic patterns that are recurring in all middleware architectures. A number of more elaborate patterns are built by extending and combining these basic constructs. The chapter starts with a presentation of the architectural principles and main building blocks of middleware systems, including distributed objects and multi-layer organizations. It goes on with a discussion of the basic patterns related to distributed objects. The chapter concludes with a presentation of patterns related to separation of concerns, including a discussion on implementation techniques for reflective middleware.

2.1 Services and Interfaces

A (hardware and/or software) system is organized as a set of parts, or components[1]. The system as a whole, and each of its components, fulfills a function that may be described as the provision of a *service*. Quoting a definition in [Bieber and Carpenter 2002], "a service is a contractually defined behavior that can be implemented and provided by any component for use by any component, based solely on the contract".

In order to provide its service, a component usually relies on services provided to it by other components. For uniformity's sake, the system as a whole may be regarded as a component, which interacts with an externally defined environment; the service provided by the system relies on assumptions about the services that the environment provides to the system[2].

Service provision may be considered at different levels of abstraction. A provided service is usually embodied in a set of interfaces, each of which represents an aspect of the service. The use of these interfaces relies on elementary interaction patterns between

[1]In this chapter, we use the word *component* in a non-technical sense, to mean a unit of system decomposition. This notion is further elaborated in Chapter 7.

[2]e.g. a computer delivers a specified service, on the provision of a specified power supply, and within a specified range of environmental conditions, such as temperature, humidity, etc.

the components. In 2.1.1, we first briefly review these interaction patterns. Interfaces are further discussed in 2.1.2, and contracts are the subject of 2.1.3.

2.1.1 Basic Interaction Mechanisms

Components interact through an underlying communication system. Communication is examined in Chapter 4. Here we give an overview of a few common patterns that occur in service provision.

The simplest form of communication is an asynchronous transient event (Figure 2.1a). Component *A* (more precisely, a thread executing in component *A*) produces an event (i.e. sends an elementary message to a specified set of recipients), and continues execution. The message may just be a signal, or it may carry a value. The "transient" attribute means that the message is lost if no recipient is waiting for it. Reception of the event by component *B* triggers a reaction, i.e. starts the execution of a program (the handler) associated with that event. This mechanism may be used by *A* to request a service from *B*, when no result is expected; or it may be used by *B* to observe or monitor the activity of *A*. Communication using events is further discussed in Chapter 6.

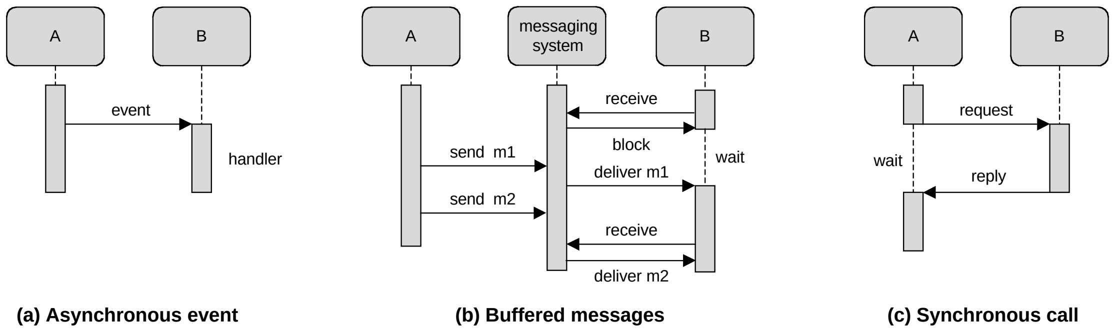


Figure 2.1. Some basic interaction mechanisms

A more elaborate form of communication is asynchronous persistent message passing (2.1b). A message is a chunk of information that is transmitted by a sender to a receiver. The "persistent" attribute means that the communication system provides a buffering function: if the receiver is waiting for the message, the communication system delivers it; if not, the message remains available until the receiver attempts to read it. Communication by messages is further discussed in Chapter 4.

Another usual mechanism is synchronous call (2.1c), in which *A* (the customer of a service provided by *B*) sends a request message to *B* and waits for a reply. This pattern is that used in RPC, as seen in 1.3.

Synchronous and asynchronous interactions may be combined, e.g. in various forms of "asynchronous RPC". The intent is to allow the service requester to continue execution after issuing the request. The problem is then for the requester to retrieve the results, which may be done in several ways. For instance, the provider may inform the requester, by an asynchronous event, that the results are available; or the requester may call the provider at a later time to find out about the state of the execution.

It may happen that the provision of a service by B to A relies on the use by B of a service provided by A (the contract between service provider and customer implies commitment by both parties). For instance, in Figure 2.2a, the execution of the call from A to B relies on a *callback* from B to a function provided by A. In the example, the callback is executed by a new thread, while the original thread keeps waiting for the completion of the initial call.

Exceptions are a mechanism that deals with conditions considered as being outside the normal execution of a service, such as failures, out of range parameter values, etc. When such a condition occurs, execution of the service is cleanly terminated (e.g. resources are released) and control is returned to the caller, with an information on the nature of the exception. Thus an exception may be considered as a "one-way callback". It is the responsibility of the requester of the service to provide a handler for all possible exceptions.

The notion of a callback can be extended one step further. The service provided by B to A may be requested from an outside source, with A still providing one or several callbacks to B. This interaction pattern (Figure 2.2b) is called *inversion of control*, because the flow of control is from B (the provider) to A (the requester). It typically occurs when B is "controlling" A, i.e. providing administrative services such as monitoring or persistent saving; in this situation, the request for service originates from the outside, e.g. is triggered by an external event such as a timing signal.

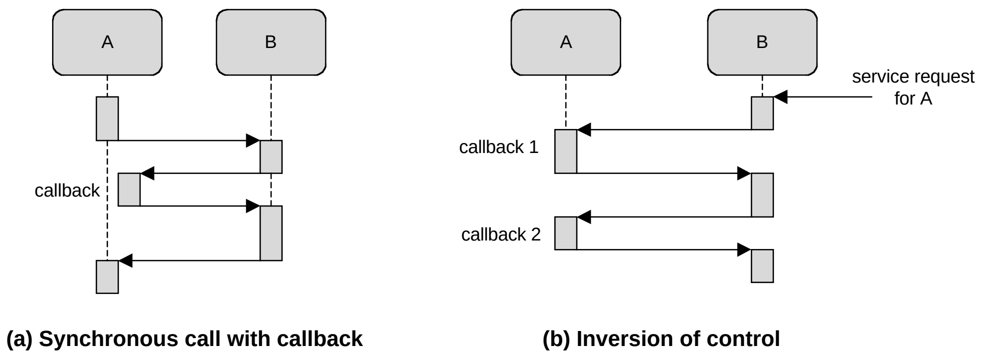


Figure 2.2. Inversion of control

This use of callbacks is further discussed in Chapter 7.

The above interactions do not explicitly imply a notion of time other than event ordering. Continuous media, such as multimedia data, need a form of real-time synchronization. Multimedia data are exchanged through *data streams*, which allow continuous transmission of a sequence of data subject to timing constraints. This form of communication is examined in Chapter 12.

2.1.2 Interfaces

An elementary service provided by a software component is defined by an *interface*, which is a concrete description of the interaction between the requester and the provider of the service. A complex service may be defined by several interfaces, each of which represents a particular aspect of the service. There are actually two complementary views of an interface.

- the usage view: an interface defines the operations and data structures to be used for the provision of a service;
- the contract view: an interface defines a contract between the requester and the provider of a service.

The actual definition of an interface therefore requires a concrete representation for both views, e.g. a programming language for the usage view and a specification language for the contract view.

Recall that both the usage view and the contract view involve two parties[3]: the requester and the provider. As a consequence, the provision of a service actually involves *two* interfaces: the interface provided by the component that delivers a service, and the interface expected by the customer of the service. The provided (or server) interface should be "conformant" (i.e. compatible) with the required (or client) interface; we shall come back to the definition of conformance.

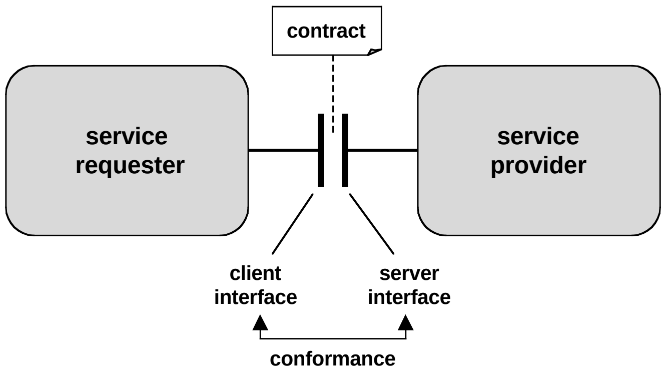


Figure 2.3. Interfaces

The concrete representation of an interface, be it provided or expected, consists of a set of operations, which may take a variety of forms, corresponding to the interaction patterns described in 2.1.1.

- synchronous procedure or method call, with parameters and return value;
- access to an attribute, i.e. a data structure (this can be converted into the previous form by means of "getter" or "setter" functions on the elements of the data structure);
- asynchronous procedure call;

[3]Some forms of service involve more than two parties, e.g. one provider with multiple requesters, etc. It is always possible to describe such situations by one to one relationships, e.g. by defining virtual interfaces that multiplex actual interfaces, etc.

- event source or sink;
- data stream provider (output channel) or receiver (input channel);

The concrete representation of contracts is examined in 2.1.3.

A number of notations, known as Interface Description Languages (IDL), have been designed to formally describe interfaces. There is currently no single common model of an IDL, but the syntax of most existing IDLs is inspired by that of a procedural programming language. Some programming languages (e.g. Java, C#) actually include the notion of an interface and therefore define their own IDL. A typical interface definition specifies the signature of each operation, i.e. its name, the type and mode of transmission of its parameters and return values, and the exceptions it may raise during execution (the requester is expected to provide handlers for these exceptions).

The representation of an interface, together with the associated contract, completely defines the interaction between the requester and the provider of the service that the interface represents. Therefore neither the requester nor the provider should make any assumption on the other party, beyond the information explicitly specified in the interface. In other words, anything beyond the client or server interface is seen by the other party as a "black box". This rule is known as the *encapsulation principle*, which is a special instance of separation of concerns. The encapsulation principle ensures independence between interface and implementation, and allows a system to be modified by "plug and play", replacing a part by a different one provided the interfaces between the replaced part and the rest of the system remain compatible.

2.1.3 Contracts and Interface Conformance

The contract between the provider and the customer of a service may take a variety of forms, depending on the specified properties and on the more or less formal expression of the specification. For instance, the term *Service Level Agreement* (SLA) is used for a legal contract between the provider and the customer of a global, high level service (e.g. between an Internet Service Provider (ISP) and its clients).

From a technical point of view, different kinds of properties can be specified. Following [Beugnard et al. 1999], we may distinguish four levels of contracts.

- Level 1 applies to the form of the operations, usually by defining *types* for the operations and parameters. This part of the contract can be statically verified.
- Level 2 applies to the dynamic behavior of the operations of the interface, by specifying the semantics of each operation.
- Level 3 applies to the dynamic interactions between the operations of an interface, by specifying synchronization constraints between the execution of these operations. If the service is composed of several interfaces, there also may exist constraints between the execution of operations belonging to different interfaces.
- Level 4 applies to the extra-functional properties, i.e. those that do not explicitly appear in the interfaces. The term "Quality of Service" (QoS) is also used for these properties, which include performance, security, availability, etc.

Note again that the contract goes both ways, at all levels, i.e. it constrains the requester as well as the provider. For example, the parameters passed to a function call are constrained by their type; if the interface involves a callback, the callback procedure must be provided (this amounts to specifying a procedure-valued parameter).

The essence of an interface's contract is expressed by the notion of conformance. An interface *I2* is said to *conform* to an interface *I1* if a component that implements all methods specified in *I2* may be used anywhere a component that implements all the methods specified in *I1* may be used. In other words, *I2* conforms to *I1* if *I2* satisfies *I1*'s contract.

Conformance may be checked at each of the four above-specified levels. We examine them in turn.

Syntactic Contracts

A syntactic contract is based on the form of the operations. A common way of expressing such a contract is by using types. A *type* defines a predicate that applies to objects[4] of that type. The type of an object X is noted $T(X)$. The notion of conformance is expressed by *subtyping*: if *T2* is a subtype of *T1* (noted $T2 \sqsubseteq T1$), any object of type *T2* is also an object of type *T1* (in other words, an object of type *T2* may be used anywhere an object of type *T1* is expected). The subtyping relationship thus defined is called *true* (or conformant) subtyping.

Let us consider interfaces defined as a set of procedures. For such interfaces, conformant subtyping is defined as follows: an interface *I2* is a subtype of an interface of type *I1* (noted $T(I2) \sqsubseteq T(I1)$) if *I2* has at least the same number of procedures as *I1* (it may have more), and if for each procedure defined in *I1* there is a conformant procedure in *I2*. A procedure *Proc2* is said to be conformant with a procedure *Proc1* when the following relationships hold between the signatures of these procedure.

- *Proc1* and *Proc2* have the same number of parameters and return values (declared exceptions are considered as return values).
- For each return value *R1* of *Proc1*, there is a matching return value *R2* of *Proc2* such that $T(R2) \sqsubseteq T(R1)$ (this is called a *covariant* relationship).
- For each entry parameter *X1* of *Proc1*, there is a matching entry parameter *X2* of *Proc2* such that $T(X1) \sqsubseteq T(X2)$ (this is called a *contravariant* relationship).

These rules illustrate a general principle of substitutability: an entity *E2* may be substituted for another entity *E1* if *E2* "provides more and requires less" than *E1*. Here "provides" and "requires" must be adapted to each specific situation (e.g. in a procedure call, the entry parameters are "required" and the result is "provided"). Also "more" and "less" respectively refer to the notions of subtype and supertype, and include equality.

Note that the subtyping relationship defined in most programming languages usually fails to satisfy parameter type contravariance and is therefore not a true subtyping re-

[4]Here the term *object* designates any entity of interest in the present context, e.g. a variable, a procedure, an interface, a component.

lationship. In such case (e.g. in Java), some conformance errors may not be statically detected, and must be caught by a run time check.

The notion of conformance may be extended to other forms of interface definitions, e.g. those containing event sources or sinks, or data streams. Examples are found in Chapters 6, 7, and 12.

Recall that the relationship between types is purely syntactic and does not catch the semantics of conformance. Verifying semantics is the goal of behavioral contracts.

Behavioral Contracts

Behavioral contracts are based on a method proposed in [Hoare 1969] to prove properties of programs, using pre- and post-conditions together with proof rules based on first-order logic. Let A be a sequential action. Then the notation

$$\{P\}\ A\ \{Q\},$$

in which P and Q are assertions (predicates on the state of the program's universe), says the following: if the execution of A is started in a state in which P holds, and if A terminates, then Q holds at the end of A. An additional condition may be specified in the form of an invariant predicate I that should be preserved by the execution of A. Thus if P and I hold initially, Q and I hold at the end of A, if A terminates. The invariant may be used to specify a consistency constraint.

This may be transposed as follows in terms of services and contracts. Before the execution of a service, it is the responsibility of the requester to ensure that the pre-condition P and the invariant I actually hold. It is the responsibility of the provider of the service to ensure that the service is actually delivered in finite time, and that the post-condition Q and the invariant I hold at the end. Possible cases of abnormal termination must be specified in the contract and handled by retrying or by exception raising. This method has been developed under the name of "design by contract" [Meyer 1992] through extensions to the Eiffel language allowing the expression of pre- and post-conditions and invariant predicates. These conditions are checked at run time. Similar tools have been developed for Java [Kramer 1998].

The notion of subtyping may be extended to behavioral contracts, by specifying the conformance constraints for assertions. Consider a procedure *Proc1* defined in interface *I1*, and the corresponding (conformant) procedure *Proc2* defined in interface *I2*, such that $T(I2) \sqsubseteq T(I1)$. Let *P1* and *Q1* (resp. *P2* and *Q2*) be the pre- and post-condition defined for *Proc1* (resp. *Proc2*). The following conditions must hold:

$$P1 \Rightarrow P2 \text{ and } Q2 \Rightarrow Q1$$

In other words, a subtype has weaker pre-conditions and stronger post-conditions than its supertype, which again illustrates the substitutability principle.

Synchronization Contracts

The expression of program correctness by means of assertions may be extended to concurrent programs. The goal here is to separate, as much as possible, the description of synchronization constraints from the code of the procedures. An early proposal is path expressions [Campbell and Habermann 1974], which specify constraints on the ordering and on the concurrency of procedure executions. Further developments (synchronization counters, synchronization policies) were essentially extensions and improvements of this construct, whose implementation relies on run time mechanisms generated from the static constraint description. Several articles describing proposals in this area may be found in [CACM 1993], but these techniques have not found a wide application.

A very simple form of synchronization contract is the `synchronized` clause of Java, which specifies execution in mutual exclusion. Another example includes the selection of a queue management policy (e.g. FIFO, priority, etc.) for a shared resource, among a predefined set.

Current efforts aim at allowing compile-time checking of the synchronization constraints, in order to detect incompatibilities at an early stage. An example of recent work in this area is [Chakrabarti et al. 2002].

Quality of Service Contracts

The specifications associated with the interface of a system or part of a system, be they or not expressed in a formal way, are called *functional*. A system may be subject to additional specifications, which apply to some of its aspects that do not explicitly appear in the interface. Such specifications are called *extra-functional*[5]. Quality of Service (another name for these properties) includes the following aspects.

- *Availability*. The availability of a service is a statistical measure of the fraction of the time during which the service is ready for use. This depends both on the failure rate of (parts of) the system that delivers the service and on the time it takes to restore service after a failure.

- *Performance*. This quality covers several aspects, which are essential for real-time applications (applications whose correctness or usability relies on timing constraints). Some of these aspects are related to communication (bounds on latency, jitter, bandwidth); others apply to processing speed or data access latency.

- *Security*. Security covers properties related to the correct use of a service by its users, according to specified rules of usage. It includes confidentiality, integrity, authentification, and access rights control.

Other extra-functional aspects that are difficult to quantify include maintainability and ease of evolution.

Since most aspects of quality of service depend on a changing environment, it is important that the policies of QoS management should be adjustable. Therefore QoS contracts usually include the possibility of negotiation, i.e. redefining the terms of the contract through run-time exchanges between the requester and the provider of the service.

[5]Note that the definition of a specification as "functional" or "extra-functional" is not absolute, but depends on the current state of the art: an aspect that is extra-functional today may become functional when technical advances allow its expression to be integrated into an interface.

2.2 Architectural Patterns

In this section, we review a few basic principles for structuring middleware systems. Most of the systems examined in this book are organized around these principles, which essentially provide guidelines for decomposing a complex system into parts.

2.2.1 Multilevel Architectures

Layered Architectures

Decomposing a complex system into layers of abstraction is an old and powerful organizational principle. It pervades many areas of system design, through such widely used notions as virtual machines and protocol stacks.

Abstraction is a conceptual process by which a designer builds a simplified view of a system as a set of interfaces. The implementation of these interfaces in terms of more detailed entities is left to a further refinement step. A complex system may thus be described at different levels of abstraction. In the simplest organization (Figure 2.4a), each level i defines its own entities, which provide an interface to the upper level (i+1). These entities are implemented using the interface provided by the lower level (i-1), down to a predefined base level (usually implemented in hardware). This architecture is described in [Buschmann et al. 1995] as the Layers pattern.

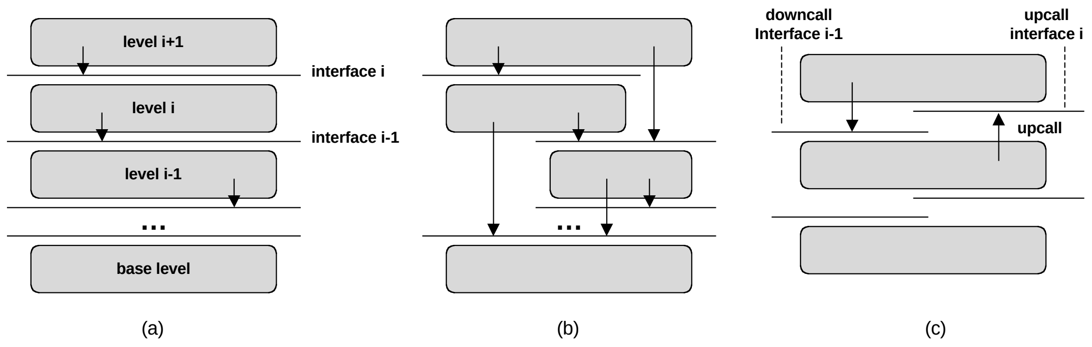


Figure 2.4. Layered system organizations

The interface provided by each level may be viewed as set of functions defining a library, in which case it is often called an Application Programming Interface[6] (API). An alternative view is to consider each level as a virtual machine, whose "language" (i.e. instruction set) is defined by its interface. By virtue of the encapsulation principle, a virtual machine hides the implementation details of all the lower levels. Virtual machines [Smith and Nair 2005] have been used to emulate a computer, an operating system, or a network on top of a different one, to emulate a number of computers in order to multiplex physical resources, or to implement the run time environment of a programming language (e.g. the Java Virtual Machine [Lindholm and Yellin 1996]).

This basic scheme may be extended in several ways. In the first extension (Figure 2.4b), a layer at level i may use (part of) the interfaces provided by the machines at the

[6]a complex interface may also be partitioned into several APIs, each one related to a specific function.

lower layers. In the second extension, a layer at level i may callback the layer at level $i+1$, using a callback interface provided by that layer. In this context, callbacks are known as *upcalls* (referring to the "vertical" layer hierarchy).

Although upcalls may be synchronous, their most frequent use is to propagate asynchronous events up the layer hierarchy. Consider the structure of an operating system kernel. The upper (application) layer activates the kernel through synchronous downcalls, using the system call API. The kernel also activates hardware-provided functions (e.g. updating a MMU, sending a command to a disk drive) through the equivalent of synchronous calls. On the other hand, the hardware typically activates the kernel through asynchronous interrupts (upcalls), which trigger the execution of handlers. This calling structure is often repeated in the upper layers, i.e. each layer receives synchronous calls from the upper layer, and asynchronous calls from the lower layer. This organization is described in [Schmidt et al. 2000] as the Half Sync, Half Async pattern. It is widely used in communication protocols, as described in Chapter 4.

Multitier Architectures

The advent of distributed systems has promoted a different form of multilevel architecture. Consider the historical evolution of a common form of client-server applications, in which a client's requests are processed using information stored in a database.

In the 1970s (Figure 2.5a), both the data management functions and the application itself are executed on a mainframe. The client's terminal is a simple display, which implements a primitive form of user interface.

In the 1980s (Figure 2.5b), workstations are available as client machines, and allow elaborate graphical user interface (GUI) facilities to be implemented. The processing capabilities of the workstation allow it to take up a part of the application processing, thus reducing the load of the server and improving scalability (since the addition of a new client station contributes processing power to the application).

The drawback of this architecture is that the application is now split between the client and the server machines; the communication interface is now internal to the application. Modifying the application may now involve changes both on the client and the server machines, and possibly a change in the communication interface.

These drawbacks are corrected by the architecture shown on Figure 2.5c, introduced in the late 1990s. The functions of the application are split between three machines: the client station only supports the functions of the GUI, the application proper resides on a dedicated server, and the management of the database is devoted to another machine. Each of these "horizontal" divisions is called a *tier*. Further specialization of the functions leads to other multitier architectures. Note that each tier may itself be subject to a "vertical" layered decomposition into abstraction levels.

The multitier architecture still has the benefits of scalability, as the application machine may be incrementally upgraded (e.g. by adding a machine to a cluster). In addition, the interfaces between the tiers may be designed to favor separation of concerns, since logical interfaces now coincide with communication interfaces. For example, the interface between the application tier and the data management tier can be made generic, in order to easily accommodate a new type of database, or to connect to a legacy application, using an adapter (2.3.3) for interface conversion.

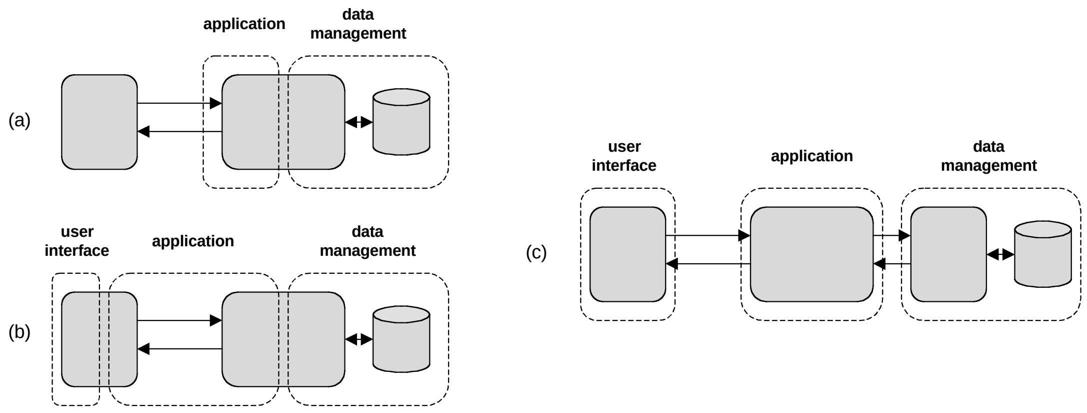


Figure 2.5. Multitier architectures

Examples of multitier architectures are presented in Chapter 7.

Frameworks

A software framework is a program skeleton that may be directly reused, or adapted according to well-defined rules, to solve a family of related problems. This definition covers many cases; here we are interested in a particular form of frameworks that consists of an infrastructure in which software components may be inserted in order to provide specific services. Such frameworks illustrate some notions related to interfaces, callbacks, and inversion of control.

The first example (Figure 2.6a) is the microkernel, an architecture introduced in the 1980s in an attempt to develop flexible operating systems. A microkernel-based operating system consists of two layers.

- The microkernel proper, which manages the hardware resources (processor, memory, I/O, network communication), and provides an abstract resource management API to the upper level.
- The kernel, which implements a specific operating system (a "personality") using the API of the microkernel.

In most microkernel-based organizations, an operating system kernel is structured as a set of *servers*, each of which is in charge of a specific function (e.g. process management, file system, etc.). A typical system call issued by an application is processed as follows.

- The kernel analyzes the call and downcalls the microkernel using the appropriate function of its API.
- The microkernel upcalls a server in the kernel. Upon return, the microkernel may interact with the hardware; this sequence may be iterated, e.g. if more than one server is involved.

- The microkernel returns to the kernel, which completes the work and returns to the application.

Adding a new function to a kernel is done by developing and integrating a new server.

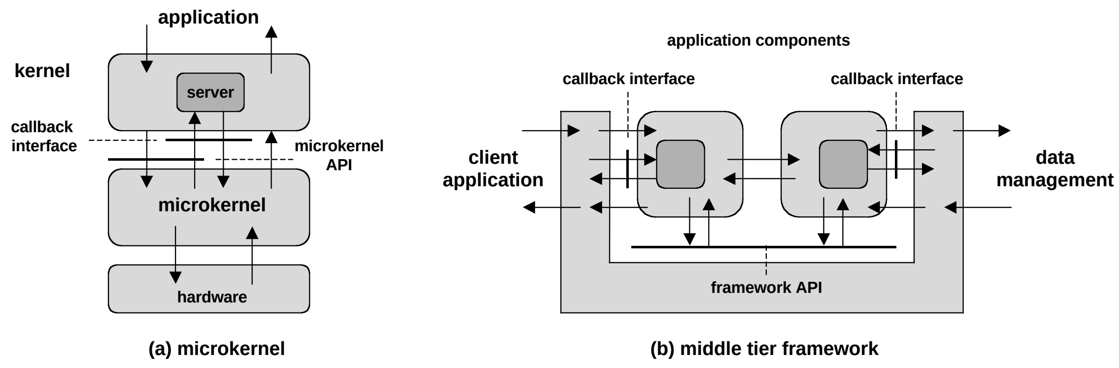


Figure 2.6. Framework architectures

The second example (Figure 2.6b) illustrates a typical organization of the middle tier of a 3-tier client-server architecture. The middle tier framework interacts with both the client and the data management tiers, and mediates the interaction between these tiers and the server application program. This program is made up of application components, which use the API provided by the framework and must supply a set of callback interfaces. Thus a client request is handled by the framework, which activates an appropriate application component, interacts with it using its own API and the component's callback interface, and finally returns to the client.

Detailed examples of this organization are presented in Chapter 7.

Both above examples illustrate inversion of control. To provide its services, the framework uses callbacks to externally supplied software modules (servers in the microkernel example, or application components in the middle tier example). These modules must respect the framework contract, by providing a specified callback interface, and by using the framework API.

The layered and multitier organizations define a large grain structure for a complex system. Each layer or tier (or layer in a tier) is itself organized using finer grain entities. Objects, a common way of defining this fine grain structure, are presented in the next section.

2.2.2 Distributed Objects

Objects in Programming

Objects have been introduced in the 1960s as a means of structuring computing systems. While there are many definitions for objects, the following properties capture the most common object-related concepts, especially in the context of distributed computing.

An *object*, in a programming model, is a software representation of a real-world entity (such as a person, a bank account, a document, a car, etc.). An object is the association of a state and of a set of procedures (or methods) that operate on that state. The object model that we consider has the following properties.

- *Encapsulation*. An object has an interface, which comprises a set of methods (procedures) and attributes (values that may be read and written). The only way of accessing an object (consulting or changing its state) is through its interface. No part of the state is visible from outside the object, other than those explicitly present in the interface, and the user of an object should not rely on any assumption about its implementation. The type of an object is defined by its interface.

  As explained in 2.1.2, encapsulation achieves independence between interface and implementation. The interface acts as a contract between the user and the implementer of an object. Changing the implementation of an object is invisible to its users, as long as the interface is preserved.

- *Classes and instances*. A *class* is a generic description that is common to a set of objects (the instances of the class). The instances of a class have the same interface (hence the same type), and their state has the same structure; they differ by the value of that state. Each instance is identified as a distinct entity. Instances of a class are dynamically created, through an operation called *instantiation*; they may also be dynamically deleted, either explicitly or automatically (by garbage collection) depending on the specific implementation of the object model.

- *Inheritance*. A class may be derived from another class by specialization, i.e. by defining additional methods and/or additional attributes, or by redefining (overloading) existing methods. The derived class is said to *extend* the initial class (or base class) or to *inherit* from it. Some models also allow a class to inherit from more than one class (multiple inheritance).

- *Polymorphism*. Polymorphism is the ability, for a method, to accept parameters of different types and to have a different behavior for each of these types. Thus an object may be replaced, as a parameter of a method, by a "compatible" object. The notion of compatibility, or conformance (2.1.3) is expressed by a relationship between types, which depends on the specific programming model or language being used.

Recall that these definitions are not universal, and are not applicable to all object models (e.g. there are other mechanisms than classes to create instances, objects may be active, etc.), but they are representative of a vast set of models used in current practice, and are embodied in such languages as Smalltalk, C++, Eiffel, Java, or C#.

Remote Objects

The above properties make objects specially well suited as a structuring mechanism for distributed systems.

- Heterogeneity is a dominant feature of these systems. Encapsulation is a powerful tool in a heterogeneous environment: the user of an object only needs to know an interface for that object, which may have different implementations on different locations.

- Dynamic creation of object instances allows different objects to be created with the same interface, possibly at different remote locations; of course middleware must again provide a mechanism for remote object creation, in the form of factories (2.3.2).
- Inheritance is a mechanism for reuse, as it allows a new interface to be defined in terms of an existing one. As such, it is useful for distributed applications developers, who are confronted with a changing environment and have to define new classes to deal with new situations. In order to use inheritance, a generic (base) class is first designed to capture a set of object features that are common to a wide range of expected situations. Specific, more specialized, classes are then defined by extending the base class. For example, an interface for a color video stream may be defined as an extension of that of a (generic) video stream. An application that uses video stream objects also accepts color video streams, since these objects implement the video stream interface (this is an instance of polymorphism).

The simplest and most common way of distributing objects is to allow the objects that make up an application to to be located on distributed sites (other ways of distributing objects are described in Chapter 5). A client application may use an object located on a remote site by calling a method of the object's interface, as if the object were local. Objects used in this way are called *remote objects*, and a method call on a remote object is called Remote Method Invocation; it is a transposition of RPC to the world of objects.

Remote objects are an example of a client-server system. Since a client may use several different objects located on a remote site, different words are used to designate the remote site (the *server* site) and an individual object that provides a specific service (a *servant* object). To make the system work, an appropriate middleware must locate an implementation of the servant object on a possibly remote site, send the parameters to the object's location, actually perform the call, and return the results to the caller. A middleware that performs these tasks is an Object Request Broker, or ORB.

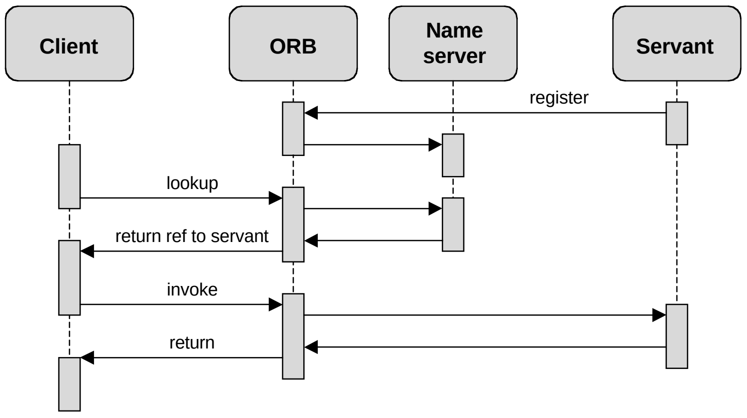


Figure 2.7. Remote Method Invocation

The overall structure of a call to a remote object (Figure 2.7) is similar to that of an RPC: the remote object must first be located, which is usually done by means of a name server or trader (Chapter 3); then the call itself is performed. Both the lookup and the invocation are mediated through the ORB. The internal organization of an ORB is examined in detail in Chapter 5.

2.3 Patterns for Distributed Object Middleware

Remote execution mechanisms rely on a few design patterns, which have been widely described in the literature, specially in [Gamma et al. 1994], [Buschmann et al. 1995], and [Schmidt et al. 2000]. In this presentation, we concentrate on the specific use of these patterns for distributed object middleware, and we discuss their similarities and differences. For an in-depth discussion of these patterns, the reader is directed to the specified references.

2.3.1 Proxy

The Proxy pattern is one of the first design patterns identified in distributed programming [Shapiro 1986]. While its application domain has been extended to many other aspects [Buschmann et al. 1995], we only discuss here the use of Proxy for distributed objects.

1. **Context**. This pattern is used for applications organized as a set of objects in a distributed environment, communicating through remote method invocation: a client requests a service provided by some possibly remote object (the servant).

2. **Problem**. Define an access mechanism that does not involve hard-coding the location of the servant into the client code, and does not necessitate deep knowledge of the communication protocols by the client

3. **Desirable Properties**. Access should be efficient at run time. Programming should be simple for the client; ideally there should be no difference between local and remote access (this property is known as access transparency).

4. **Constraints**. The main constraint results from the distributed environment: the client and the server are in different address spaces.

5. **Solution**. Use a local representative of the server on the client site. This representative has exactly the same interface as the servant. All information related to the communication system and to the location of the servant is hidden in the proxy, and thus invisible to the client.

   The organization of the proxy is shown on Figure 2.8.

   The internal structure of the proxy follows a well-defined pattern, which facilitates its automatic generation.

   - a pre-processing phase, which essentially consists of marshalling the parameters and preparing the request message.
   - the actual invocation of the servant, using the underlying communication protocol to send the request and to receive the reply.

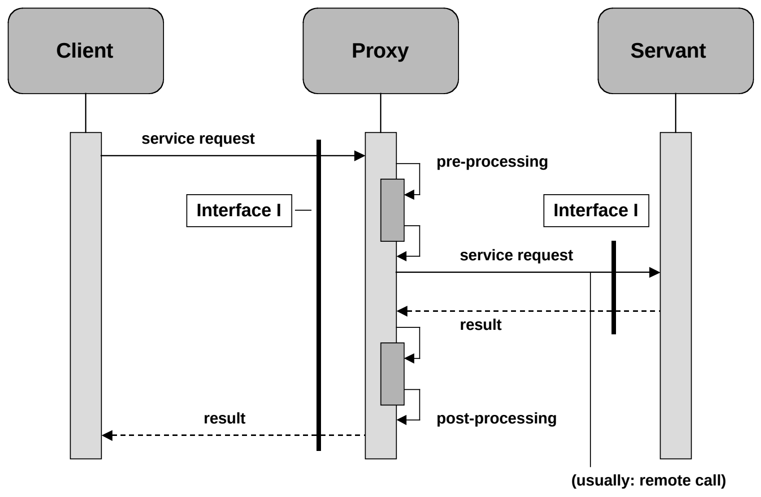


Figure 2.8. Proxy

- a post-processing phase, which essentially consists of unmarshalling the return values.

6. **Known Uses**.

   In middleware construction, proxies are used as local representatives for remote objects. They do not add any functionality. Examples may be found in Chapter 5.

   Some variants of proxies contain additional functions. Examples are client-side caching and client-side adaptation. In this latter case, the proxy may filter server output to adapt it to specific client display capabilities, such as low resolution. Such "smart" proxies actually combine the standard functions of a proxy with those of an interceptor (2.3.4).

7. **References**.

   Discussions of the PROXY pattern may be found in [Gamma et al. 1994], [Buschmann et al. 1995].

2.3.2 Factory

1. **Context**. Applications organized as a set of objects in a distributed environment (the notion of "object" in this context may be quite general, and is not limited to objects as defined in object-oriented languages).

2. **Problem**. Dynamically create families of related objects (e.g. instances of a class), while allowing some decisions to be deferred to run time (e.g. choosing a concrete class to implement a given interface).

3. **Desirable Properties**. The implementation details of the created objects should be abstracted away. The creation process should allow parameters. Evolution of the mechanism should be easy (no hard-coded decisions).

4. **Constraints**. The main constraint results from the distributed environment: the client (requesting object creation) and the server (actually performing creation) are in different address spaces.

5. **Solution**. Use two related patterns: an ABSTRACT FACTORY defines a generic interface and organization for creating objects; the actual creation is deferred to concrete factories. ABSTRACT FACTORY may be implemented using FACTORY METHODS (a creation method that is redefined in a subclass).

   Another way of achieving flexibility is to use a Factory Factory, as shown on Figure 2.9 (the creation mechanism itself is parameterized).

   A Factory may also be used as a manager of the objects that it has created, and may thus implement a method to look up an object (returning a reference for it), and to remove an object upon request.

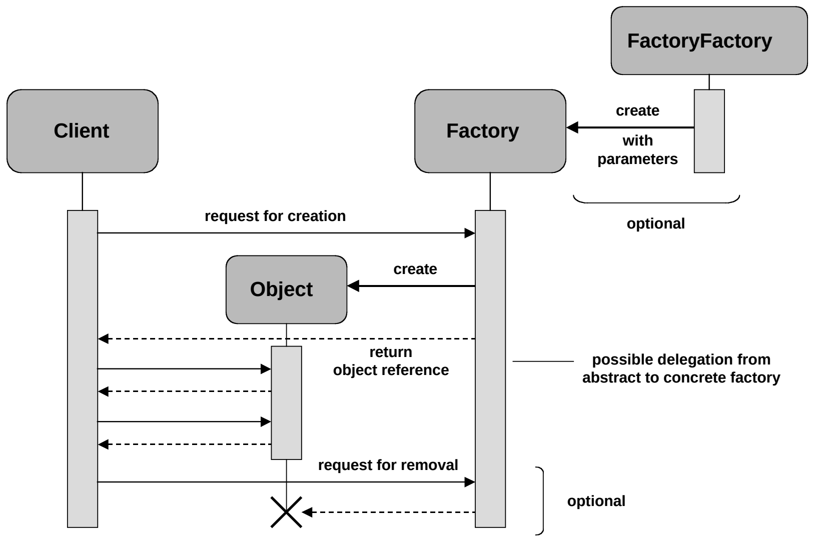


Figure 2.9. Factory

6. **Known Uses**.

   FACTORY is one of the most widely used patterns in middleware. It is both used in applications (to create remote instances of application objects) and within middleware itself (one example is binding factories, described in Chapter 3). Factories are also used in relation to components (Chapter 7).

7. **References**.

   The two patterns ABSTRACT FACTORY and FACTORY METHOD are described in [Gamma et al. 1994].

2.3.3 Adapter

1. **Context**. Service provision, in a distributed environment: a service is defined by an interface; clients request services; servants, located on remote servers, provide services.

2. **Problem**. Reuse an existing servant by providing a different interface for its functions in order to comply to the interface expected by a client (or class of clients).

3. **Desirable Properties**. The interface conversion mechanism should be run-time efficient. It should also be easily adaptable, in order to respond to unanticipated changes in the requirements (e.g. the need to reuse a new class of applications). It should be reusable (i.e. generic).

4. **Constraints**. No specific constraints.

5. **Solution**. Provide a component (the adapter, or wrapper) that screens the servant by intercepting method calls to its interface. Each call is prefixed by a prologue and followed by an epilogue in the adapter (Figure 2.10). The parameters and results may need to be converted.

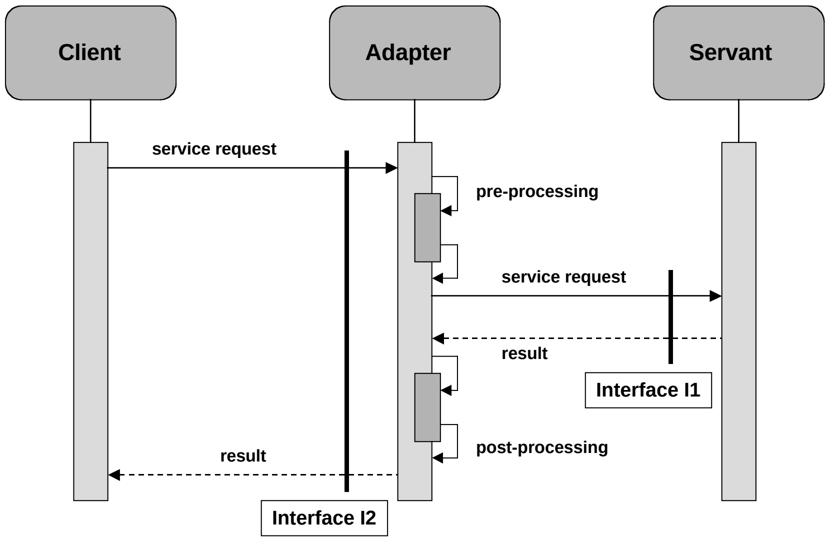


Figure 2.10. Adapter

In some simple cases, an adapter can be automatically generated from the description of the provided and required interfaces.

6. **Known Uses**.

Adapters are widely used in middleware to encapsulate server-side functions. Examples include the Portable Object Adapter (POA) of CORBA (Chapter 5), and the various adapters for reusing legacy systems, such as the Java Connector Architecture (JCA).

7. **References**.

   Adapter (also known as Wrapper) is described in [Gamma et al. 1994]. A related pattern is Wrapper Façade ([Schmidt et al. 2000]), which provides a high-level (e.g. object-oriented) interface to low level functions.

2.3.4 Interceptor

1. **Context**. Service provision, in a distributed environment: a service is defined by an interface; clients request services; servants, located on remote servers, provide services. There is no restriction on the form of communication (e.g. uni- or bi-directional, synchronous or asynchronous).

2. **Problem**. One wants to enhance an existing service with new capabilities, or to provide it by different means.

3. **Desirable Properties**. The mechanism should be generic (applicable to a wide variety of situations). It should allow static (compile time) or dynamic (run time) service enhancement.

4. **Constraints**. Services may be added or removed dynamically.

5. **Solution**. Create interposition objects (statically or dynamically). These objects intercept calls (and/or returns) and insert specific processing, that may be based on contents analysis. An interceptor may also redirect a call to a different target.

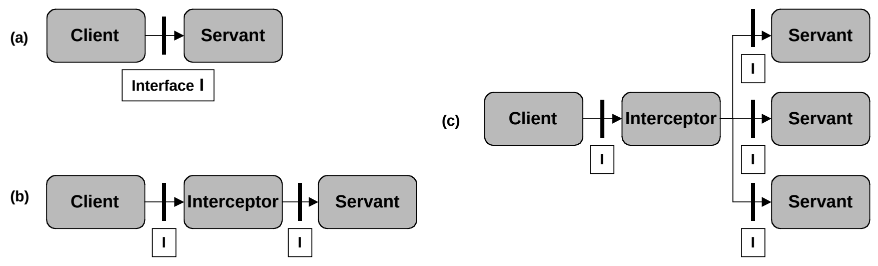


Figure 2.11. Simple forms of Interceptor

This mechanism may be implemented in a variety of forms. In the simplest form, an interceptor is a module that is inserted at a specified point in the call path between the requester and the provider of a service (Figure 2.11a and 2.11b). It may also be used as a switch between several servants that may provide the same service with different enhancements (Figure 2.11c), e.g. provision for fault tolerance, load balancing or caching.

In a more general form (Figure 2.12, interceptors and service providers (servants) are managed by a common infrastructure and created upon request. The interceptor uses the servant interface and may also rely on services provided by the infrastructure. The servant may provide callback functions to be used by the interceptor.

6. **Known Uses**.

   Interceptors are used in a variety of situations in middleware systems.

   - to enhance existing applications or systems with new capabilities. An early example is the subcontract mechanism [Hamilton et al. 1993]. The CORBA Portable Interceptors (further described in Chapter 5) provide a systematic way to extend the functionality of the Object Request Broker by inserting interception modules at predefined points in the call path. Other uses include the support of fault tolerance mechanisms (e.g. providing support for object groups), as described in Chapter 11.
   - to select a specific implementation of a servant at run time.
   - to implement frameworks for component-based applications (see Chapter 7).
   - to implement reflective middleware (see 2.4.1 and 2.4.3).

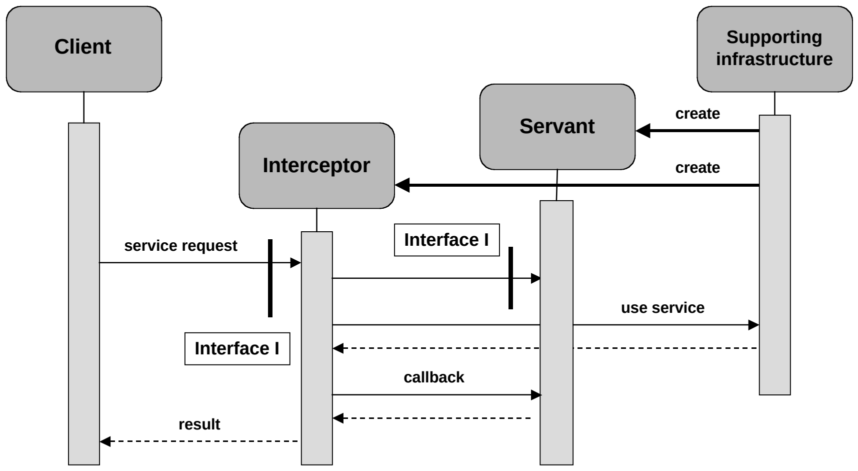


Figure 2.12. General Interceptor

7. **References**.

   The Interceptor pattern is described in [Schmidt et al. 2000].

2.3.5 Comparing and Combining Patterns

Three of the patterns described in the previous section (Proxy, Adapter, and Interceptor) have close relationships to each other. They all involve a software module being inserted between the requester and the provider of a service. We briefly discuss their similarities and differences.

- Adapter *vs* Proxy. Adapter and Proxy have a similar structure. Proxy preserves the interface, while Adapter transforms the interface. In addition, Proxy often (not always) involves remote access, while Adapter is usually on-site.

- ADAPTER *vs* INTERCEPTOR. ADAPTER and INTERCEPTOR have a similar function: both modify an existing service. The main difference is that ADAPTER transforms the interface, while INTERCEPTOR transforms the functionality (actually INTERCEPTOR may completely screen the initial target, replacing it by a different servant).
- PROXY *vs* INTERCEPTOR. A PROXY may be seen as a special form of an INTERCEPTOR, whose function is restricted to forwarding a request to a remote servant, performing the data transformations needed for transmission, and abstracting away the communication protocol. Actually, as mentioned in 2.3.1, a proxy may be combined with an interceptor, making it "smart" (i.e. providing new functionalities in addition to request forwarding, but leaving the interface unchanged).

Using the above patterns, we may draw a first approximate and incomplete picture of the overall organization of an ORB (Figure 2.13).

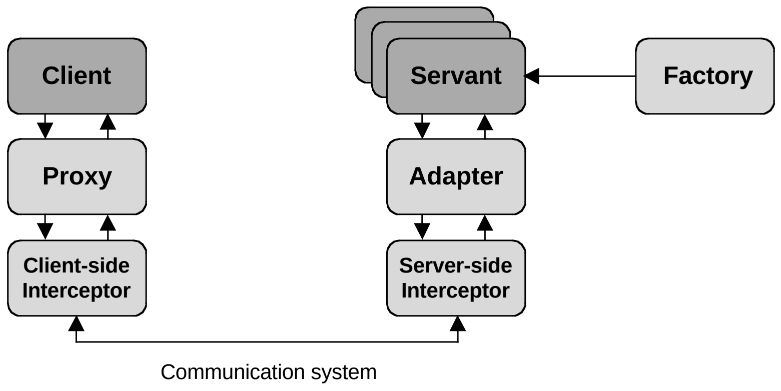


Figure 2.13. Using patterns in an ORB

The main missing aspects are those related to binding and communication, which are described in Chapters 3 and 4, respectively.

2.4 Achieving Adaptability and Separation of Concerns

Three main approaches are being used to achieve adaptability and separation of concerns in middleware systems: meta-object protocols, aspect-oriented programming, and pragmatic approaches. They are summarized in the following subsections.

2.4.1 Meta-Object Protocols

Reflection has been introduced in 1.4.2. Recall that a reflective system is one that is able to answer questions about itself and to modify its own behavior, by providing a causally connected representation of itself.

Reflection is a desirable property for middleware, because a middleware system operates in a changing environment and needs to adapt its behavior to changing requirements.

Reflective capabilities are present in most existing middleware systems, but they are usually introduced locally, for isolated features. Middleware platforms that integrate reflection in their basic architecture are being developed as research prototypes [RM 2000].

A general approach to designing a reflective system is to organize it into two levels.

- The *base level*, which provides the functionalities defined by the system's specifications.
- The *meta-level*, which uses a representation of the entities of the base level in order to observe or modify the behavior of the base level.

This decomposition may be iterated, by considering the meta-level as a base level for a meta-meta-level, and so on, thus defining a so-called "reflective tower". In most practical cases, the height of the tower is limited to two or three levels.

Defining a representation of the base level, to be used by the meta-level, is a process called *reification*. It results in the definition of meta-objects, each of which is a representation, at the meta-level, of a data structure or operation defined at the base level. The operation of the meta-objects, and their relationship to the base level entities, are specified by a *meta-object protocol* (MOP) [Kiczales et al. 1991].

A simple example of a MOP (borrowed from [Bruneton 2001]) is the reification of a method call in a reflective object-oriented system. At the meta-level, a meta-object `Meta_Obj` is associated with each object `Obj`. A method call `Obj.meth(params)` is executed in the following steps (Figure 2.14).

1. The method call is reified into an object `m`, which contains a representation of `meth` and `params`. The precise form of this representation is defined by the MOP. This object `m` is transmitted to the meta-object, which executes `Meta_Obj.meta_MethodCall(m)`.

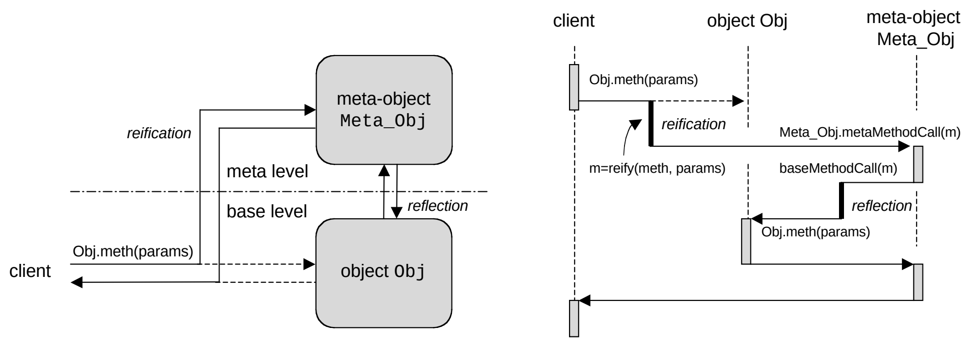


Figure 2.14. Performing a method call in a reflective system

2. The method `meta_MethodCall(m)` then executes any processing specified by the MOP. To take simple examples, it may print the name of the method (by calling a method such as `m.methName.printName()`) before actually executing it (for tracing) or it may save the state of the object prior to the method call (to allow undo operations), or it may check the value of the parameters, etc.

3. The meta-object may now actually execute the initial call[7], by invoking a method `baseMethodCall(m)` which essentially performs `Obj.meth(params)`[8]. This step (the inverse of reification) is called *reflection*.

4. The meta-object then executes any post-processing defined by the MOP, and returns to the initial caller.

Likewise, the operation of object creation may be reified by calling a meta-object factory (at the meta-level). This factory creates a base-level object, using the base-level factory; the new object then upcalls the meta-object factory, which creates the associated meta-object, and executes any additional operations specified by the MOP (Figure 2.15).

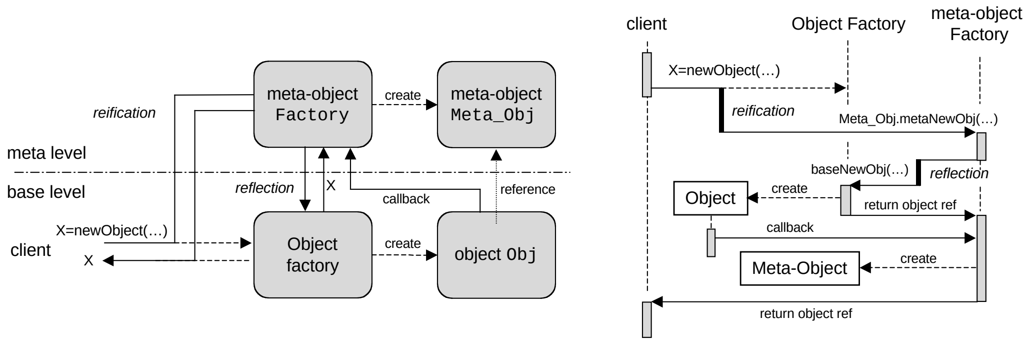


Figure 2.15. Object creation in a reflective system

Examples of using meta-object protocols in middleware may be found in Chapters 5 and 11.

2.4.2 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [Kiczales 1996] is motivated by the following remarks.

- Many different concerns (or "aspects") are usually present within an application (common examples include security, persistence, fault-tolerance, and other extra-functional properties).
- The code related to these concerns is usually tightly intermixed with the "functional" application code, which makes changes and additions difficult and error prone.

The goal of AOP is to define methods and tools to better identify and isolate the code related to the various aspects present in an application. More precisely, an application developed using AOP is built in two phases.

[7]it does not *have to* execute the initial call; for example, if the MOP is used for protection, it may well decide that the call should not be executed, and return to the caller with a protection violation message.

[8]note that it is not possible to directly invoke `Obj.meth(params)` because only the reified form of the method call is available to the meta-object and also because a post-processing step may be needed.

- The main part of the application (the base program), and the parts that deal with different additional aspects are written independently, possibly using specialized languages for the aspect code.
- All these pieces are integrated to form the global application, using a composition tool (aspect weaver).

A *join point* is a place, in the source code of the base program, where aspect-related code can be inserted. Aspect weaving relies on two main notions: *point cut*, i.e. the specification of a set of join points according to a given criterion, and *advice*, i.e. the definition of the interaction of the inserted code with the base code. For example, if AOP is added to an object-oriented language, a particular point cut may be defined as the set of invocation points of a family of methods (specified by a regular expression), or the set of invocations of a specified constructor, etc. An advice specifies whether the inserted code should be executed before, after, or in replacement for the operations located at the point cuts (in the latter case, these operations may still be called from within the inserted code). Composition may be done statically (at compile time), dynamically (at run time), or using a combination of static and dynamic techniques.

One important problem with AOP is the composition of aspects. For instance, if different pieces of aspect-related code are inserted at the same join point, the order of insertion may be relevant if the corresponding aspects are not independent. Such issues cannot usually be settled by the weaver and call for additional specification.

Two examples of tools that implement AOP are AspectJ [Kiczales et al. 2001] and JAC [Pawlak et al. 2001]. Both apply to base programs written in Java.

AspectJ

AspectJ allows aspects to be defined by specifying pointcuts and advices, in a Java-like notation. A weaver integrates the aspects and the base program into Java source code, which may then be compiled.

A simple example gives an idea of the capabilities of AspectJ. The following code describes an aspect, in the form of pointcut definition and advice.

```
public aspect MethodWrapping{

/* point cut definition */
    pointcut Wrappable(): call(public * MyClass.*(..));

/* advice definition    */
    around(): Wrappable() {
        /* prelude: a sequence of code to be inserted before the call */
           proceed (); /* performs the call to the original method    */
        /* postlude: a sequence of code to be inserted after the call */
    }
}
```

The first part of the description defines a point cut as the set of invocations of any public method of class `MyClass`. The advice part says that a call to such a method should be replaced by a specified prelude, followed by a call to the original method, followed by a specified postlude. In effect, this amounts to placing a simple wrapper (without interface modification) around each method call specified in the pointcut definition. This may be used to add logging facilities to an existing application, or to insert testing code to evaluate pre- and post-conditions for implementing design by contract (2.1.3).

Another capability of AspectJ is *introduction*, which allows additional declarations and methods to be inserted at specified places in an existing class or interface. This facility should be used with care, since it may break the encapsulation principle.

JAC

JAC (Java Aspect Components) has similar goals to AspectJ. It allows additional capabilities (method wrapping, introduction) to be added to an existing application. JAC differs from AspectJ on the following points.

- JAC is not a language extension, but a framework that may be used at run time. Thus aspects may be dynamically added to a running application. JAC uses bytecode modification, and the code of the application classes are modified at class loading time.
- The point cuts and the advices are defined separately. The binding between point cuts and advices is delayed till the weaving phase; it relies on information provided in a separate configuration file. Aspect composition is defined by a meta-object protocol.

Thus JAC provides added flexibility, but at the expense of a higher runtime overhead due to the dynamic weaving of the aspects into the bytecode.

2.4.3 Pragmatic Approaches

Pragmatic approaches to reflection in middleware borrow features from the above systematic approaches, but tend to apply them in an ad hoc fashion, essentially for efficiency reasons. These approaches are essentially based on interception.

Many middleware systems involve an invocation path from a client to a remote server, traversing several layers (application, middleware, operating system, communication protocols). Interceptors may be inserted at various points of this path, e.g. at the send and receive operations of requests and replies.

Inserting interceptors allows non-intrusive extension of middleware functionality, without modifying the application code or the middleware itself. This technique may be considered as an ad hoc way of implementing AOP: the insertion points are the join points and the interceptors directly implement aspects. By adequately specifying the insertion points for a given class of middleware, conforming to a specific standard (e.g. CORBA, EJB), the interceptors can be made generic and may be reused with different implementations of the standard. The functions that may be added or modified through interceptors include monitoring, logging and measurement, security, caching, load balancing, replication. A detailed example of using interceptors in CORBA may be found in Chapter 5.

This technique may also be combined with a meta-object protocol, i.e. the interceptors may be inserted in the reified section of the invocation path (i.e. within a meta-level).

Interception techniques entail a run time overhead. This may be alleviated by using *code injection*, i.e. directly integrating the code of the interceptor into the code of the client or the server (this is the analog of inlining the code of procedures in an optimizing compiler). To be efficient, this injection must be done at a low level, i.e. in assembly code, or (for Java) at the bytecode level, using bytecode manipulation tools such as BCEL [BCEL], Javassist [Tatsubori et al. 2001], or ASM [ASM]. To maintain flexibility, it should be possible to revert the code injection process by going back to the separate interception form. An example of use of code injection may be found in [Hagimont and De Palma 2002].

2.4.4 Comparing Approaches

The main approaches to separation of concerns in middleware may be compared as follows.

1. Approaches based on meta-object protocols are the more general and systematic. However, they entail a potential overhead due to the back and forth interaction between meta- and base-levels.
2. Approaches based on aspects operate on a finer grain that those based on MOPs and provide more flexibility, at the expense of generality. The two approaches may be combined, e.g. aspects can be used to modify operations both at the base and meta levels.
3. Approaches based on interception provide restricted capabilities with respect to MOP or AOP, but provide acceptable solutions for a number of frequent situations. They are still lacking a formal model on which design and verification tools could be based.

In all cases, optimization techniques based on low-level code manipulation may be applied. This area is the subject of active research.

2.5 Historical Note

Architectural concerns in software design appeared in the late 1960s. The THE operating system [Dijkstra 1968] was an early example of a complex system designed as a hierarchy of abstract machines. The notion of object-oriented programming was introduced in the Simula-67 language [Dahl et al. 1970]. Modular construction, an approach to systematic program composition as an assembly of parts, appeared in the same period. Design principles developed for architecture and city planning [Alexander 1964] were transposed to program design and had a significant influence on the emergence of software engineering as a discipline [Naur and Randell 1969].

The notion of a design pattern came from the same source a decade later [Alexander et al. 1977]. Even before that notion was systematically used, the elementary patterns described in the present chapter had been identified. Simple forms of wrappers were developed for converting data from one format to another one, e.g. in the context of database systems, before being used to transform access methods. An early use of interceptors is found in the implementation of the first distributed file system, Unix United

[Brownbridge et al. 1982]: a software layer interposed at the Unix system call interface allows operations on remote files to be transparently redirected. This method was later extended [Jones 1993] to include user code in system calls. Stacked interceptors, both on the client and server side, were introduced in [Hamilton et al. 1993] under the name of subcontracts. Various forms of proxies have been used to implement remote execution, before the pattern was identified [Shapiro 1986]. Factories seem to have first appeared in the design of graphical user interfaces (e.g. [Weinand et al. 1988]), in which a number of parameterized objects (buttons, window frames, menus, etc.) are dynamically created.

A systematic exploration of software design patterns was initiated in the late 1980s. After the publication of [Gamma et al. 1994], activity expanded in this area, with the launching of the PLoP conference series [PLoP] and the publication of several specialized books [Buschmann et al. 1995, Schmidt et al. 2000, Völter et al. 2002].

The idea of reflective programming was present in various forms since the early days of computing (e.g. in the evaluation mechanism of functional languages such as Lisp). First attempts towards a systematic use of this notion date from the early 1980s (e.g. the metaclass mechanism in Smalltalk-80); the foundations of reflective computing were laid out in [Smith 1982]. The notion of a meta-object protocol [Kiczales et al. 1991] was introduced for the CLOS language, an object extension of Lisp. Reflective middleware [Kon et al. 2002] has been the subject of active research since the mid-1990s, and some of its notions begin to slowly penetrate commercial systems (e.g. through the CORBA standard for portable interceptors).

Chapter 3

Naming and Binding

Naming and binding are fundamental ingredients of any computing system. Naming deals with the designation of the various resources that compose a system, while binding is concerned with actual access to objects through names. This chapter starts with an introduction to the basic concepts of naming. It goes on with a discussion of design patterns for distributed name services, illustrated by case studies. Then comes an introduction to binding, followed by a discussion of a general pattern for distributed binding. The chapter concludes with a presentation of the naming and binding framework of Jonathan, a kernel that provides basic tools for the construction of distributed binding services.

3.1 Names

In a computing system, a *name* is an information associated with an object (the name *designates* the object) in order to fulfill two functions:

- to identify the object, i.e., to distinguish it from other objects, so the object can be (usually unambiguously) referred to.
- to provide an access path for the object, so the object can actually be used according to its specification.

In the above definition, the term "object" has a very general meaning and could be replaced by "resource", meaning anything that we may wish to consider as an independent entity, be it material or logical. Examples of objects, in that sense, are: a variable or a procedure in a program; a file; a software component; a memory cell, a track on a disk, a communication port; a device, either fixed (e.g., disk, printer, captor, actuator) or mobile (e.g., PDA, mobile phone, smart card); a user; a host on the Internet; a network. The specification of an object describes its interface, i.e., the set of operations that it provides to the outside world. Familiar instances of names are identifiers used in programming languages, addresses of memory cells, domain names in the Internet, e-mail addresses, file names, port numbers, IP addresses, URIs.

The two functions of a name (identification and access) impose different sets of requirements.

- A name used for identification should be permanently and unambiguously associated with the object that it identifies. For instance, [Wieringa and de Jonge 1995] state three requirements for "identifiers" (names used for identification purpose): an identifier designates at most one object; an object is designated by at most one identifier; an identifier always designates the same object (in particular, it should not be reused for a different object).
- A name used for access should allow the object to be physically located. As a consequence, such a name may change when an object moves; a new name may be created for efficiency reasons, e.g., if a copy of the object is put in a cache to speed up access. Such names are typically reused and cannot serve as unique identifiers.

Two remarks are in order: first, the requirements for identifiers are so stringent that they are almost never met in practice[1]; second, the requirements for identification and access are contradictory. As a consequence, in the naming schemes used in practical situations, an object is usually designated by two kinds of names: names that are primarily used for designation (but do not necessarily have all the uniqueness properties of an identifier; for instance, aliases are allowed, and names may be reused), and names that are primarily used for access, which typically contain some form of location-related information, e.g., a memory address, a network address, a port number, etc. Such names are usually called *references*. However, a reference to an object may not always be directly used for access (for example, in order to access a remote object, a communication protocol must be set up). Associating designation names with references, and making references usable for access, is the function of binding, the subject of Section 3.3.

Other criteria have been proposed to classify names. A common distinction is between names intended for use by humans, usually in the form of a character string (e.g., a symbolic file name, or an e-mail address) and names intended to be interpreted by some computing system (e.g., a bit string representing a memory address or an internal identifier). While intuitive, this distinction does not have deep implications. A more significant distinction is between "pure" and "impure" names [Needham 1993]. A pure name does not give any information as to the physical identity or location of the object it refers to, while an impure name does contain such information. Names used for designation may be pure or impure, while names used for access are, by definition, impure. The advantage of using an impure name is to speed up access to the named object; however, if the physical location of an object changes, an impure name that refers to it must usually be changed as well, while a pure name remains invariant. Changing a name may be problematic, because the name may have been copied and distributed, and it may be expensive or unfeasible to retrace all its occurrences. The choice between pure and impure names is a trade-off between flexibility and efficiency, and hybrid solutions are used in many cases (i.e., compound names, partly pure and partly impure).

The distinction between pure and impure names is also related to the distinction between identity and representation, as illustrated by the following examples. Using a pure name often amounts to introducing an additional level of abstraction and indirection.

[1]The naming scheme for Ethernet boards uses 48-bit identifiers, of which 24 bits uniquely identify a board manufacturer, and the 24 remaining bits are internally allocated (without reuse) by each manufacturer. These identifiers satisfy the above uniqueness properties.

Examples.

1. Usually, the mail address associated with a specific function within an organization is an impersonal one, such as `webmaster@objectweb.org`. Mail sent to such an address is redirected to the person (or group of persons) that performs the function. In addition to preserving anonymity, this simple device allows the function to be transparently transferred to a different person or group.
2. To increase availability and performance, some systems maintain multiple copies (replicas) of critical objects. The user of such an object is not usually aware of the replication; he sees a unique (abstract) object, designated by a single name. In contrast, the object management system must access and update replicas to maintain the illusion of a single copy, and therefore needs to identify each replica by a different name.

Names are essential for sharing information and other resources between different users or different programming systems. Names allow objects to be shared by reference, by embedding a name referring to the shared object in each of the objects that share it. This is usually cheaper, easier to manage, and more flexible than sharing by copy or inclusion (including the shared object, or a copy of it, in the sharing objects); for physical objects, sharing by reference is the only possibility. Many of the difficulties of binding names to objects are related to the management of such names embedded in objects.

Examples

1. Sharing by reference is the basic device of the World Wide Web, in which objects (web pages) are referred to by URIs embedded in web pages.
2. Code libraries (such as mathematical functions, string manipulation procedures, etc.) are usually shared by reference in the source programs, by embedding their (symbolic) name in the programs that use them. In the executable (compiled) programs, they still may be shared by reference (through a common address, if the system supports shared libraries), or they may be shared by inclusion (including a copy of the library programs in the executable code segment, and using its local address in that segment).

A *naming system* is the framework in which a specific category of objects is named; it comprises the rules and algorithms that are used to deal with the names of these objects. Usually, many naming systems coexist in a given application (e.g., there is a naming system for the users, another one for the machines, yet another one for the files, etc.), although some systems (e.g., Plan 9 [Pike et al. 1995]) have attempted to use a uniform naming system. In a given naming system, a *name space* defines the set of valid names, usually by providing an alphabet and a set of syntax rules.

The main issues in the design of a naming system are the following.

- Organizing the name space; this is done by means of *naming contexts*, as described in Section 3.1.1.
- Finding the object, if any, associated with a name; this process, called *name resolution*, is examined in Section 3.1.2

- *Binding* names to objects, i.e., managing the association between names and objects. Binding is the subject of Section 3.3

In a more general view of naming, an object may be characterized by some properties, i.e., assertions on some of its attributes. This characterization may or may not be unique. For example, one can look for a server delivering a specified service, identified by a set of attributes. This aspect of naming is further developed in Section 3.2.3.

3.1.1 Naming Contexts.

One could envision a name space as a uniform set of identifiers, such as the integers, or the strings on some alphabet. While conceptually simple, this scheme has practical limitations: searching in a large flat space is inefficient, and there is no simple way of grouping related objects. Therefore a name space is usually organized into naming contexts. A *naming context* is a set of associations, or bindings, between names and objects. Naming contexts correspond to organizational or structural subdivisions of a global name space, such as directories in a file system, departments in an organization, hosts in a computer network, or domains in the Internet DNS (3.2.2).

Context Graphs and Contextual Names

When dealing with names defined in different naming contexts, e.g., *NC1* and *NC2*, it is useful to define a new context *NC* in which *NC1* and *NC2* have names (e.g., *nc1* and *nc2*, respectively). This leads to the notion of a *context graph*: designating *NC1* by name *nc1* in *NC* creates an oriented arc in that graph, labeled by *nc1*, from *NC* to *NC1* (Figure 3.1a.

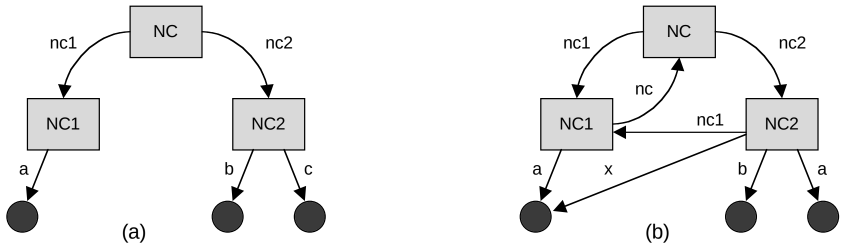


Figure 3.1. Naming contexts

A context graph may have any configuration (e.g., it may have cycles, multiple arcs between nodes, and it may be disconnected), and it may evolve dynamically when contexts and names are created or removed. For example, we may do the following operations on our context graph: create a new name, *x* in *NC2* for the object named *a* in *NC1*; create a name *nc1* in *NC2* for context *NC1*; create a name *nc* in *NC1* for context *NC*; change the name of object *c* in *NC2* to *a*. The resulting graph is shown on Figure 3.1b.

Composite names may now be defined: if *a* is the name of an object *A* in *NC1*, and *b* is the name of an object *B* in *NC2*, then *nc1.a* and *nc2.b* (where the separator "." denotes a name composition operator) respectively designate *A* and *B* in *NC* (Figure 3.1a). A name thus constructed, be it simple or composite, is called a *contextual name*: it is relative to some context, i.e., it is interpreted within that context.

The main operations on names, e.g., searching or resolution, involve navigating in a context graph by following inter-context arcs. Two notions are important here.

- The starting point, a specified context called a *root*. There may be one or several roots (multiple roots are mandatory in the case of a disconnected context graph but may be convenient in other situations).
- The current position, usually called the current context. An example is the notion of a working directory, present in most file systems.

Roots are intended to be stable reference points that seldom, if ever, change. A name relative to a root is said to be *absolute*; in the case of multiple roots, the name must specify which root it starts from. An absolute name has the same meaning in any context, as opposed to a *relative* name, which is only significant in a specified context (by default, the current context). Absolute and relative names are distinguished by syntax (e.g., in the Unix file system, absolute names start with "`/`", which is also the absolute name of the root).

Example.

In the context graph of Figure 3.1b, let *NC* and *NC2* be chosen as roots. Assume that the notation for an absolute name is <*root*>:<*name*>, where <*root*> is a universally known identifier for the specified root and <*name*> is a name relative to that root. Then, *NC:nc1.a*, *NC2:nc1.a*, and *NC2:x* are absolute names for *A*, and *a* is a relative name for *A* if the current context is *NC1*.

Note that there is a need for universal (or global) denotations, that are valid in any context. Examples are the names of the root(s) and the denotations of well-known invariant entities such as integers or character strings[2].

Restructuring Context Graphs

As seen in the previous section, a context graph may evolve by creating or deleting objects and naming contexts. Even in the absence of creations and deletions, it may be useful to perform structural changes. These include renaming entities, moving objects or naming contexts within a context graph, and combining context graphs.

An object that is shared between several contexts has a name in each of these contexts. In addition, an existing object or naming context, which has a name in some context, may receive a new name in another context, for the following reasons.

- Shortening the name. In a tree-structured naming space, for example, designating an object in a "cousin" naming context entails using an absolute name, or at least

[2]There is a distinction here again between identity and representation. The denotation "7" universally designates the integer 7 as an abstract entity. However, specifying an actual representation for this entity (e.g., on 16 or 32 bits, little- or big-endian, etc.) is resolved at a different level (this issue is similar to that of the actual identity of the person(s) behind `webmaster@objectweb.org`).

going up to the first common ancestor if parent contexts can be named. Such a name may be quite long if the tree is deep. Creating a direct link allows for a shorter, more convenient name.

- Logical grouping. Objects already named in different contexts may be logically regrouped by giving them new names in a common context.
- Access speed-up. Creating a new name may speed up access to the named object (this depends, however, on the implementation of name to object bindings).

Giving a new name to an object may be done in two different ways, illustrated by the notions of "hard" and "soft" links in the Unix file system.

A hard link has exactly the same status as the original name of the object. It may be created in the same context as the original name or in a different context. A soft link, by contrast, is an indirect designation: it points to a holder that contains a name for the object. In that sense, a soft link is just a hint: if the initial name of the object changes, the soft link becomes invalid because it still points to the original name.

These notions are illustrated by a simple (slightly contrived) example.

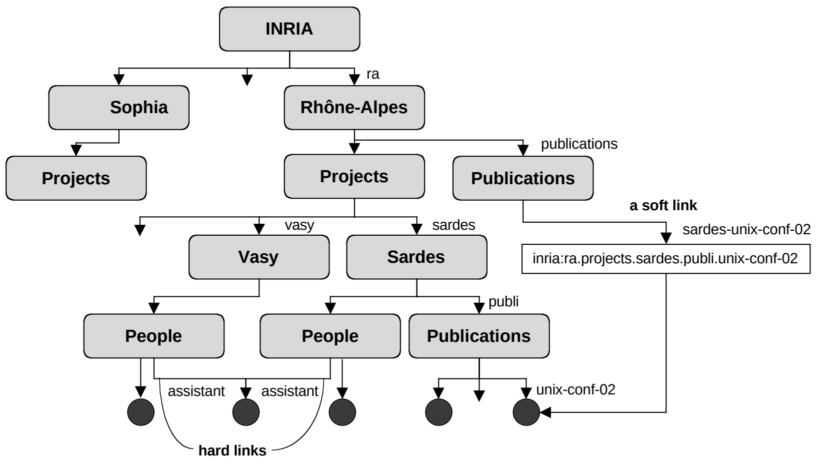


Figure 3.2. Hard and soft links

Figure 3.2 shows part of a name space representing research projects at INRIA. Projects Sardes and Vasy happen to share a project assistant. This person is designated by the two names *inria:ra.projects.sardes.people.assistant* and *inria:ra.projects.vasy.people.assistant*, which have the same status, i.e., a hard link. On the other hand, suppose INRIA Rhône-Alpes decides to group all the publications of its projects in a common naming context, *inria:ra.publications*. One way of doing it is to create soft links in this context to all existing publications of the projects. Thus, in this context, a link named *sardes-unix-conf-02* is made to point to a holder containing the original name *inria:ra.projects.sardes.publi.unix-conf-02*. However, if the original name is changed because of a restructuring of the Sardes publication context, the link becomes invalid and should be explicitly reestablished to the new name.

Some situations call for combining (parts of) previously separated context graphs. This happens when physically connecting separate devices or subsystems, each of which has its own context graph. Another instance is merging two companies or reorganizing an administration.

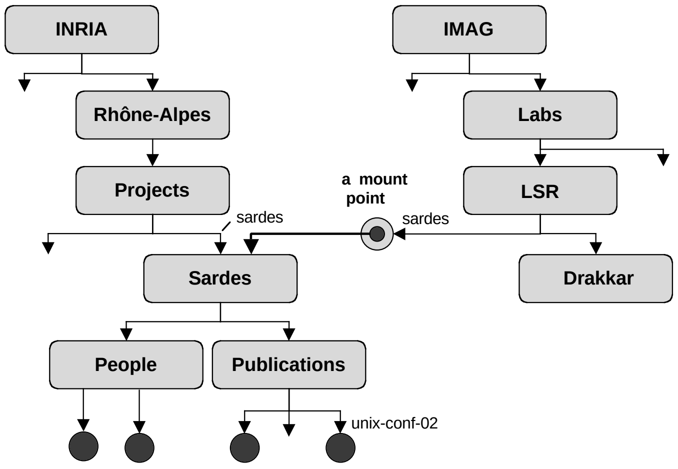


Figure 3.3. Mounting

Following up on the previous example, suppose that project Sardes of INRIA becomes a joint project of INRIA and another research institute, IMAG. This may be done in the following fashion:

- Create a new entry of name *sardes*, called a *mount point*, in the context *imag:labs.lsr* (the new context in which the project should be known). This is essentially a holder for a link to another context.

- Connect the existing context *inria:ra.projects.sardes* to this holder; this is done through a *mount* operation.

From now on, project Sardes may be named in the IMAG context graph exactly like IMAG's own projects. Nothing in the name's structure reveals the existence of the link through the mount point (compare *imag:labs.lsr.sardes* with *imag:labs.lsr.drakkar*).

In addition to name management, there is another aspect to mounting: access must be guaranteed to the mounted object. This aspect is considered in Section 3.3

Mounting was initially introduced to incorporate removable devices (e.g., a removable disk stack) to the Unix file system, by making the device's directories and files part of the file system. It was generalized to integrate the file systems of several networked machines in a single unified file system, the Network File System (NFS) [Sandberg et al. 1985].

Another way of combining separate context graphs is to give them names in a common root context. This was done, for instance, in Unix United [Brownbridge et al. 1982], the first attempt towards unifying separate file systems. This technique, however, entails changing all absolute names of existing objects. The GNS system [Lampson 1986] also allows upwards extensions, but proposes a remedy to the name change problem, by defining an imaginary super-root which has all directories as its children and uses global identifiers. Any name may then be made absolute by prefixing it with the global identifier of an appropriate directory[3], and does not need to be changed if the context tree is extended upwards. When a directory is created as an upward extension, or as a common root of existing context trees, a mapping is set up between the global identifiers of its directory children and their local names in the new directory, so that the old (global) names may be correctly interpreted in the new context. Note that the problem is now to define and to manage unique global identifiers in the super-root directory.

Conclusion

In summary, a naming scheme based on naming contexts has the following benefits.

- Contexts are convenient for grouping related objects; devices like current context, relative naming, symbolic links, allow for shorter names;
- Names may be chosen independently in different contexts, and nothing prevents the same name from being reused in different contexts (e.g., in Figure 3.1b name *a* exists in contexts *NC1* and *NC2* and refers to different objects; name *nc1* exists in contexts *NC* and *NC2* and refers to the same object, context *NC1*).
- Contexts may be used to speed up navigation and searching in the name space.
- The naming system may be indefinitely extended by creating new contexts, and by linking existing contexts together.

3.1.2 Name Resolution.

In order to determine the object, if any, referred to by a valid name, a process called *name resolution* must be carried out. Name resolution starts in an initial naming context, and proceeds in steps. At each step, a component of the name (a label in the current context) is resolved. This operation either delivers a result, the *target*, or fails (if the current label is not bound to any object in this context). Three cases may occur.

- The target is a typed value (not a name), i.e., a couple [`T`, `V`], where `T` is a type and `V` a value of that type. The type does not have to be explicit, i.e., it may be determined by the naming context or derived from the name.
- The target is a primitive name (also called an address), i.e., a name that cannot be further resolved. If the designated object is a physical entity (e.g., a memory cell, a disk drive, a host, a mobile device), the address identifies the object. If the designated object is a data structure (e.g., a record, a Java object, a file), the address identifies a physical container or location for the object[4].

[3]this may be done automatically by the system using the notion of a current context.

[4]It often happens that the designated object has a complex structure and is not physically identified by a single address. In that case, the target is a data structure called a *descriptor* (a form of reference), which contains the information on the physical layout of the object, together with other information (e.g., protection). An example of a descriptor is the *i*-node, a data structure that describes a file in the Unix operating system, and includes the disk addresses of the blocks that contain the file. Another example is a couple (network address, port number), which refers to a service on a network.

- The target is a non primitive name (in some naming context). The resolution process is then called again on the target name in that context.

We now propose a general scheme for the name resolution process. Assuming that an initial naming context is known (we shall see later how it is determined), we represent names and contexts as objects (in the sense of object-oriented programming, i.e., the association of a state and a set of methods), and the resolution process then takes the form of the following method call:

`target = context.resolve (name)`[5]

in which `context` is the naming context, `resolve` is a method that embodies the name resolution algorithm (the resolver) in this context, and `name` is the name to be resolved. If the resolution succeeds, `target` is the result, which may have different forms:

- a value associated with the name.
- another name, together with a new context.

In the latter case, the name must in turn be resolved in the new context by calling its resolver. The resolution process is repeated[6] until either it fails (in which case the original name cannot be resolved) or it delivers a value, i.e., either the object itself or the name of a physical container for it (e.g., a memory address, a descriptor). Note that the resolution may be repeated using either iteration or recursion. Let ($name_{i+1}$, $context_{i+1}$) the couple (name, context) delivered by the resolver of $context_i$.

- In the iterative scheme, the results are delivered to the initial resolver, which keeps control of the resolution process, by successively calling the resolver of $context_i$ on the $name_i$, for increasing values of i.
- In the recursive scheme, control goes from one resolver to the next: $context_i$ calls the resolver of $context_{i+1}$ on the name $name_{i+1}$, and so on.

The recursive scheme generates less messages than the iterative scheme. However, there is a heavier load on the servers, and recovery from failure is more complex, since the initial resolver does not keep track of the successive calls. See [Tanenbaum and van Steen 2006], 4.1.3) for a detailed discussion.

As an example, consider the resolution of an URI in the World Wide Web (Figure 3.4), which may be summarized as follows. The initial naming context is determined by examining the protocol part (i.e., the first string of the URI, up to the first ":"). For instance, if the protocol is `http` or `ftp`, the context is the Internet; if it is `file`, the context is the local file system; etc.

[5] A logically equivalent scheme would be `target = name.resolve (context)`, i.e., the `resolve` method may as well be borne by the name as by the context. See the description of the Jonathan platform (Section 3.4) for practical examples.

[6] If the context graph contains cycles, the resolution process may enter an infinite loop. In practice, termination is ensured by setting an upper limit to the number of resolution steps.

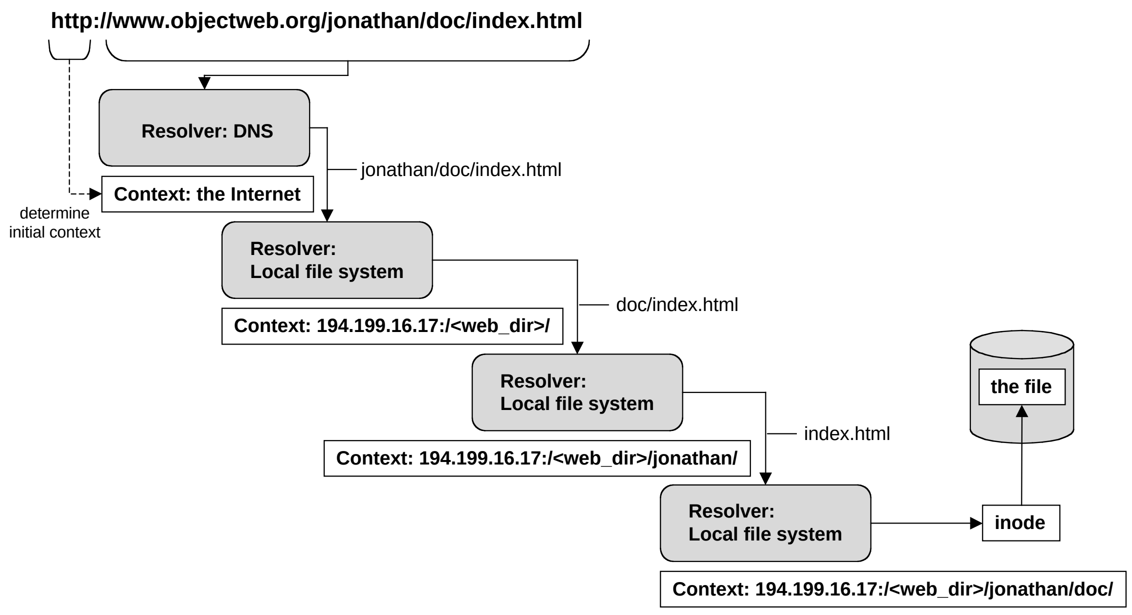

http://www.objectweb.org/jonathan/doc/index.html
Resolver: DNS
Context: the Internet
determine initial context
jonathan/doc/index.html
Resolver: Local file system
Context: 194.199.16.17:/<web_dir>/
doc/index.html
Resolver: Local file system
Context: 194.199.16.17:/<web_dir>/jonathan/
index.html
Resolver: Local file system
Context: 194.199.16.17:/<web_dir>/jonathan/doc/
inode
the file

Figure 3.4. Resolving a name in the World Wide Web

In our example, the protocol is `http`. Therefore the string between `//` and the first following `/`, i.e., `www.objectweb.org` is assumed to be the domain name of a server, and is resolved in the context of the Internet (the resolver in this context is the DNS, see 3.2.2). Let us assume this resolution succeeds, delivering the IP address of a host (194.199.16.17). Its result is a new context (the web server directory on the host, noted `/<`*web_dir*`>`), together with a new name (the remaining part of the URI, i.e., whatever follows the `/` that ends the host name). The new name is now resolved in the new context. This in turn may lead to a chain of resolutions, going down the file hierarchy on the server. Assume the named object is a file, like in our example (*index.html*). Then the resolution ends with the descriptor of that file, from which the file itself may be retrieved from the disk. If the string following the last `/` contains a `?`, then the substring on the left of `?` is interpreted as the name of the resolve procedure in the current context, and the sub-string on the right as the name to be resolved. This is used, in practice, to perform an attribute-based look-up (the equivalent of a query in a database, to find objects satisfying a given predicate). In this example, the structure of the name reflects the chain of contexts in which the resolution takes place (the name of the object explicitly contains the names of the contexts). This, however, needs not be the case (e.g., the original name may designate a descriptor which contains both the name of the new context and the name to be resolved in this context, and so on down the chain).

The question of how the first context is determined remains to be settled. There are several possibilities.

- It may be inferred from the name to be interpreted (e.g., in the example of URIs, where the initial context is determined by the name of the protocol).

- It may be found in a predefined location (e.g., in the case of a virtual address, where the context is determined by the contents of the MMU and of the page tables, which are addressed by a predefined register).
- It may be part of the current environment (e.g., the "working directory" determines the context for file name interpretation in an operating system).

A name may be valid in several different contexts, and one may wish to select a given context for a specific execution. One solution in that case is to explore the set of possible contexts according to a specified search rule (e.g., linear order in a list, depth first in a directed graph, etc.) until the name can be resolved. The set of contexts, together with the search rules, defines a *search path*; by changing the order of the contexts in the search path, a given name may be resolved into different objects. This technique is used by linkers for resolving the names of library routines. It allows (for example) a user to supply his own version of a routine instead of that provided by the system, by including his private version in a directory and placing that directory at the head of the search path.

3.2 Patterns for Distributed Name Services

The function of a *name service* is to implement name resolution for a given name space. In this section, we present the main design principles and patterns applicable to name services in a distributed environment. We use a few examples for illustration, but we do not intend to present full case studies of working name services. These may be found in the provided references.

Section 3.2.1 presents the main problems presented by the design of a large scale distributed naming service, and some general principles and techniques used to solve them. While these apply to all naming services, different situations occur in practice. Broadly speaking, the main distinctive criterion is *dynamism*, the rate at which change occurs in the structure and composition of the system.

In section 3.2.2, we examine the case of a naming system for a relatively stable environment, in which the rate of change is slow with respect to the rate of consultation of the naming service. This situation is that of the global Internet, and the Domain Name Service is a representative example of this class of systems.

Section 3.2.3 deals with the additional constraints introduced by highly dynamic environments, characterized by mobility, attribute-based search, and rapidly changing composition of the system. A typical application is *service discovery*, in which the object being looked up is a service provider.

3.2.1 Problems and Techniques of Distributed Naming

The requirements of a naming service are similar to those of any distributed application. However, some of them are made more stringent by the function and environment of a naming service.

- *Availability.* Availability is defined as the fraction of time the service is ready for use (more details in Chapter 11). Given the central role of the name service in any working environment, availability is perhaps its most important requirement.

- *Performance.* The main performance criterion for a naming service is latency. This is because name resolution introduces an incompressible delay in the critical path of any request for a remote service.
- *Scalability.* A service is scalable if its performance remains acceptable when its size grows. Measures of size include the number of objects, the number of users, and the geographical extent. A naming service may potentially manage a large number of objects and its span may be worldwide. In some applications, new objects are created at a very high rate; merging several applications may give rise to a large name space. Therefore scalability is also an important requirement for a name service.
- *Adaptability.* A distributed name service often operates in a changing environment, due to the varying quality of service of the communication network, to the mobility of objects and users, and to evolving requirements. The service therefore needs to be adaptable, in order to maintain its performance in spite of these variations.

To meet these requirements, a few design principles and heuristics have shown to be efficient.

- Avoid an organization that involves a single point of decision. This is detrimental both to performance (bottleneck) and availability (single point of failure).
- Use *hierarchical decomposition* as a guide for partitioning the work and for delegating responsibility.
- Use *redundancy*; arrange for several independent ways to get an answer.
- Use *hints* in order to speed up the search. A hint may be invalid, but there should be a way to get up to date information if needed. *Caching* is the main relevant technique.
- Most information becomes stale with time; use *timeouts* to discard out of date information.
- Use *late binding* when adaptability is needed.
- *Avoid broadcast*, which does not scale well; however, some use of broadcast or multicast is unavoidable in dynamic environments; in that case, use *partitioning* to restrict the range.

3.2.2 Naming Services in a Slowly Changing World

In this section, we review the design principles of a naming service for a relatively stable environment, in which the frequency of change is small with respect to that of consultation. The Domain Name System (DNS), our first case study, was designed under this hypothesis (however, relative stability does not preclude change: the Internet has actually expanded at an exponential rate since the introduction of the DNS). We then examine the techniques used to deal with mobility, still at a relatively slow rate.

The Domain Name System

The Domain Name System [Albitz and Liu 2001] is the name service of the Internet. In this section, we briefly examine the principles of its organization and operation.

The name space is organized as a hierarchy of spaces called *domains*, as shown on Figure 3.5. The notation for DNS global names is similar to that of files names in a hierarchical file system, except that the local names are concatenated in ascending order, separated by dots, and that the name of the root (denoted by a dot) is usually omitted; thus the hierarchy is actually considered as a forest, starting from a set of top-level names. For example, `turing.imag.fr` denotes a leaf object (in this case a machine in the domain `imag.fr`), while `research.ibm.com` denotes a domain.

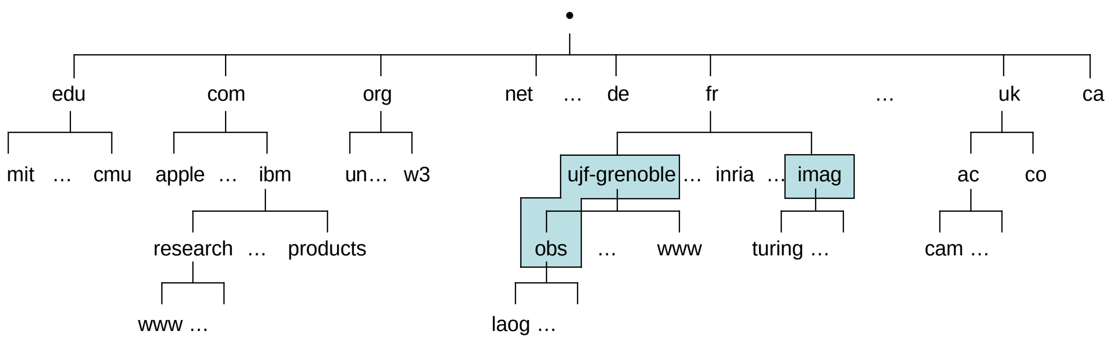


Figure 3.5. The DNS domain hierarchy

There are a few hundreds top-level domains, which may be generic (e.g., `com`, `edu`, `org`, etc.) or geographical (e.g., `ca`, `de`, `fr`, `uk`, etc.). These names are allocated by a global authority, the ICANN (Internet Corporation for Assigned Names and Numbers), an internationally organized, non-profit company. The management of lower level domains is delegated to specific authorities, which themselves delegate the management of their sub-domains. Thus the name `ujf-grenoble.fr` was allocated by AFNIC, the French authority in charge of the top-level `fr` domain, and the `ujf-grenoble.fr` domain is managed by the local administration of Université Joseph Fourier (UJF).

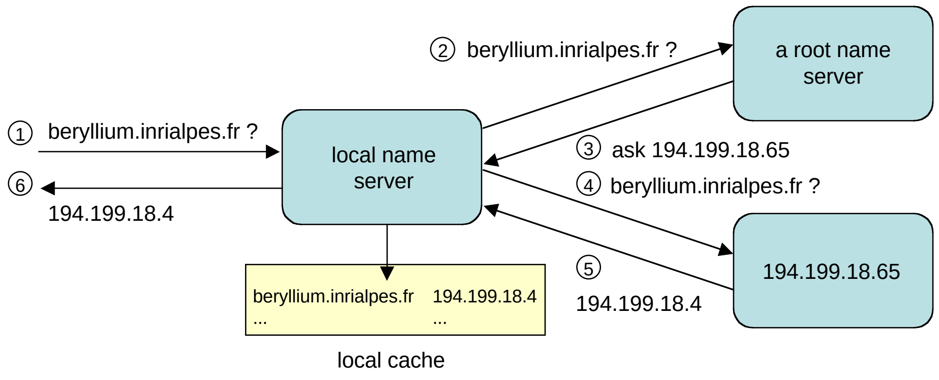
The physical organization of the DNS relies on a set of name servers. As a first approximation, one may consider that each server manages a domain. The real situation deviates from this ideal scheme on the following points.

- There is no server for the domain ".". The root server manages the first two levels of the hierarchy, i.e., it knows all the servers for the second-level domains such as `mit.edu`, `ibm.com`, `inria.fr`, etc.

- Below the second level, some sub-domains may be grouped with their parent domain to form a *zone* (for example, a university department that has its own domain may rely on the system administrators of the university for the management of that domain). Thus a zone may include a single domain or a subtree of domains (examples of zones are shown in gray on Figure 3.5). A zone is a management unit, and a server is associated with a zone.

- The servers are replicated, to ensure both scalability and fault tolerance. Each zone is managed by at least two servers. The root server has currently a dozen of replicas.

A server maintains a set of records for the objects included in its zone. Each record associates a name and some attributes of the object that the name designates (in most cases, the IP address of a host). Resolving a name, i.e., finding the object associated with the name, follows the general scheme outlined in 3.1.2, and illustrated on Figure 3.6. The search starts form a local server (the address of at least one local server is part of any client's environment). If the local server does not hold a record for the requested name, it interrogates the root server (i.e., it consults the closest replica of that server). If the root server cannot answer the query, it finds the address of a server that is more likely to do it, and so on until the name is resolved or declared as unknown. The search usually combines an iterative and a recursive scheme: all requests involving the root server are iterative[7], i.e., the root server returns the IP address of a server instead of querying that server, and the requests further down the chain are usually recursive.



② beryllium.inrialpes.fr ?
a root name server
① beryllium.inrialpes.fr ?
local name server
③ ask 194.199.18.65
④ beryllium.inrialpes.fr ?
⑥
194.199.18.4
194.199.18.65
⑤
194.199.18.4
beryllium.inrialpes.fr 194.199.18.4
... ...
local cache

Figure 3.6. Resolving a name in the Domain Name System

In order to speed up name resolution, each server maintains a cache containing the most recent resolved records. The cache is first looked up before any search involving the server. However, an answer coming from a server's cache is only indicative, since the caches are not directly updated after a change, and may thus contain obsolete data. An answer coming from a record held by a zone server is said to be *authoritative* (it can be trusted), while an answer coming from a cache is *non-authoritative* (it is only a hint). Caching greatly improves the efficiency of the search: a vast majority of DNS requests are satisfied in at most two consultations.

The DNS is a highly successful system: its structure and operation are still those defined by its initial design [Mockapetris and Dunlap 1988], while the size of the Internet has scaled up by five orders of magnitude! This success may be ascribed to the following features of the design.

- A highly decentralized structure, following the domain hierarchy. There is no single point of decision. Adding a new domain (usually as a new zone) automatically adds a server, which helps scalability.

[7]in order to reduce the load on the root server.

- Server replication, both for performance and for fault tolerance. The servers located at the higher levels of the domain hierarchy are the most heavily replicated, since they are consulted often; this does not induce a high consistency maintenance cost, since changes are less frequent at the higher levels.
- Intensive use of caching, at all levels. Since locality of reference also applies to names, the performance of caching is good.

As mentioned above, the DNS is representative of a situation in which the universe of names evolves slowly. Another popular name service in this class is LDAP (Lightweight Directory Access Protocol) [LDAP 2006], a model of directory services for large scale enterprise systems, based on the X.500 standard [X.500 1993].

Techniques for "Slow" Mobility

The DNS was designed to deal with name space extension, but not with mobility. The association of a name with an address is assumed to be permanent: to change the association, one should remove the target object, and recreate it with the new address.

Here we examine some techniques used to deal with mobile targets, still assuming relative stability with respect to the consultation rate. We consider two typical situations.

1. Mobility of a target software object between different hosts of a network.
2. Mobility of a host itself.

In the first case, the successive nodes hosting the target object provide a support that can be used for tracking the target. In the second case, there is no such support, so a new fixed element (the so-called home) is introduced.

Consider a situation in which an object, initially located on host $N1$, moves to host $N2$. The technique of *forwarding pointers* [Fowler 1986] consists in leaving a pointer to host $N2$ in the place of the initial target on host $N1$. If the object moves further, a new pointer to the new location $N3$ is placed on host $N2$. This is illustrated on Figure 3.7 (a and b).

However, after a number of moves, the chain becomes long, which has a cost in access efficiency (multiple indirections to remote hosts) and availability (the target may be lost if one of the intermediary hosts crashes). One may then shorten the chain, by updating the initial reference to the object (Figure 3.7c). The remaining pointers are no longer reachable and may be garbage collected. Note that this technique implies the existence of a descriptor for the object, which serves as a unique access point.

The approach to host mobility is different, but still relies on indirection. The problem is to allow a mobile host to attach to different subnetworks of a fixed internetwork, while keeping the same network address in order to ensure transparency for the applications running on this host.

The Mobile IP protocol [Perkins 1998], designed for mobile communication on the Internet, works as follows. The mobile host has a fixed address $< home\ network, node >$. On the home network (a local area network that is part of the Internet), a *home agent*

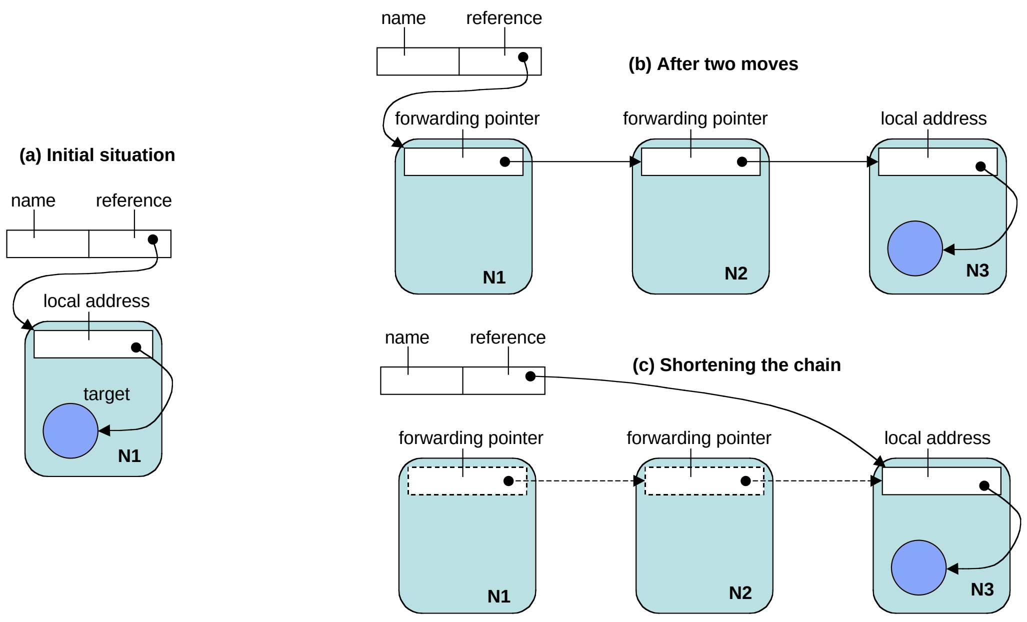


Figure 3.7. Forwarding pointers

running on a fixed host is in charge of the mobile host. All messages directed to the mobile host go to the home agent (this is done by setting up redirection tables on the routers of the home network). If the mobile host moves and connects to a different network, it acquires an address on that network, say $< foreign\ network, node1 >$. A foreign agent located on a fixed host of the foreign network takes care of the mobile host while it is connected to that network. The foreign agent informs the home agent of the presence of the mobile host on the foreign network. Messages directed to the mobile host still go to the home agent, which forwards them to the foreign agent using a technique called *tunneling* (or encapsulation) to preserve the original address of the mobile host.

This is an extremely schematic description, and various details need to be settled. For example, when a mobile host connects to a foreign network, it first needs to find a foreign agent. This is done by one of the service discovery techniques described in 3.2.3, for instance by periodic broadcast of announcements by the available foreign agents. The same technique is used initially by a mobile host to choose its home agent.

3.2.3 Dynamic Service Discovery Services

Distributed environments tend to become more dynamic, due to several factors: increasing mobility of users and applications, favored by universal access to wireless communication; development of new services at a high rate, to satisfy new user needs; service composition by dynamic aggregation of existing services and resources.

This situation has the following impact on naming services.

- The objects being searched are *services* rather than servers, since the location or identity of a server delivering a specific service is likely to change frequently.

- A service is usually designated by a set of attributes, rather than by a name. This is because several servers may potentially answer the need, and their identity is not necessarily known in advance. The attributes provide a partial specification of the requested service.
- If the communication is mainly wireless, additional constraints must be considered, such as energy limitation and frequent disconnection.

In the rest of this section, we first introduce a few basic interaction schemes, which may be combined to build patterns for service discovery. We next present the principles of a few current service discovery systems.

Interaction Patterns for Service Discovery

In the context of service discovery, we define the state of a client as its current knowledge about the available services. In an environment that is changing at a high rate, with a high probability of communication failures, the so-called *soft state* approach is preferred for state maintenance. The notion of soft state has initially been introduced for network signaling systems [Clark 1988]. Rather than relying on a explicit (*hard state*) procedure to propagate each change of state, the soft state approach maintains an approximate view of the state through periodic updates. These updates may be triggered by the clients (*pull* mode), by the services (*push* mode), or by a combination of these two modes. In addition, in the absence of updates after a preset time interval, the state is considered obsolete and must be refreshed or discarded.

The soft state maintenance scheme, which combines periodic update and multicast, is called *announce-listen* [Chandy et al. 1998]. It may be implemented using several interaction patterns, as explained below.

Service discovery involves three types of entities: *clients* (service requesters), *services* (short for "service providers"), and *servers* (short for "directory servers"). The clients need to find available services that match a certain description. Service description is briefly examined at the end of this section.

There are two kinds of interactions: announcement (a service declares itself to a server) and search (a client queries a server for a service matching a description). Both may use the pull or push mode. These interaction patterns are summarized on Figure 3.8

We assume, when needed, that each client and each service knows the address of at least one server (we describe later how this is achieved). A service registers its description and location with one or several servers (Figure 3.8a), using single messages or multicast. A client's query may be satisfied either in pull mode, by interrogating one or several servers (Figure 3.8b1 and b2), or in push mode, by listening to periodic servers' broadcast announcements (Figure 3.8c). Since the number of clients is usually much larger that the number of servers, the push mode should be used with care (e.g., by assigning each server a restricted multicast domain).

In a small scale environment, such as a LAN, the servers may disappear, and the services directly interact with the clients. This again may be done in pull mode (Figure 3.9a) or in push mode (Figure 3.9b).

Two issues remain to be examined;

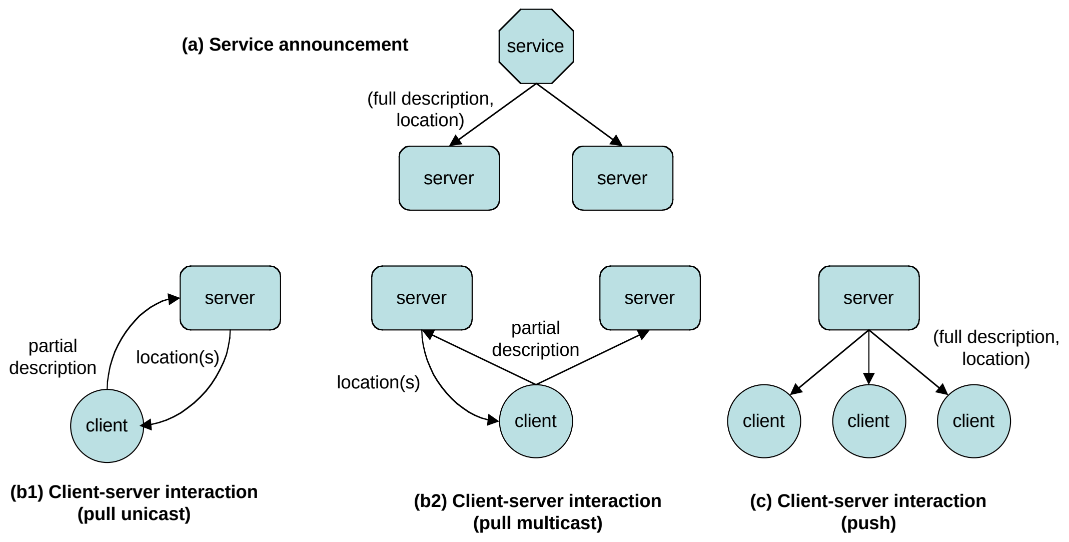

(a) Service announcement
service
(full description, location)
server
server
server
partial description
location(s)
client
(b1) Client-server interaction (pull unicast)
server
server
partial description
location(s)
client
(b2) Client-server interaction (pull multicast)
server
(full description, location)
client
client
client
(c) Client-server interaction (push)

Figure 3.8. Interaction patterns for service discovery

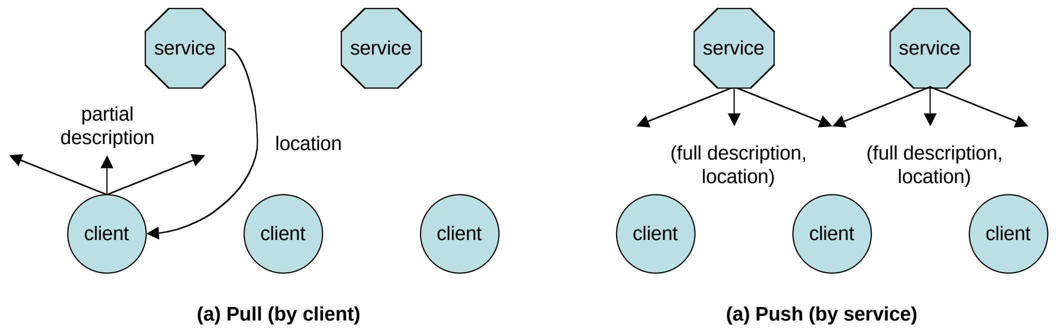


Figure 3.9. Interaction patterns for service discovery (without servers)

- *Describing the service.* In the simplest case, a service description is a couple (attribute, value), such as used in the early trading servers. In the current practice, a description is a complex ensemble of such couples, usually organized as a hierarchy, and often described using the XML formalism and the associated tools. A query is a pattern that matches part or whole of this structure, using predefined matching rules. In pull mode, the matching process is done by the server; in push mode, it is done by the client.

- *Finding servers.* A client or a service may again use pull (broadcasting a query for servers) or push (listening for server announcements) to find the servers. This is illustrated on Figure 3.10. Again the broadcast range should be restricted to a local environment.

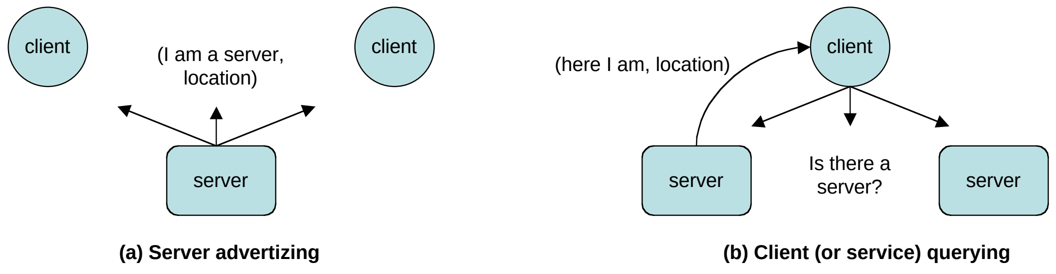


Figure 3.10. Discovering a server

Finally, in an open environment, security must be ensured to prevent malicious agents from masquerading as bona fide servers or services. This is achieved through authentication techniques (Chapter 13). The case studies presented below provide trusted service discovery.

Examples of Service Discovery Services

Many service discovery services have been proposed in recent years, both as research projects and as standards proposals. As an example, we briefly describe the principle of the Service Location Protocol (SLP), a standard proposal developed by an Internet Engineering Task Force (IETF) working group.

The intended range of SLP is a local network or an enterprise network operating under a common administration. The SLP architecture [Guttman 1999] is based on the three entities defined in the above discussion, namely clients (called User Agents, or UA), services (called Service Agents, or SA), and servers (called Directory Agents, or DA). UAs and SAs can locate DAs by one of the methods already presented (more details below). In the standard mode of operation, an SA registers its service with a DA, and an UA queries a DA for a specified service; both registration and query use unicast messages. The response to a query contains a list of all SAs, if any, that match the client's requirements. A service is implicitly de-registered after a specified timeout (it thus has to be re-announced periodically).

The above mode of operation works well on a local area network. For a larger scale environment, consisting of interconnected networks, scalability is ensured through the following techniques.

1. *Multiple Directory Agents.* An SA registers with all the DAs that it knows. This favors load sharing between the DAs, and increases availability.
2. *Scoping.* A *scope* is a grouping of resources according to some criterion (e.g., location, network, or administrative category); SAs may be shared between scopes. A scope is designated by a single level name (a character string). Access of UAs to scopes is restricted, and an UA may only find services in the scopes to which it has access. This limits the load on the directory agents.

SLP can also operate without DAs, in which case an UA's request is multicast to the SAs (pull mode), which answer by a unicast message to the UA.

At system startup, UAs and SAs attempt to discover the DAs using a protocol called multicast convergence. This protocol is also used by the UAs if no DA is found. It works as follows. An agent attempting to discover DAs (or services, in the case of an UA operating without DAs) multicasts a request in the form of a *Service Request* message. In response, it may receive one or more unicast messages. It then reissues its request after a wait period, appending the list of the agents that answered the first request. An agent receiving the new request does not respond if it finds itself on the list. This process is repeated until no response is received (or a maximum number of iterations is reached). This ensures recovery from message loss, while limiting the number and size of the messages exchanged.

Security is ensured by authenticating the messages issued by SAs and DAs through digital signatures. Authentication is provided in each administrative domain.

Another example, developed as a research project, is the Ninja Service Discovery Service (SDS) [Czerwinski et al. 1999]. SDS shares many features of SLP, but has investigated the following additional aspects.

- *Scalability.* While the name space of SLP scopes is flat, SDS servers (the equivalent of DAs) are organized in a hierarchy, each server being responsible for one or several domains. In addition, a server may spawn new servers to cope with a peak of load. Servers use caching to memorize other server's announcements.
- *Security.* In contrast with SLP, SDS provides cross-domain authentication, and authenticates clients (user agents), to control clients' access to services. Access control is based on capabilities.

Other service discovery services are part of industrial developments such as Universal Plug and Play (UPnP) [UPnP] and Jini [Waldo 1999]. A comparison of several service discovery protocols may be found in [Bettstetter and Renner 2000, Edwards 2006].

3.3 Binding

Binding is the process of interconnecting a set of objects in a computing system. The result of this process, i.e., the association, or link, created between the bound objects, is also called a binding. The purpose of binding is to create an access path through which an object may be reached from another object. Thus a binding associates one or several sources with one or several targets.

We first present in 3.3.1 a few instances of binding in various contexts, and the main properties of binding. We then introduce in 3.3.2 a general model for distributed binding. This is the base of the main pattern for binding, the subject of 3.3.3. We conclude in 3.3.4 with an overall view of the place of binding in distributed services.

3.3.1 Examples and Basic Techniques

Usual instances of bindings are the following:

- Associating a name with an object in a context creates an access path through which the object can be accessed using the name. A typical example is language level binding (e.g., associating the identifier of a variable with a storage location containing its value) performed by a compiler and a linking loader.
- Opening a file, an operating system primitive, creates a link between a file descriptor (an object local to the calling process) and the file, which resides in secondary storage. This involves setting up internal data structures, e.g., a local cache for accessing a portion of the file and a channel between that cache and the disk locations that contain the file. After opening a file, read and write operations on the local descriptor are translated into actual data transfers from and to secondary storage (data present in the cache is read from there).
- Compiling an interface definition for a remote procedure call creates client and server stubs (relays for transmission) and a network connection, allowing the procedure to be executed remotely as a result of a local call.
- Setting up a chain of forwarding pointers to a mobile object is a binding, which is preserved by updating the chain when the object migrates to a new location.

Binding may be done at the language level, at the operating system level, or at the network level; most frequently, several levels are involved. Language level binding typically takes place within a single address space, while operating system and network bindings usually bridge several different address spaces. Binding may be done in steps, i.e., the access chain between two objects may be partially set up, to be completed at a further stage.

In addition to setting up an access path, binding may involve the provision of some guarantees as to properties of this access path. For example, binding may check access rights (e.g., at file opening), or may reserve resources in order to ensure a prescribed quality of service (e.g., when creating a channel for multimedia transmission, or when creating a local copy of a remote object to speed up access).

An important notion is that of *binding time*, i.e., the point, in the lifetime of a computing system, at which binding is completed. Typical examples follow.

- Binding may be static (e.g., the denotation of a constant in a program is statically bound to that constant's value).

- Binding may occur at compile time (a compiler binds a variable identifier to an offset in a memory segment; a stub compiler binds a procedure identifier to the location of a stub, itself bound to a channel to a remote location).

- Binding may occur at link time (a linker explores a search path to find the objects associated with the names that remain unresolved after compilation; a component interconnection language is interpreted by a script that sets up access paths from a component to remote components).

- Finally, binding may be dynamic, i.e., deferred until execution time. Dynamic binding is performed when an unbound object is accessed; it may be done at first access or at each access. Binding a virtual address to a storage location in a paged virtual memory is an example of dynamic binding, which occurs at each page fault; another example is the setting up of a forwarding pointer at object migration.

Delayed binding allows greater flexibility, since the decision about how to reach the target of the binding is made at the last possible moment and may then be optimized according to the current situation. Dynamic binding also simplifies evolution: in an application made of many components, only the modified components need to be recompiled, and there is no global relink. Virtual memory binding uncouples the management of virtual addresses from that of physical storage, allowing each management policy to be optimized according to its own criteria.

Two basic techniques are used for binding: name substitution and indirection. These techniques are usually combined, and may be used recursively, to create a chain of bindings.

- *Name substitution* consists in replacing an unbound name by another name containing more information on the target of the binding. This is the main technique used for language level bindings; for example, after compilation, all occurrences of a variable identifier in the text of the program are replaced, in the executable file, by the address of the memory location allocated to the variable. This may be done in steps (e.g., if the output of the compilation is a relocatable file, then an offset is provided, not a complete address).

- In the *indirection* technique, an unbound name is replaced by (the address of) a descriptor that contains (or points to) the target object. This technique is well suited to dynamic binding, because all that is needed is to change the contents of the descriptor (there is no need to locate all the occurrences of the original name). On the other hand, each access to the target incurs the cost of at least one indirection. Forwarding pointers used in distributed systems to locate mobile objects (Section 3.2.2) are an example of this technique.

Note that name substitution creates an efficient binding at the expense of the loss of the original name. If that name is still needed, e.g., for debugging, then it is necessary to explicitly keep an association between the original name and its binding (for example in the form of a symbol table in language level binding).

3.3.2 A Model for Distributed Binding

Distributed systems are built by interconnecting hardware and software components located on different hosts over a network. There is an important difference between centralized and distributed systems as regards binding. In a centralized system, a memory address may be directly used to access an object located at that address. In a distributed system, a reference to a remote object (the equivalent of an address), such as [*host network address*, *port number*] is not directly usable for access. One first needs to actually create a binding to the remote object by building an access chain involving a network protocol, as shown in the following simple example.

Consider the case of a client to server connection using sockets. The server associates a server socket with a port number corresponding to some provided service, and makes this socket wait for incoming requests, through an `accept` operation (Figure 3.11 a). Assume that the client knows the name of the server and the port number associated with the service. Binding consists in creating a socket on the client host and connecting this socket, through a `connect` operation, to the above server socket. The server socket, in turn, creates a new socket linked to the client socket (Figure 3.11 b). The name of the client socket is now used for accessing the server, while the server socket remains available for new connections. This is an instance of binding by indirection: the sockets on the client and server site act as the intermediate descriptors between the client and server processes.

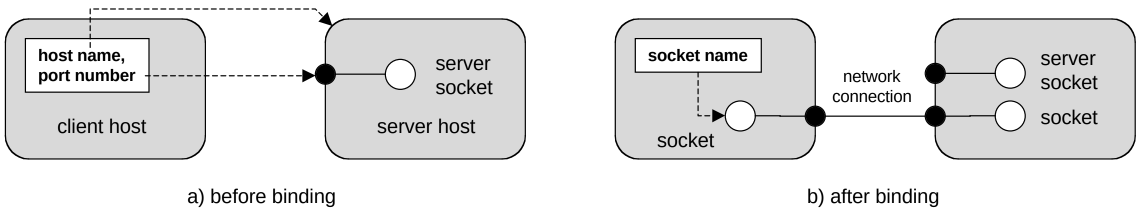


Figure 3.11. Client to server binding using sockets

It should be noted that the binding may only be set up if there is actually a server socket accepting connections on the target port. As a rule, the binding must be prepared, on the target's side, by an operation that enables binding and sets up the adequate data structures.

Binding Objects

To introduce distributed binding, we use the main notions defined in the Reference Model of Open Distributed Processing[8], a general framework for the standardization of Open Distributed Processing (ODP) jointly defined by ITU and ISO [ODP 1995a, ODP 1995b].

[8]We do not attempt, however, to give a full detailed account of this model. In particular, the framework considers five viewpoints according to which the notions may be presented. We are only concerned with the so-called computational and engineering viewpoints, which deal, respectively, with the functional decomposition of a system, and with the mechanisms needed to support distributed interaction between the parts of a system.

The ODP model is based on objects. An object is defined by one or more interface(s), each of which specifies the set of operations that may be performed on the object, together with other properties (e.g. quality of service) that define a contract between the object and its environment. Therefore a binding between objects actually sets up a connection between the objects' interfaces.

In order to provide a uniform view of binding, the ODP model considers the bindings themselves as objects: a *binding object* is an object that embodies a binding between two ore more objects. A binding object has a distinct client or server interface for each object to which it is connected. In addition, a binding may have a specific control interface, which is used to control the behavior of the binding, e.g., as regards the quality of service that it supports or its reactions to errors. The type of a binding object is defined by the set of its interfaces.

Each interface of the binding object acts as an interface to one or several bound objects (e.g., in a client-server system, the interface provided by the binding object to the client is that of the server object, and vice-versa). Figure 3.12 represents an overall view of a binding object.

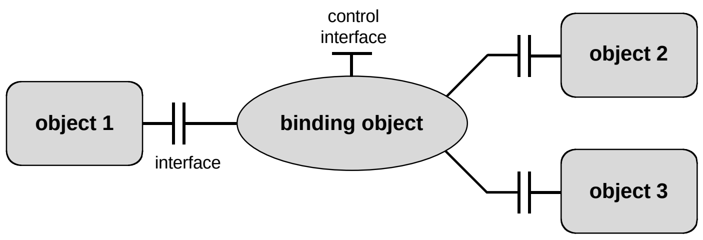


Figure 3.12. A binding object

We have presented the computational view (in the ODP terminology), i.e., a description in terms of functional components. Let us now consider the engineering view, still in ODP terminology, i.e., the implementation of the above mechanism in a distributed setting. The objects to be bound reside on different sites interconnected by a network.

A binding operation (Figure 3.13) typically takes as parameter an unbound reference to an object (the target of the binding), and delivers a *handle* to the object, in the same naming context (recall that a reference is a form of name that refers to the physical location of the object). The reference itself may be obtained through various means, e.g., by looking up a name server using an identifier for the object. The handle may take a variety of forms, e.g., the address of the object, the address of a local copy of the object, or the address of a local proxy for the object, such as a stub.

In some cases, a multiple binding may be set up, i.e., the target may consist of several objects.

The socket example can also be described in this framework, although it is implemented at a lower level than objects. The binding object, in this example, is composed of the client socket, the server socket, and the network connection set up between these sockets.

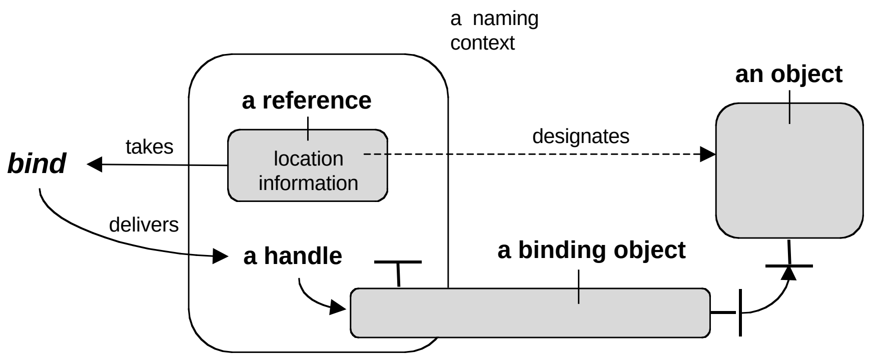


Figure 3.13. The binding process

Binding Factories

The notion of a binding factory has been introduced to define a systematic way of setting up bindings. A *binding factory* is an entity responsible for the creation and administration of bindings of a certain type. Since the implementation of a binding object is distributed, a binding factory may itself be distributed, and usually comprises a set of elementary factories dedicated to the creation of the different parts that make up the binding object, together with coordination code that invokes these factories.

For example, in the simple case of a client-server system using a remote procedure call, the task of the binding factory is split between the stub compiler, which generates client and server stubs using an interface description, the compiler, which compiles the stubs, and the linker, which binds these stubs with the client and server programs, together with the needed libraries. The coordination code consists of a script that calls these tools, using for instance a distributed file system to store intermediate results.

Detailed examples of binding factories may be found in the case studies of the present chapter (3.4) and of Chapters 4, 5 and 8.

3.3.3 The `export-bind` Pattern

The function of a binding factory may be described by a design pattern, which consists of two generic operations called `export` and `bind`.

1. The `export` operation takes as parameters a naming context and an object. Its function is to make the object known in the naming context, by creating a name for it in the context. Since the intent is to allow for a future binding, a side effect of `export` is usually to prepare data structures to be used for binding the object.

2. The `bind` operation takes as parameter a name of the object to be bound. The object must have previously been `export`ed. The operation delivers a handle that allows the object to be accessed. As noted in Section 3.3.2, the handle may take a variety of forms.

The socket example in Section 3.3.2 may be described in terms of this pattern: on the server site, `accept` is an instance of `export`, while on the client site, `connect` is an instance of `bind`. The following two examples give an overview of the use of the pattern for communication and for client-server binding. They are developed in more detail in Chapters 4 and 5, respectively.

Communication Binding

A communication service is implemented by a stack (or acyclic graph) of protocols, each one using the services of a lower-level protocol, down to an elementary communication mechanism. A protocol manages sessions: a session is an abstraction of a communication object, which provide primitives for message sending and receiving.

In our example, The concrete implementation of a session for a client-server communication consists of two objects, one on the server side, the other one on the client side. These objects communicate through a lower level mechanism, which we call a connection (Figure 3.14).

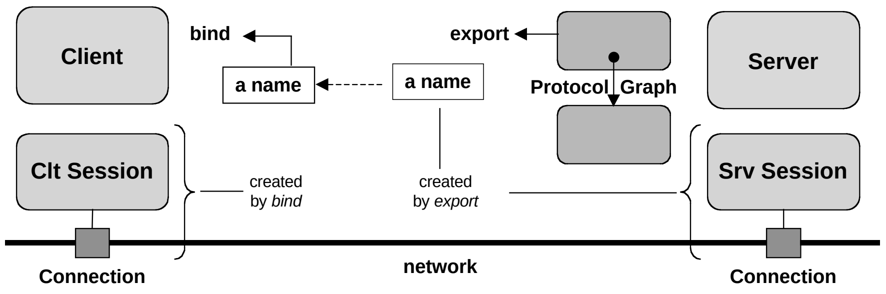


Figure 3.14. Using the `export-bind` pattern for communication binding

The server side session object is created by an `export` operation on the protocol graph, which acts as a session factory. The client uses a distribution-aware name (e.g., a name that contains the address and port number of the server) to create a client-side session object through a `bind` operation. The name may be known by convention, or retrieved from a name server.

Client-server Binding

We present a simplified view of the binding phase of Java RMI, an extension of RPC to Java objects. Java RMI allows a client process to invoke a method on a Java object, the target, located on a remote site. Like RPC, Java RMI relies on a stub-skeleton pair. Binding proceeds as follows (Figure 3.15).

On the server site (a), the target object is exported in two steps.

1. An instance of the target object is created, together with the skeleton and a copy of the stub (to be later used by the client).

2. The stub is registered in a name server under a symbolic name

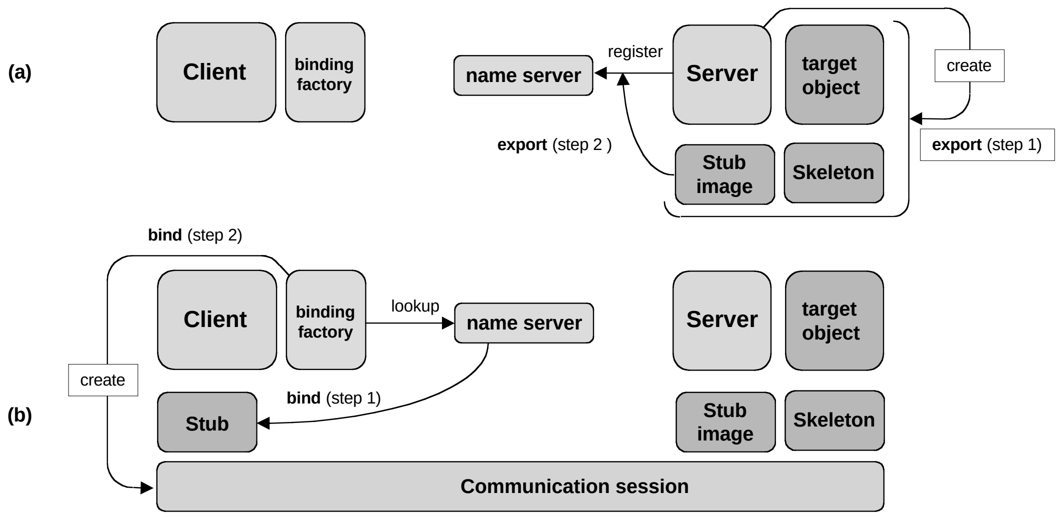


Figure 3.15. Using the `export-bind` pattern for client-server binding

On the client side (b), the client calls a binding factory, which also proceeds in two steps to complete the binding.

1. The stub is retrieved from the name server using the symbolic name.

2. A communication session is created, to be used by the client to invoke the remote object through the stub (the information needed to create the communication session was written into the stub during the `export` phase)

Note that the target object is exported twice: first to a local context on the server site, then to the name server. This is a usual situation; actually `export` is often called recursively, on a chain of contexts.

Also note that a phase is missing here: how the symbolic name is known by the client. This relies on an external mechanism (e.g., naming convention, or explicit transmission from the server to the client, etc.).

The Jonathan binding framework, described in Section 3.4, is based on the `export-bind` pattern. Further detailed examples may be found in Chapters 4 and 5.

Another view of binding may be found in [Shapiro 1994].

3.3.4 Putting it All Together: A Model for Service Provision

We may now present an overall view of the process of service provision, involving three parties: the service provider, the service requester, and the service directory. The process is organized along the following steps (Figure 3.16).

1. *Service creation.* The service provider creates a concrete implementation of the service (this may be a servant object that implements the service interface).

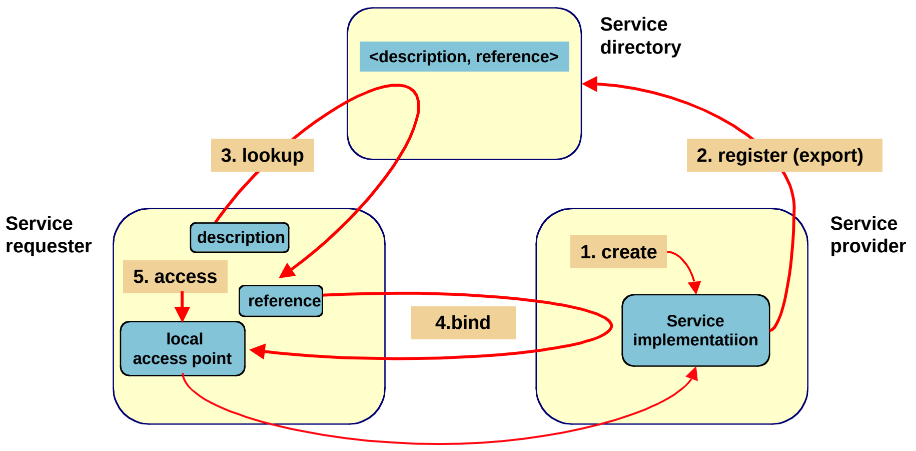


Figure 3.16. Global view of service provision

2. *Service registration.* The service provider registers the service with the service directory, by providing a link to the service location together with a name (or a set of attributes). This step actually implements an `export` operation.

3. *Service lookup.* The requester looks up the directory, using a description of the service. This description may be a name or a set of attributes. If successful, the lookup returns a reference to the service (or possibly a set of references, among which the requester may choose, e.g., according to QoS criteria).

4. *Service binding.* The requester performs the `bind` operation on the reference, thus obtaining a handle, i.e., a local access point to a binding object connected to the server. Depending on the implementation, lookup and binding may be implemented as a single operation.

5. *Service access.* Finally, the requester invokes the service, by a local call through the handle. The binding object forwards the call to the service implementation and returns the result to the caller.

This is a general scheme, which is defined in terms of abstract entities. This scheme is embodied in many different forms (e.g., Java RMI, CORBA, Web Services), in which the abstract entities are have specific concrete representations. Examples may be found in the next chapters.

3.4 Case Study: Jonathan: a Kernel for Distributed Binding

The notions related to naming and binding in an object-oriented environment are illustrated by the example of Jonathan, a framework for the construction of distributed middleware that may embody various binding policies. Jonathan is entirely written in Java and is available, as open source, on the site of the ObjectWeb consortium, at http://forge.objectweb.org/projects/jonathan.

3.4.1 Design Principles

Jonathan [Dumant et al. 1998] is a framework for building Object Request Brokers (ORBs), which are central components of middleware systems. The development of Jonathan was motivated by the lack of openness and flexibility of the currently available middleware systems. In order to facilitate the construction of ORBs adapted to specific run time constraints or embodying specific resource management policies, Jonathan provides a set of components from which the various pieces of an ORB may be assembled. These components include buffer or thread management policy modules, binding factories, marshallers and unmarshallers, communication protocols, etc. In addition, Jonathan includes configuration tools that facilitate the task of building a system as an assembly of components and allow the developer to keep track of the description of a system.

Jonathan is organized in four frameworks, each of which provides the Application Programming Interfaces (APIs) and libraries dedicated to a specific function. The current frameworks are the following.

- *Binding.* The binding framework provides tools for managing names (identifiers) and developing binding factories (or extending available ones). Different inter-object binding models may be managed, allowing for example the use of different qualities of service. This framework is based on the `export-bind` pattern presented in Section 3.3.3.
- *Communication.* The communication framework defines the interfaces of the components implied in inter-object communications, such as protocols and sessions. It provides tools for composing these pieces to construct new protocols. The communication framework is presented in Chapter 4.
- *Resources.* The resource framework defines abstractions for the management of various resources (threads, network connections, buffers), allowing the programmer to implement new components to manage these resources, or to reuse or extend existing ones.
- *Configuration.* The configuration framework provides generic tools to create new instances of components, to describe a specific instance of a platform as an assembly of components, and to create such an instance at boot time.

The main components of Jonathan (i.e., the classes that make up the frameworks) have themselves been developed using uniform architectural principles and a common set of patterns and tools (factories, helpers).

Jonathan includes two specific ORBs ("personalities"), which have been developed using the above frameworks. These personalities are Jeremie (5.4), an implementation of Java RMI, and David (5.5), an implementation of the OMG Common Request Broker Architecture (CORBA). The personalities essentially include implementations of binding factories, together with common services.

3.4.2 Naming and Binding in Jonathan

We first describe the structure of naming contexts, and then present the binding process and the binding factories.

Identifiers and Naming Contexts

Jonathan provides the notions of *identifier* and *naming context*, embodied in the `Identifier` and `NamingContext` interfaces. These notions are closely related: an identifier is created by a naming context, and this naming context remains associated with the identifier. Since Jonathan is a framework for building distributed object-based middleware, the entities designated by identifiers are objects. More precisely, since objects may only be accessed through an interface, an identifier created in a specific naming context is associated with an object type, defined by a (Java) interface. For example, the identifiers created by a protocol (a kind of naming context, see Chapter 4) designate sessions, which are communication objects defined by an interface associated with the protocol.

Note that the identifiers, as defined in Jonathan, do *not* have the uniqueness properties defined in 3.1 (e.g., an object may be designated by several identifiers). Identifiers may usually be viewed as (unbound) references.

To allow the construction of compound contexts (e.g., hierarchical contexts, such as described in Section 3.1.1), identifiers may be associated with chains of naming contexts. Such a chain may be (conceptually) represented as, for instance, `a.b.c.d` in which `a.b.c.d` is an identifier in the first context of the chain, `b.c.d` an identifier in the next context, etc. Note, however, that the identifiers do not necessarily explicitly exhibit this concatenated form, which is only presented here as an aid to understanding. Two operations, `export` and `resolve`, are respectively used to construct and to parse such chains. It should be noted that these operations may be used in a wide variety of situations; therefore, their signature and specification may differ according to the environment in which they appear.

- `id = nc.export(obj, hints)` is used to "export" an object `obj` to the target naming context, `nc`. This operation returns an identifier that designates object `obj` in the naming context `nc`. The initial designation of `obj` may have various forms, e.g., `obj` may be an identifier for the object in a different context, or a low-level name such as a Java reference. The `hints` parameter may contain any additional information, e.g., the name of another context to which `obj` should be exported. In most cases, `export` also has the side effect of building additional data structures that are subsequently used when binding to object `obj` (see Section 3.4.2). The `unexport` operation cancels the effect of `export` and precludes further use of the target identifier.

- `next_id = id.resolve()` is used to find the "next" identifier in a chain. This operation is the inverse of `export`: if `id1 = nc.export(obj, nc1)` exports `obj` to contexts `nc` and `nc1`, then `id1.resolve()` returns `id`, the identifier of `obj` in `nc`.

Using the conceptual concatenated representation, if identifier `id1` is represented as `a.b.c.d`, then `id1.resolve()` returns `id` represented as `b.c.d`, etc. Conversely, calling `nc.export(id1)` where `id1` is represented by the chain `x.y.z` returns an identifier `id` associated with `nc` and represented as `w.x.y.z`. Calling `resolve()` on an identifier that is not a chain returns `null`.

Since identifiers may need to be transmitted on a network, they may have to be encoded into a byte array, and decoded upon reception. The operations `encode` and `decode` respectively perform the encoding and decoding. While encoding is borne by the `Identifier` interface, decoding must be borne by each destination naming context since it returns identifiers that are only valid in that context.

- `byte_array` = `id.encode()` encodes `id` into a byte array `byte_array`.
- `id` = `nc.decode(byte_array)` decodes `byte_array` into an identifier `id` in the target naming context `nc`.

Bindings and Binding Factories

A *binding* is a connection between an identifier and the object that it designates (the target of the binding). A *binding factory* (or *binder*) is a special form of a naming context, which can create bindings. The form of the binding depends on the way of accessing the target from the binder. If they are in the same address space, the binding may take the form of a simple reference; if they are in different address spaces, the binding usually involves intermediate objects (known as proxies, delegates, or stubs), and possibly one or several communication objects if the target is remote.

Consider an identifier `id` in a naming context. If the naming context is also a binder, then a binding may be set up by invoking `id.bind()`, possibly through a chain of binders. If not, then the identifier may be `resolved`, returning another identifier associated with another context, and the resolution is iterated until the naming context associated with the identifier is a binder (Figure 3.17). An identifier that may neither be resolved nor bound is said to be *invalid* and should not be used.

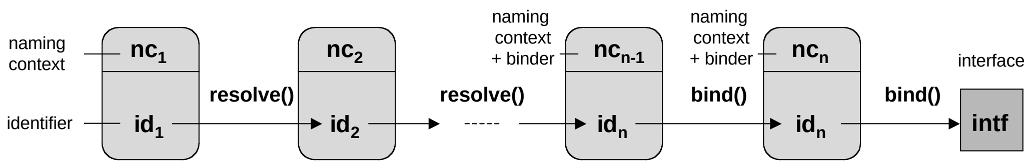


Figure 3.17. Resolving and binding identifiers in Jonathan

If the naming context of an identifier `id` is also a binder, then

`s = id.bind()`

returns an object `s` (the handle) through which the target of the binding may be accessed (the interface of `s` is therefore conform to that of the target). The target is an object designated by the identifier. The returned object `s` may be the target itself, but it is usually a representative of the target (a proxy). A special form of a proxy is a stub, which essentially holds a communication object to be used to reach the target. The `bind` operation may have parameters that specify e.g., a requested quality of service.

In order for `bind` to work, the target must have previously been `export`ed. The `export-bind` pattern is central in the Jonathan binding process and is applied in a wide variety of situations, e.g. communication protocols or remote object invocation.

3.4.3 Examples

The main examples illustrating the use of the `export-bind` pattern may be found in Chapters 4, 5, and 8.

In this section, we present two simple examples which will be used in more elaborate constructions.

Minimal Object Adapter

An *adapter* (further discussed in Chapter 5) is used to encapsulate an object in order to use it with a different interface, using the *Wrapper* design pattern (2.3.3). In the present case, the adapter is used on the server side of an ORB to encapsulate a set of servants. A servant is an object that implements a specific service. Requests arrive to the server in a generic form, such as `invoke(obj_ref, method, params)`, where `obj_ref` is a name which designates an object managed by the server. The function of the adapter is to convert this request into an invocation on a specific servant, using that servant's interface. This operation may be fairly complex, as explained in Chapter 5.

In this example, the adapter (called `MinimalAdapter`) is reduced to it simplest form: a binder that manages a set of servants, implemented as Java objects. Let `adapt` be a instance of `MinimalAdapter` and `serv` a servant. Then the operation `adapt.export(serv, hints)` creates a name `id`, of type `MoaIdentifier`, which designates the object `serv` in the context of the adapter. The parameter `hints` may be used to specify another naming context to which the object may be exported, thus allowing recursive exportation. The operation `adapt.unexport(id)` cancels the effect of `export` by disconnecting `id` from the object that it designates.

The operation `id.bind()`, where `id` is a `MoaIdentifier`, returns a reference on a servant object (in this specific implementation, a Java reference), if the name `id` is actually bound in the adapter.

In this example, the adapter is essentially a table of Java objects, implemented as a set of holders managed through a hash-coding scheme. Each holder contains a couple (identifier, object reference). The implementation includes some optimizations: there is a single holder pool for all adapter instances; holders that have been freed (by `unexport`) are kept on a reusable list, which reduces the number of holder creations. In order to prevent the holder table from being garbage collected (since no permanent reference points to it), a waiting thread is associated with the table when the first holder is allocated and disconnected when the last holder is freed.

When a new object is exported, a reusable holder is selected; if none is available, a new holder is created. In both cases, the holder is filled with a new instance `id` of `MoaIdentifier` and the reference of the target object. If the `hints` parameter specifies a naming context `nc`, the object is also exported to that naming context. Its identifier in this context, `id_nc,` is returned (and also copied in the holder). The `unexport` operation returns the holder to the reusable list and calls `unexport` on `id_nc` (thus recursively unexporting the object from a context chain).

The `bind` operation on a `MoaIdentifier` tries to find a matching holder using the hash code. The operation returns the object reference if a match is found, and returns `null` otherwise.

Domain

We now consider the implementation of a *domain* in Jonathan. Domains are used to solve the following problem arising in distributed computing. When an identifier is transmitted over a network, it is encoded (marshalled) on one side and decoded (unmarshalled) on the other side. In order to decode an identifier, one needs to know the naming context in which it has been encoded (because `decode` is borne by that context). If multiple protocols coexist, each protocol defines its own naming context and a mechanism is needed to "pack" an identification of this context together with an encoded identifier.

For example, let `id` be an identifier to be sent over a network, and let `nc` be its naming context. Then it is not enough to send the value `encoded_id = id.encode()` because `nc` is needed, at the destination site, to retrieve `id` by calling `nc.decode(encoded_id)`.

The solution consists in using a domain as a naming context, available on all sites, that identifies other naming contexts and wraps their identifiers in its own identifiers. `JDomain` is the standard implementation of a domain in Jonathan.

Let us consider again the problem of sending `id` (of naming context `nc`) over a network (Figure 3.18). Instead of `id.encode()`, the value actually sent is `encoding = jident.encode()`, where `jident = JDomain.export(id)`. `jident` is of type `JId` (the identifier type managed by `JDomain`). At the receiving end, `jident` is first retrieved as `JDomain.decode(encoding)`, and `id` is finally obtained as `jident.resolve()` (recall that `resolve` is the inverse operation of `export`).

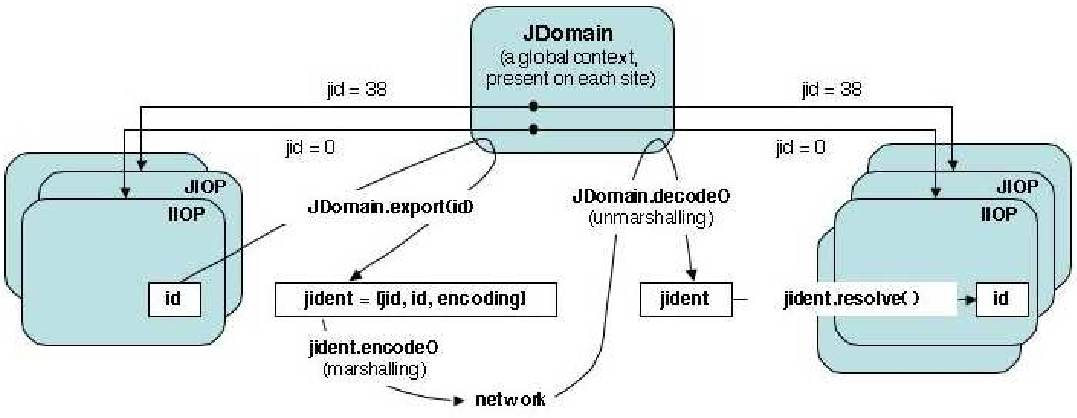


Figure 3.18. Transmitting an identifier over a network

We describe the implementation of `JDomain` in some detail, because it provides a good practical illustration of the main concepts related to identifiers in Jonathan. The inner working of `JDomain` is quite simple: each different naming context is associated with an integer (e.g., the naming context of the IIOP protocol is associated with 0, etc.). A `JId` encapsulates the identifier `id`, the integer value `jid` associated with the identifier's context, and an `encoding` of the pair (`jid, id`). Depending on how it was generated, a `JId` may actually contain (`jid, id`) or (`jid, encoding`). As is shown below, `id` can be generated from `encoding` and vice versa, if `jid` is known.

The main data structure is a linked list of holders (Figure 3.19). A holder contains an integer value (`jid`), a reference to a naming context, and the type (i.e., the name of the Java class) of this context. When a domain is created, this list is empty. An initial context (whose value is specified in the configuration phase) describes the predefined associations between binder classes and integer values.

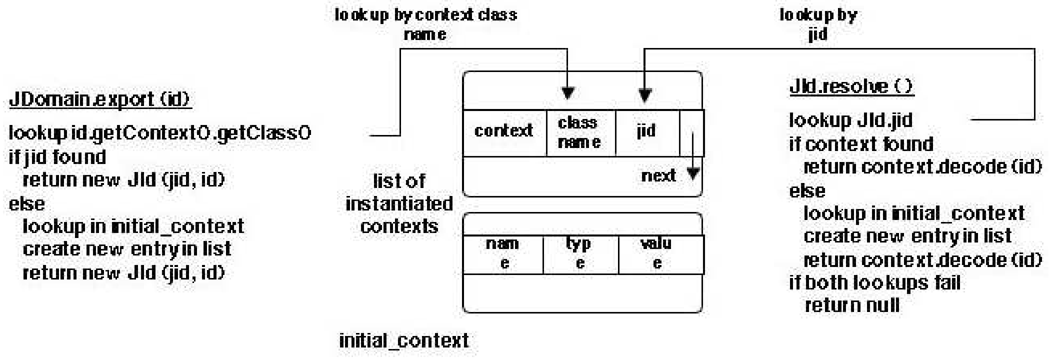


Figure 3.19. The implementation of a domain in Jonathan

A summary of the main operations follows.

- `export`. Exporting an identifier `id` to `JDomain` results in "wrapping" it into a `JId`. When `id` is exported, the context list is looked up for an entry (a holder) containing `id`'s class name in order to retrieve the corresponding `jid`. If no such entry is found, the initial context is looked up for a value of `jid`, and a new entry (`id`'s context, `id`'s class name, `jid`) is created in the list.

- `resolve`. Calling `resolve` on a `JId` returns `id`, which is found either directly or by decoding the encapsulated `encoding`. In the latter case, the context of `id` must be retrieved in order to call `decode`. This is done by looking up the context list using `jid` as a key; if this fails, the initial context is looked up and a new entry (`id`'s context, `id`'s class name, `jid`) is created in the list.

- `encode`. Calling `encode` on a `Jid` returns the value of the encapsulated encoding; if this value is null, an encoding of (`jid, id`) is generated (prefixing `id`'s encoding, generated by `id`'s encode method, with a 4-byte encoding of `jid`).

- `decode`. Two situations may occur. In the simplest case, the parameter of `decode` is a buffer containing the `encoding`. In that case, the value of `jid` is decoded from the first 4 bytes of `encoding` and a new `JId` is created with (`jid, encoding`). In the other case, `decode` is called on an unmarshaller, i.e., data delivered by a communication protocol; the data consists of `jid` and `encoding`. In that case, the identifier has been encoded by a remote system and a local naming context must be found to decode it. This is done by looking up the context list using `jid` as a key; if no context is found, then again the initial context is looked up.

- `bind`. This operation is provided for completeness, because it is not needed for identifier transmission. The operation `JDomain.bind(jid)` returns `nc`, the naming context associated with id.

An illustration of the use of `JDomain` for marshalling and unmarshalling identifiers is given in Chapter 5.

3.5 Historical Note

Names (be it in the form of identifiers or addresses) have been used since the very beginning of computing. However, a unified approach to the concepts of naming and binding in programming languages and operating systems was only taken in the late 1960s and early 1970s (e.g., [Fraser 1971]). A landmark paper is [Saltzer 1979], which introduces dynamic binding, drawing on the experience of the Multics system. Early work on capabilities [Dennis and Van Horn 1966, Fabry 1974] links naming with protection domains.

Distributed naming and binding issues are identified in the design of the Arpanet network [McQuillan 1978], and an early synthetic presentation of naming in distributed systems appears in [Watson 1981]. Later systematic presentations include [Comer and Peterson 1989], [Needham 1993], and Chapter 4 of [Tanenbaum and van Steen 2006].

In the early 1980s, with the development of networking, the issue of scale becomes central. Grapevine [Birrell et al. 1982], developed at Xerox PARC, introduces the main concepts and techniques of a scalable naming system: hierarchical name space, replicated servers, caching. Other advances in this area include the Clearinghouse [Oppen and Dalal 1983], GNS [Lampson 1986], and the design described in [Cheriton and Mann 1989] (but available in report form in 1986). At that time, the redesign of the Internet (then Arpanet) naming service is underway, and the Domain Name System [Mockapetris and Dunlap 1988] benefits from the experience acquired in all of these projects.

In the context of the standardization efforts for network interconnection, the X.500 standard [X.500 1993] is developed. Its main application is LDAP (Lightweight Directory Access Protocol) [LDAP 2006], a model that is widely used for enterprise directory services.

Trading systems use attributes, rather than names, as a key for finding distributed services and resources. Notable advances in this area are described in [Sheldon et al. 1991] and [van der Linden and Sventek 1992].

In the 1990s, flexibility, dynamism and adaptability are recognized as important criteria for open distributed systems. The issue of dynamic binding is reexamined in several research efforts (e.g., [Shapiro 1994]). The RM-ODP standard [ODP 1995a] defines a framework for flexible binding, which stimulates further research in this area, among which Jonathan [Dumant et al. 1998], Flexinet [Hayton et al. 1998], OpenORB [Coulson et al. 2002].

The trend toward flexible and adaptable naming and binding is amplified in the early 2000s by the wide availability of wireless networks and the advent of Web services. The notion of a service discovery service combines trading (search by attributes), fast dynamic evolution, flexibility, and security. There is no established standard in this area yet, but several research and development efforts (e.g., [Czerwinski et al. 1999, Adjie-Winoto et al. 1999, Guttman 1999, Waldo 1999]) have helped to identify relevant design principles and implementation techniques.

Chapter 4

Communication

Middleware relies on an underlying communication service, usually at the transport level. Providing communication services is also the main function of some middleware systems, which supply a higher level communication interface to applications. This chapter presents an architectural view of communication systems: how a communication system is constructed by combining more primitive ones, using uniform patterns. The main communication paradigms are first introduced, followed by a brief discussion of the main characteristics of communication systems. Then comes an introduction to the internal organization of a communication system, and a discussion of some underlying construction patterns. The chapter concludes with a description of the communication framework of Jonathan, an experimental open source communication toolkit, illustrated by simple use cases.

4.1 Introducing Communication Systems

In its simplest form, communication is the process of transmitting information between two entities: a *sender* and a *receiver*. A more general form of communication (broadcast and multicast), to be examined later, involves a sender and several receivers. In middleware systems, senders and receivers are usually activities (processes or threads) executing in a specified context (a machine, an application, an operating system, etc.).

The transmitted information is called a *message*; it may be as simple as a single bit (the occurrence of an elementary state transition), or as complex as an elaborate flow of data subject to timing constraints, such as a video stream.

Communication relies on a physical process whose interface is expressed in terms of actions on some physical medium. At the application level, communication is more conveniently viewed as a *service*, whose interfaces include high-level communication primitives. These interfaces may be specified in terms of messages, represented as data structures defined in application-specific terms. They also may have a more abstract form, an example of which is RPC (1.3), which encapsulates elementary message interchange within a high-level communication interface.

A *communication system* is the combination of hardware and software that provides a communication service to applications. In order to bridge the gap between the physical and application levels, a communication system is usually organized as a set of layers

(2.2.1), in the form of a *protocol stack*. Each protocol specifies and implements the rules that govern communication at a given level; it defines its own communication abstractions and relies on the communication API provided by the lower level protocol in order to implement its own service, down to the physical level. Thus, at each level, communication may be seen as taking place over an *abstract channel*, which relies on (and hides) all the protocol layers below that level. Protocols and protocol stacks are discussed in detail in section 4.3.

At this stage, we only need to point out one aspect of this organization. A particular level may be chosen as the "base level" upon which more abstract upper layers are developed.

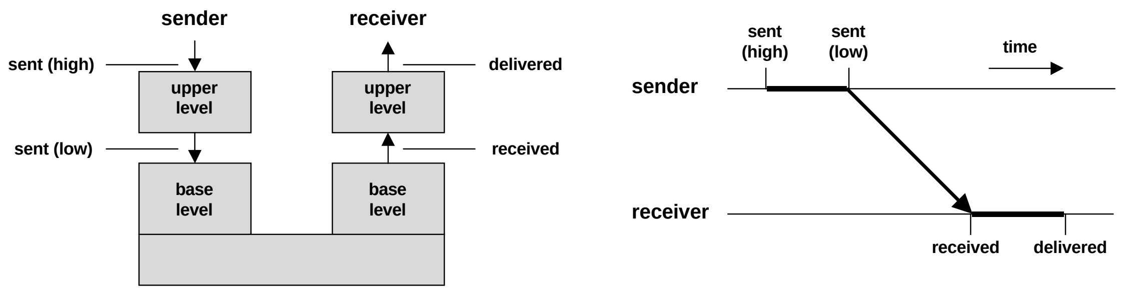


Figure 4.1. A view of a layered communication architecture

We can then introduce the notion of *delivery* of a message (at the upper layer interface) as distinct from its *receipt* (at the base layer interface). For example, the upper layer may reorder messages, so that the order of delivery may differ from the order of receipt; or it may drop duplicate messages. A similar distinction may be made at the sender end.

The base level and its interface are chosen to reflect the current interest: if the interest is in middleware, then the base level is usually the transport interface; if the interest is in the complete implementation of the transport protocol, then the base level may be chosen as the physical interface.

In the rest of this section, we examine the main characteristics that define a communication service, introducing them progressively. We concentrate on *specification*, leaving the architectural aspects for the next sections. The notions presented apply at any level of the protocol stack: communication takes place over the abstract channel implemented at that level.

We start with the simple case of an isolated message, assuming reliable transmission (4.1.1). We then assume the occurrence of failures, and we consider the constraints implied by multiple messages and multiple receivers (4.1.2). Other requirements (quality of service and security) are examined in section 4.2.

4.1.1 Single Message, Reliable Channel

We consider the sending of a single message between one sender and one or more receivers, and we assume that the communication is reliable, i.e., the message is delivered uncorrupted to all of the receivers.

In order to characterize the properties of communication, we need a classification framework. Various classification criteria have been proposed (e.g., [Tai and Rouvellou 2000], [Tanenbaum and van Steen 2006, Chap. 2], and [Eugster et al. 2003]). The framework that we propose borrows elements from these sources. It takes three aspects of communication into account.

1. *Designation of receivers.* The designation may be *explicit*, i.e., the receivers of a message are directly designated by their names, or *implicit*, i.e., the set of receivers is determined by some other criterion (which may depend on the contents of the message). More details on multiple receivers are given in 4.1.2 and in Chapter 6.
2. *Message persistence.* The message may be *transient*, i.e., it is lost if no receiver is ready to receive it, or *persistent*, i.e., the communication system keeps the message until it is delivered to it receiver(s).
3. *Synchronization.* The communication may be *blocking*, i.e., the sender thread or process is blocked until a response comes from the receiver (the nature of the response is elaborated further below), or *non-blocking*, i.e., the sender continues execution after the message has been sent[1].

For each aspect, these options define different degrees of coupling between the sender and the receiver, as summarized in Table 4.1.

Coupling	strong	weak
Designation	*explicit*	*implicit*
Persistence	*persistent*	*transient*
Synchronization	*blocking*	*non-blocking*

Table 4.1. Degree of coupling in communication

There may be additional variations within these main classes. For instance, blocking message sending may have three possible meanings, for a single receiver, in order of increasing coupling: the sender is blocked until it receives notice that either a) the message has been received on the receiver's site; b) the message has been delivered to the receiver; c) the message has been processed by the receiver (assuming an answer is actually expected). In the case of multiple receivers, there are many more combinations, from the weakest (the message has been received on one receiver's site) to the strongest (the message has been processed by all receivers).

The strongest degree of coupling, for a single receiver, is represented by the combination (*explicit*, *persistent*, *blocking-c*), in which a message is sent to a specified receiver, the message persists until delivered, and the sender is blocked until the message has been answered. The weakest coupling is (*implicit*, *transient*, *non-blocking*), in which a message is sent to an unspecified set of receivers (the sender does not need to know the selection criteria for receivers), the message is transient (it is lost if no receiver is ready) and the sender continues execution after having sent the message.

[1]The terms *synchronous* (resp. *asynchronous*) are often used in place of *blocking* (resp. *non-blocking*). However, these terms have multiple meanings, and we use them in another context (4.1.2).

Strong coupling tends to build a communication pattern close to that of centralized computing. Examples of strongly coupled communication services are RPC (1.3) and the various object invocation primitives described in Chapter 5. Weak coupling, by contrast, tends to favor independence and late binding between sender an receivers. Examples of weakly coupled services are events, messages, and other coordination mechanisms described in Chapter 6.

4.1.2 Multiple Receivers and Messages, Unreliable Channel

We now consider a more general situation, in which multiple senders and receivers communicate over a channel subject to failures. We identify three subproblems.

1. How to build a reliable channel on top of an unreliable one.
2. How to characterize the timing properties of communication.
3. How to specify the properties of communication involving multiple senders and receivers.

We discuss these points in turn.

Building a Reliable Channel

Messages may be corrupted or lost, due to three main causes: (i) transmission errors at the physical level, due to noise or signal weakening; (ii) failures or overload conditions in communication software (e.g., packet loss in overloaded routers); and (iii) accidental cuts or disconnection, which break the physical connection between sender and receiver.

Fault tolerance mechanisms implemented at the lower levels of the protocol stack ensure that a message is delivered to its receiver as long as there exists a working physical communication link between the sender and the receiver. These mechanisms are based on redundancy, both in space (including additional bits in the message for error detection or correction) and in time (resending a message if loss or corruption has been detected).

The global effect of these mechanisms is to provide a *reliable channel*. Consider two processes, A and B, connected by such a channel; assume that A and B do not fail, and that the physical connection between them is preserved (i.e., restored after a cut or a disconnection). Then the following properties hold.

- If A sends a message m to B, then m will be delivered to B.
- A message m is delivered only once to a receiver B, and only if some process sent m to B.

This essentially says that a message is eventually delivered unaltered to its receiver and that the channel does not generate spurious messages or duplicate actual messages. However, there may be no guarantee on the time needed to deliver the message. This is the subject of the next subsection.

Timing and Ordering Properties of Communication

A communication system is *synchronous* if there is a known upper bound on the transmission time of an elementary message[2]. If no such bound is known, the system is *asynchronous*. This property is essential in a distributed system, because it allows the use of timeouts to detect the failure of a remote node, assuming the channel is reliable. In an asynchronous system, it is impossible to distinguish a slow processor from a faulty one, which leads to a number of impossibility results for distributed algorithms in the presence of failures.

Unfortunately, most usual communication systems (e.g., the Internet) are asynchronous, because they rely on shared resources (e.g., routers) on which the load is unpredictable. In practice, many implementations of distributed algorithms use timeouts based on an estimated upper bound for message transmission time, and then must deal with late messages, which may arrive after the timeout.

Some applications have hard real-time constraints (i.e., a late message causes the application to fail), and therefore require a guaranteed upper bound on transmission time. This is usually achieved by resource reservation (see 4.2.1).

Some applications need to be aware of physical time. If the application is distributed, the physical clocks of the different nodes must then be synchronized, i.e., the drift (time difference) between the clocks at any two sites must be bounded. If the absolute time is relevant, then the drift between any local clock and an external time reference (a time server) must also be bounded. This is achieved by clock synchronization algorithms (see e.g., [Coulouris et al. 2005], section 11.3).

Even if an application has no constraints related to physical time, the relative order of events is important, since process synchronization is expressed by constraints on the order of events. Events on a site may be ordered by dating them with a local clock. Event ordering on different sites relies on communication, and is based on the causality principle: for any message m sent from process A to process B, the event "sending m" on A precedes the event "receiving m" on B. Based on this remark, one defines a system of logical clocks (first introduced in [Lamport 1978b]), which preserves this ordering property, as well as local order on each site. These clocks capture a causal precedence relationship, noted $\rightarrow$ (or *happens before*). For any two events e and e', $e \rightarrow e'$ means that e is a potential cause of e', while e' cannot be not a cause of e. More elaborate ordering systems have been designed to capture causality more closely (see e.g., [Babaoğlu and Marzullo 1993]). Thus communication is a basic mechanism for event ordering in a distributed system.

Broadcast and Multicast

We now turn to the definition of communication involving multiple receivers. We start by defining a *process group* as a set of related processes associated with a set of protocols for managing group membership (e.g., joining or leaving the group, determining the current composition of the group) and communication. The main motivations for process groups are the following.

[2]When considering a distributed system (i.e., a set of processors connected by a communication system), synchrony also implies that there is a known bound on the relative speed ratio of any two processors.

- Defining a set of processes that behave like a single (reliable) process, i.e., any process of the group may replace another one if this latter fails. This implies that all the members of the group have a consistent view of the system's state.
- Defining a set of processes that share common privileges (e.g., for access to information), and that may participate in collaborative work.

Two main operations are defined for group communication: broadcast and multicast. Both involve a single sender an several receivers.

- In *broadcast*, the receivers are the processes that belong to a single set, implicitly defined (e.g., the processes that are members of a specified group; all the processes in the system). The sender is also a member of that set.
- In *multicast*, the receivers are the members or one or several process groups. These groups may have common members. The sender may or not belong to the set of receivers.

Process groups and group communication are examined in more detail in Chapter 11 (see 11.3.2, 11.4). Specific techniques for large scale broadcast are discussed in 11.4.4

A number of properties may be specified for group communication[3]. They essentially deal with the behavior of the communication system in case of failures. Here we assume that the group communication protocols are built over a reliable channel, and that the processes may fail. We assume *fail-stop* failures, i.e., a process either behaves according to its specification, or is stopped and does nothing (see 11.1.3). A *correct* process is one that does not fail.

The weakest property is *reliable delivery*, also called "all or nothing": if a message is delivered to a correct process, then it is delivered to all correct processes. A broadcast or multicast system that does not guarantee reliable delivery is not of much practical use. For instance, if the members of a process group are used to update multiple copies of a database, an unreliable broadcast protocol may cause inconsistency (divergent copies) in case of failure, even if the relative order of updates is irrelevant. Reliable broadcast can be implemented even if the underlying reliable channel is asynchronous (see e.g., [Hadzilacos and Toueg 1993]).

When several messages are issued by a sender, the order of delivery is a relevant factor. We need to consider the order of delivery with respect to the order of sending, and the relative order of delivery in the case of multiple receivers. We note $sent(m)$ (resp. $delivered(m)$) the event of sending (resp. delivering) message m.

FIFO delivery means that the messages are delivered to a receiver in the order in which they have been sent. Causal delivery means that the order in which messages are delivered respects causality, i.e., if $sent(m1) \rightarrow sent(m2)$, then $delivered(m1) \rightarrow delivered(m2)$. FIFO delivery is also causal.

Atomic (or totally ordered) delivery means that for any two messages *m1* and *m2* that have a set of common receivers, *m1* and *m2* are delivered in the same order to all of

[3]We only give an informal specification. An accurate and unambiguous specification of group communication properties is a delicate task, and is outside the scope of this discussion. See [Chockler et al. 2001].

these receivers. Note that the atomicity property is orthogonal to the causal and FIFO properties (i.e., FIFO or causal delivery may or may not be atomic).

It is impossible to implement atomic broadcast or multicast (by a deterministic algorithm) over an asynchronous communication system. The methods used in practice, based on the notion of “imperfect” failure detectors, rely on timeouts and therefore need to relax the asynchrony hypothesis (see Chapter 11).

Physical time constraints on message delivery are important in applications involving control operations and multimedia transmission. These aspects are examined in 4.2.1.

The rest of this chapter is organized as follows. Section 4.2 gives an overview of two application requirements: quality of service, availability, and security, and of their impact on network protocols. Section 4.3 describes the organization of a communication system as a protocol graph. Section 4.4 examines middleware and application-level protocols. Section 4.5 describes frameworks for the construction of communication systems. A case study of one such framework is presented in Section 4.6. Finally, Section 4.7 gives a brief account of the history of communication systems.

4.2 Application Requirements

The range of services offered by the new networked applications is expanding, and these services tend to pervade new areas of activity. As a consequence, service providers must satisfy new requirements. Service specifications are expressed in a service level agreement (see 2.1.3), to which the provider must comply. In order to do so, the application that implements the service must in turn rely on *predictable* properties of the communication infrastructure. We briefly examine three main classes of properties: performance (section 4.2.1), availability (section 4.2.2), and security (section 4.2.3).

4.2.1 Quality of Service

The applications that use a communication system may have specific requirements for the performance of the communication service. These requirements take two main forms: *performance assurance*, i.e., guarantees on the absolute value of some performance indicators, or *service differentiation*, i.e., guarantees on the relative treatment of different classes of applications. The general term *Quality of Service* (QoS) refers to the ability of a communication system to provide such guarantees.

The Internet was initially designed for “traditional” applications, such as mail, file transfer, remote login, etc., which do not have strict timing constraints. Its basic communication protocol, IP (4.3.2), provides best effort delivery, offers no performance guarantee, and gives uniform treatment to all its traffic. The need for performance assurance stems from the advent of time-critical applications, such as multimedia content delivery or real time control. Service differentiation aims at giving a privileged treatment to some classes of applications, whose users are willing to pay a higher price for a better service.

Performance assurance relies on resource allocation. An application that has specific performance constraints must first define the corresponding resource requirements, and then attempt to reserve resources. To this end, several reservation protocols are available;

they differ by the level of guarantees that they provide. Reservation protocols are easier to implement on a private (e.g., company-wide) network than on the Internet, since resource allocation is more readily controllable.

Differentiated services are achieved by dividing the traffic into a number of classes, each of which is subject to specific constraints as regards its resource provisioning by the network. Again, the firmness of the guarantees provided to each class depends on the global resource allocation policy, and is easier to achieve on a network operated by a single authority.

A detailed study of the architectures and mechanisms used to provide QoS guarantees on the Internet may be found in [Wang 2001]. Application-level QoS is examined in Chapter 12.

4.2.2 Availability

For a system (or an application) observed during a certain period, *availability* is the fraction of the time the system is ready to provide its service. An increasing number of activities demand an availability rate close to 100%. Availability depends on two factors: the mean time between failures (a *failure* is an event that prevents the service from being correctly delivered); and the mean time needed to restore the correct operation of the application after a failure has occurred (see 11.1.1).

The probability of occurrence of failures may be reduced by various preventive measures, but it is a fact of experience that failures will occur in spite of all such measures. Therefore a system must be designed to operate in the presence of failures. Redundancy is the universal tool used to meet this goal. For a communication system, redundancy may be achieved by several means (as seen in 4.1.2):

- By providing alternate paths from a source to a destination in a network.
- By requiring the recipient of a message to acknowledge receipt, so the message may be resent if lost or corrupted. This implies that a sender must keep a copy of unacknowledged messages, and must estimate an upper bound of the round trip time for message transmission.
- By providing redundancy in the messages themselves, allowing a transmission error to be detected or corrected.

Redundancy may be provided at several levels of the protocol hierarchy. According to the "end to end principle" [Saltzer et al. 1984], mechanisms for recovery at the intermediate levels are redundant with those provided at the application level and might be eliminated. This assumes that failures at the lower levels are relatively infrequent, since recovery at application level is more costly (e.g., retransmitting a whole file instead of a single packet); in that case, the performance optimization provided by recovery at the lower levels is marginal. The end to end principle is therefore less relevant in situations where the failure rate is high (e.g., wireless communication), or recovery at the application level has a high relative cost (e.g., broadcast).

Application-level availability is examined in Chapter 11.

4.2.3 Security

Security is the quality of a system that allows it to resist attack by a malicious party. Thus security related properties may only be defined by reference to the various classes of attacks that the system should withstand.

Taking this approach, the following security properties may be identified for a communication system:

- *Confidentiality*, i.e., resisting an attempt to obtain information that should be kept secret. This information may be the contents of a message, the identity of the communicating parties, and even the fact that two parties do communicate.

- *Integrity*, i.e., resisting an attempt to alter information. This may take many forms: deleting messages, modifying the contents of legitimate messages, injecting spurious messages, faking the identity of a legitimate party.

- *Access protection*, i.e., resisting an attempt to get access to a restricted service, e.g., a private network.

- *Service preservation*, i.e., resisting an attempt to deny access to communication services to legitimate users. Service denial may again take several forms: flooding the network or specific hosts with messages, penetrating communication equipment and modifying its software.

Other properties may be derived from these: for example, *authentification*, ensuring that a party (a person, an organization, a machine) is actually what it claims to be. Authentification is used by algorithms that implement confidentiality, integrity and access protection.

Consider a protocol that implements a communication channel at some level of the hierarchy. Then one or more of the above security features may be implemented by a protocol based on that level, making the channel secure, in a specific sense defined by the set of selected features. All levels above that one now benefit from the security properties of the secure channel.

In practice, secure channels are implemented at the application level, as each application defines its own security requirements (another instance of the end to end principle). However, security protocols have also been defined at the network level, and may become common in future versions of the Internet.

Security in middleware is the subject of Chapter 13.

4.3 Communication Systems Architecture

In this section, we present an overview of the architecture of communication systems. This is intended to be a summary, giving the application developer's view, rather than a detailed study of networking technology. We first introduce the notion of a protocol, and then present the function and interfaces of the transport protocol, the base of the vast majority of middleware systems.

4.3.1 Protocols and Layering

As explained in 4.1, a communication system is usually organized as a hierarchy of layers. The term *protocol* refers to (a) a set of rules that apply to communication at a certain level of the hierarchy; and (b) an implementation of such set of rules. Above the base level, a protocol relies on the underlying protocols for its operation, using one of the variants of the Layers pattern described in 2.2.1.

There are two views of a protocol.

- The external view (the user's view), which defines the interface (API) that the protocol provides. At the current level, the interface specifies the available abstractions (session, message, etc.), the available primitive operations (*open*, *send*, *receive*, etc.), and the rules of usage (e.g., a session must be opened before a message may be sent or received, etc.). A protocol may thus be seen as the definition of a specific language.

- The internal view (the implementer's view), which defines the internal operation of the protocol, i.e., how its abstractions and operations are implemented in terms of the APIs provided by the lower layers in the protocol stack.

At a given level, a user needs only be aware of the API and abstractions defined at that level. The user's view is that of a "horizontal" communication that takes place at the current level. However (except at the physical communication level), this view is virtual: actual communication involves the protocol layers below the current level. Typically, a message sent is propagated down the hierarchy and ultimately transmitted at the physical interface. At the receiving end, the message is propagated up the hierarchy, till the level of the initial send. This is illustrated on Figure 4.2.

The processing of a message by a layer may involve adding some information in the downward propagation phase (e.g., redundancy check for error detection or correction), and removing this added information in the upward propagation phase after it has been exploited. Thus a message sent up from a given level at the receiving end is identical to the corresponding message that entered that level at the sending end.

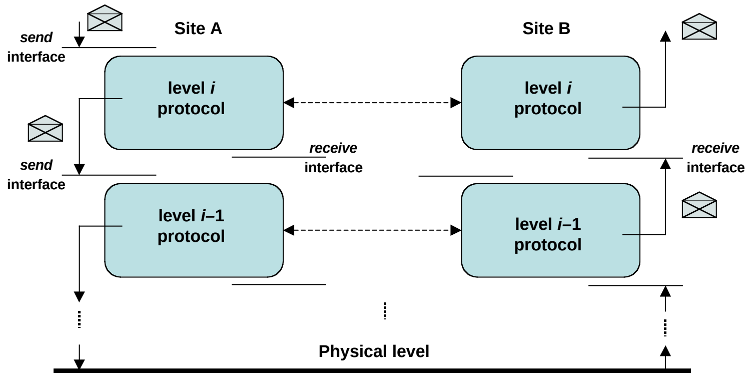


Figure 4.2. Protocol layering

There is a fundamental difference between the *send* and *receive* operations: *send* is usually synchronous, while *receive* is asynchronous, i.e., the receipt of a message triggers an asynchronous event. Thus the message flow in a layered communication system follows the Half Sync, Half Async pattern (2.2.1). This pattern is illustrated by the example presented in 4.6.1.

Communication between two users A and B may take two forms.

- *Connected*. In order to exchange messages, A and B must first set up a *session*, i.e., a channel that provides send and receive primitives. A session usually guarantees some properties, such as message ordering, reliable communication, and flow control. After a session is closed, no further messages may be exchanged.

- *Connectionless*. Messages may be exchanged without any preliminary operations. Messages are independent of each other, and there is usually no support for message ordering or quality of service.

These notions may be extended to communication involving more than two users.

At a given level, a number of different exchanges may take place at a given time (e.g., a user may have started several sessions with different partners, and may also send and receive messages in connectionless mode. Thus the lower levels must ensure message multiplexing and demultiplexing, e.g., directing an incoming message to the right session or to the right recipient.

A more general organization is that of a *protocol graph*, in which a protocol at level i may use any protocol at a level $j < i$, and not only a protocol at level $i - 1$. The graph is thus acyclic, and protocols may be shared.

4.3.2 The Protocols of the Internet

The protocols used on the Internet illustrate the above notions. The Internet is an interconnection of networks of various kinds and sizes, using a variety of physical communication links. At the upper end, applications are organized in levels, and protocols carrying out common functions are shared by several applications.

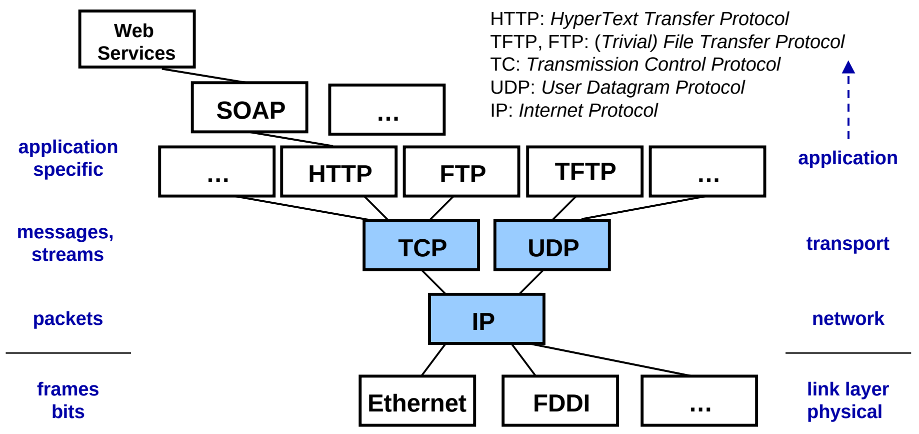


Figure 4.3. The Internet protocol stack

The key of the success of the Internet is the organization shown on Figure 4.3: a single protocol, the Internet Protocol (IP) provides a common means for transferring packets (fixed size chunks of bits) from one node to another one, regardless of the underlying physical infrastructure and specific protocols. The level at which IP operates is called the *network* level. At this level, the machines (or *hosts*) connected to the various networks are designated by IP addresses, a naming scheme that uniquely identifies a (host, network) pair. The networks are interconnected by communication devices called *routers*, which implement the IP protocol by forwarding the packets to their destination. This forwarding function is based on routing tables, which are periodically updated to reflect the current traffic and link availability conditions.

At the next level up (the *transport* level), two protocols use IP to provide basic communication facilities to applications. UDP (User Datagram Protocol) is a connectionless protocol that allows single message exchange with no guarantees. TCP (Transmission Control Protocol) is a connection-oriented protocol that allows the bidirectional transfer of byte streams with order preservation, flow control, and fault tolerance capabilities.

TCP and UDP are called transport protocols. Contrary to IP, they are end to end protocols, i.e., the intermediate nodes used for communication between two hosts are not visible. End points for communication on each host are identified by *port numbers*; this allows a host to be engaged in several different exchanges. In addition, port numbers are used to identify a service on a server; fixed port numbers are allocated, by convention, to the most common services, such as *mail*, *telnet*, *ftp*, etc.

Most applications use TCP and UDP through *sockets*, a common interface provided by current operating systems (Figure 4.4).

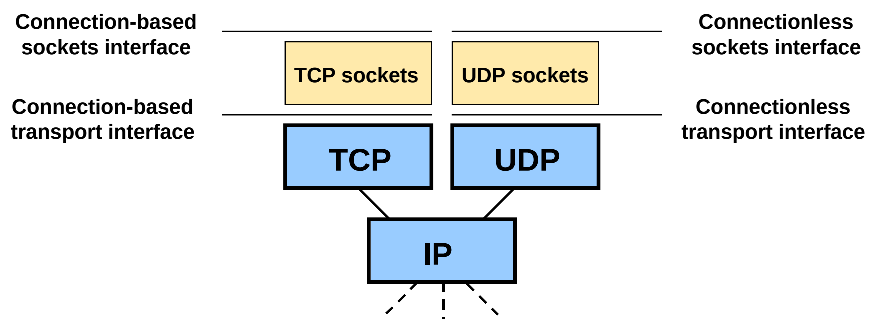

Connection-based sockets interface
Connectionless sockets interface
TCP sockets
UDP sockets
Connection-based transport interface
Connectionless transport interface
TCP
UDP
IP

Figure 4.4. The transport interface

A socket is an end point of a two-way communication link set up between two processes, which may run on the same machine or on different machines. A socket is associated with a port number. There are two kinds of sockets: stream sockets (or TCP sockets), which allow connection-oriented communication with character streams using TCP, and datagram sockets (or UDP sockets), which allow connectionless communication with messages using UDP. Libraries and APIs for using sockets are available in common programming languages, including C and Java.

Details on socket programming may be found in [Stevens et al. 2004]. The architecture of the Internet is described in [Peterson and Davie 2003], [Kurose and Ross 2004].

4.4 Middleware and Application-level Protocols

Middleware, by its nature, consists of protocol stacks running at the application level. In most cases, the base level of middleware protocols is the transport level, and the protocols are implemented using the sockets interface. Examples of middleware protocols examined further in this book are RTP (see 4.6.3) and GIOP/IIOP (see 5.3.1).

One particular way of implementing a middleware layer is by defining a new network on top of an existing one, with additional properties. This is an instance of the virtualization approach described in 2.2.1.

Such a virtual (or logical) network is called an *overlay network*. The term "overlay" refers to the fact that the functions of the new network are implemented in a set of nodes on the existing network, without interfering with the internal structure of that network. Note that this principle is used at several levels in the Internet itself: for example, the TCP transport layer is an overlay network built over the network (IP) layer, to provide additional properties such as connection-oriented communication and flow control.

In this section, we briefly discuss the main uses and properties of overlay networks. We then illustrate middleware or application-level protocols with the example of gossip protocols, a class of algorithms that finds a number of applications in large scale networks.

4.4.1 Overlay Networks

Overlay networks have been used to implement a variety of functions, some of which are listed below.

- Resilience. Resilience is the ability to continue operation in the presence of failures. A resilient overlay network [Andersen et al. 2001] allows an application deployed on a set of nodes to work in spite of failures of the underlying network. This is achieved by detecting path failures and finding alternate routes.

- Experiments with new protocols and network architectures. This may be done at various levels. For example, the MBone [Svetz et al. 1996] (now no longer used) has been developed as an experimental overlay network to implement the IP Multicast protocol before multicast-enabled routers were available. At a higher level, virtual testbeds [Peterson et al. 2004] allow a set of overlay nodes to be multiplexed between several concurrently running experiments.

- Application level multicast. Multicast may be implemented as an overlay network at the host level: the member nodes directly cooperate to minimize message duplication. This approach contrasts with the above-mentioned IP Multicast, which involves routers instead of hosts.

- Content delivery. The function of a content distribution network is to allow efficient delivery of data stored on back-end servers, by caching the most frequently accessed data on a set of widely distributed "surrogate" servers. A client wanting to access a stored information is directed to the surrogate that will most likely minimize its access time. This function of routing is done by an overlay network, whose nodes maintain information on the current load and redirect the requests accordingly.

- Distributed structured storage for Peer to Peer networks. A Peer to Peer (P2P) network is one in which all nodes are both clients and servers, and cooperate in an autonomous, decentralized manner. Locating data in a P2P network may be achieved by using a structured overlay network, i.e., one in which the location of a piece of data is determined by a key derived from its contents. This amounts to building a distributed hash table (DHT). Various organizations have been proposed for DHTs [Stoica et al. 2003, Ratnasamy et al. 2001, Rowstron and Druschel 2001]. A reference framework for structured overlay networks is proposed in [Aberer et al. 2005].

Most existing overlay networks either implement a transport layer (i.e., they are built on top of a network layer, usually IP), or an application-level layer (i.e., they are built on top of a transport layer).

The advantages and drawbacks of overlay networks may be appreciated by comparing two alternative ways of providing a given function to applications in a network: through programs running on the nodes of an overlay network or through code in the routers. This discussion is inspired from [Jannotti et al. 2000], to which we refer for further details.

An overlay network has the following benefits:

- It is incrementally deployable: nodes may be added to the overlay network without changing the existing infrastructure.
- It is adaptable, since its routing characteristics may be constantly optimized according to the needs of the application.
- It is robust, since redundancy may be added (e.g., by providing at least two independent paths between any two nodes), and since it has permanent control over its state, so it may react immediately.
- It uses standard protocols, in contrast to solutions based on reprogramming the routers

On the other hand, the designer of an overlay network is faced with the following problems.

- Management complexity. This is a general problem of distributed system administration (see Chapter 10); the manager of the network has to deal with a set of physically remote nodes.
- Security barriers. Many nodes of an actual network are behind firewalls or Network Address Translators (NAT). This complicates the task of deploying an overlay network.
- Efficiency. An overlay network has a performance penalty with respect to an implementation based on code in the routers.
- Information loss. This is the counterpart of virtualization. Since an overlay network runs on top of IP or of a transport protocol, the actual topology of the underlying network is not easily visible.

It is difficult to draw a uniform conclusion, since the trade-off between benefits and defects must be appreciated for each single application. However, the ease of deployment and adaptability of overlay networks makes them a major asset for experiments. In many cases (e.g., if the size of the network is limited), the efficiency is acceptable, so that the overlay network may be used as a long standing solution as well.

4.4.2 Gossip Protocols

Gossip (also called *epidemic*) protocols are a class of application-level communication protocols based on probabilistic methods. Their main use is information dissemination in a large scale, dynamically changing network, which makes them well suited for peer to peer environments.

Gossip protocols are based on the paradigm of *random propagation* in a large population, examples of which are the epidemic spread of a contagious disease or the propagation of a rumor via gossip talk.

Consider the problem of multicasting a message to a given population of recipients (e.g., nodes on a network). A gossip protocol works as follows. When a member of the population receives the message, it forwards it to a randomly selected set of other members. The initial sender starts the process in the same way. The process of random selection is central in a gossip algorithm. In the standard version of the protocol, the subset of recipients at each stage has a fixed size, called the *fanout* (noted f), and the members of the subset are chosen with uniform probability among the whole population.

The above algorithm is a form of flooding, and a given member will likely receive the same message several times. Since resources are finite, some bounds need to be set. Therefore, additional parameters of the protocol are the buffer capacity (noted b) of a member (i.e., the maximum number of messages that it may hold), and the number of times (noted t) that a member propagates a given message (the cases $t = 1$ and t unbounded are known as "infect and die" and "infect forever", respectively, with the epidemic disease analogy).

A central question is the following: under which conditions will a gossip-based broadcast protocol approach the coverage of a deterministic one? Define an *atomic* broadcast as one that reaches every member of the population. Let n be the size of the population. Assuming that b, the buffer size, is unbounded, the proportion p of broadcasts that are atomic (or the probability of a given broadcast to be atomic), is determined by the value of t and f. It can be shown that, to maintain the value of p when n increases, either f or t must increase as a function of n. As an example, suppose $t = 1$. Then the algorithm exhibits a bimodal behavior when f varies for a fixed n: when f is less than $log(n)$, the proportion of atomic broadcasts is close to 0; when $f > log(n)$, the proportion increases with f, and reaches 1 for a high enough value of (typically $2log(n)$).

Thus, gossip-based broadcast appears to be a powerful tool, with the advantages of simplicity and independence on the topology of the network. The load for each participant is moderate. Because of their highly redundant nature, gossip protocols tolerate node failures. The random selection of receivers at each stage also makes them resistant to transient network failures.

The actual situation is more complex than the above simple model. In particular:

- The population may change: members may enter or leave; they may experience permanent or temporary failure.
- The uniform random selection of recipients at each stage of the propagation implicitly assumes that each member knows the whole population. This is usually not the case, specially in large, dynamic networks.
- The protocol relies on a correct operation of each member, and is therefore vulnerable to attacks or misbehavior.

Various solutions exist to the random selection problem. Interestingly, many solutions rely on gossip itself, as each node may propagate lists of nodes that it knows. In many cases, the protocol relies on selection in a local cache, and it has been shown that this process is roughly equivalent to that of a global selection. The robustness of gossip protocols to failures and attacks, and ways to improve it, is surveyed in [Alvisi et al. 2007].

Applications of gossip protocols may be roughly divided in three classes [Birman 2007]:

- Information spreading. Examples are discussed in 6.3 (event dissemination) and 10.3.3 (multicast protocols).
- Anti-entropy, for reconciling differences in replicated data. This was one of the earliest applications of gossip protocols [Demers et al. 1987].
- Protocols that compute aggregates, i.e., system-wide values (e.g., a mean) derived from values collected from each member. See an example (observation) in 11.4.4 .

Gossip protocols are the subject of active research. A collection of recent articles [Kermarrec and van Steen 2007a] gives a picture of some recent results and open questions. In particular, [Kermarrec and van Steen 2007b] propose a general framework to describe and classify gossip protocols and [Birman 2007] discusses the advantages, limitations, and promises of gossip protocols.

4.5 Building a Communication System

In this section, we examine some architectural aspects of the process of building a communication system. The approach that we describe is based on abstraction and modular composition. Making explicit the structure of the communication system as an assembly of parts has a number of advantages: conceptual elegance; flexibility, including dynamic adaptation to changing operating conditions; provision of a testbed for alternative designs. A potential drawback is the performance penalty imposed by crossing the boundaries between parts; however, optimization techniques allow this overhead to be reduced to an acceptable level in most cases. A more detailed discussion of system composition may be found in Chapter 7.

In 4.5.1, we briefly present the x-kernel, a framework that pioneered the area of a modular approach to the construction of communication systems. In 4.5.2, we describe more recent work, which exploits further progress in component-based models and building tools.

4.5.1 The x-kernel, an Object-based Framework for Protocols

The x-kernel project [Hutchinson and Peterson 1991] has defined a systematic way of building a communication system, using a generic framework.

The framework is object-oriented, i.e., the abstractions that it supports are represented by objects (the notion of an object is defined in 2.2.2). The framework is based on three main notions: *protocols*, *sessions*, and *messages*. As explained in 4.3.1, a protocol defines an abstract channel through which messages may be sent. A session is a concrete representation of such a channel: it exports the API that allows a user of the protocol to send and receive messages. A given protocol may support a number of different sessions, corresponding to different groups of participants.

The general organization of the framework is as follows. A protocol graph is initially created, according to the specified protocol hierarchy; the arcs of the graph provide an explicit representation of the links between the protocols. Each protocol then acts as a session factory: it exports operations that allow sessions to be created and deleted according to the communication needs of the participants. An example of a protocol graph is shown on Figure 4.5

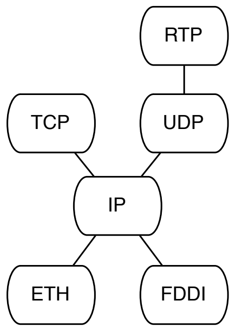


Figure 4.5. A protocol graph

A session represents a communication channel operating under a certain protocol, and used by an application to exchange messages on a network. Like protocols, sessions are organized in levels, and a number of sessions may be simultaneously operating under a given protocol. At each level, a session provides an API through which messages may be sent or received. A message sent by an application using a session is propagated downwards in the protocol hierarchy, using an `xPush` operation at each session level. A message that arrives at the receiving end of a session on a site is propagated upwards in the protocol hierarchy, towards the application, using an `xPop` operation at each session level. The `xPush` and `xPop` operations actually implement the communication algorithms at each level of the protocol stack.

As explained in 4.3.1, send and receive are not symmetrical. A session is aware of the sessions below it (because session creation is a top-down process), but not of the sessions above it. Therefore a message propagating upwards needs to be demultiplexed, using the `xDemux` operation of the protocol at the next level up, to be propagated to the right session (or to the application, at the top level). This is done using a key present in the message. This process is summarized on Figure 4.6, which shows a message being received in Session

1 and a message being sent in Session 2.

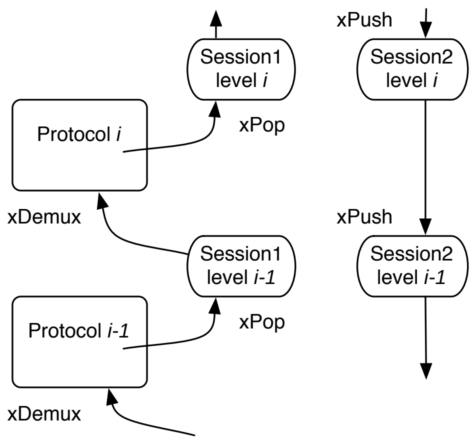


Figure 4.6. Message flow in the x-kernel

There are two approaches to organizing the processes in a communication system.

- Associating a process with each protocol: the *send* or *receive* operations are executed by protocols, and the messages are passive data.
- Associating a process with each message: the process represents the flow of the message across the protocol layers, and executes the *send* or *receive* operations as the message moves down or up the hierarchy.

The organization of the x-kernel follows the latter model. This choice is motivated by efficiency reasons: in the process-per-protocol model, crossing a layer in the protocol hierarchy incurs the cost of a context switch, while in the process-per-message model the cost is that, much lower, of a procedure call. The cost of process (or thread) creation may be amortized by using pools.

A related approach is taken by the Scout system [Mosberger and Peterson 1996], in which execution is organized in "paths" (sequences of processing units, or stages, at the different protocol levels). Each path is run by a single thread.

The x-kernel has provided inspiration to a number of communication systems, among which Jonathan (described in more detail in 4.6), Appia [Miranda et al. 2001], Horus [van Renesse et al. 1996]. It is also the starting point of a more recent generation of frameworks, described in the next section.

4.5.2 Component-based Frameworks for Protocols

After the early contribution of the x-kernel and related systems, the evolution of the area of decomposition of network protocols has been influenced by two major trends:

- The transition from objects to components as a decomposition unit. In contrast with objects, components put stress on architectural aspects (e.g., explicit specification of required resources, explicit representation of structural relationships), and preservation of the decomposition units at run time (see 7.1 for a detailed discussion).

- The requirements for customization, flexibility, and dynamic adaptation of communication protocols, imposed by the increased visibility of these protocols at the application level (e.g., in the construction of overlay networks). There is a need for a finer grain decomposition: while the composition units in the x-kernel are whole protocols, recent frameworks allow a protocol itself to be decomposed in elementary components.

  Several benefits result from this approach (see [Condie et al. 2005] for a more detailed discussion):

  – Application-level protocols may have requirements that are not well met by the existing standard transport protocols such as TCP. In that case, an application may construct its own transport protocol by assembling predefined components, leaving out unneeded functions. A variety of protocols may thus be assembled, and components may be shared between protocols for economy.

  – By preserving the component structure at run time, a protocol may be dynamically reconfigured to react to changing conditions such as overload or link failure.

  – By isolating the various functions of a protocol (such as congestion control, destination choice, recovery from failure) in separate components, various combinations of these functions may be implemented according to the needs, possibly within a single application.

We illustrate the component-based organization of communication protocols with three examples of experimental systems.

- Click, a framework for the construction of routers.
- Dream, a framework for the construction of asynchronous middleware.
- SensorNet, a framework for the construction of sensor networks.

These systems share the objectives of economy, flexibility, and explicit architectural representation, applied to various usage contexts.

The Click Modular Router

Click [Kohler et al. 2000] is a framework dedicated to the construction of configurable IP routers. The main function of a router is to implement the IP protocol by forwarding incoming packets to their appropriate next destination, which may be a host or another router. In addition, a router often performs other functions, such as packet tunneling and filtering, or implementing a firewall.

The objective of Click is to allow the program of a router to be easily configured and adapted. To that end, a router is built as an assembly of packet processing modules called *elements*. This assembly takes the form of a directed graph, whose edges represent the *connections* between the elements. This graph is an explicit representation of the architecture of the router.

Each element provides interface units called input and output *ports*, and a connection links an input port to an output port. Packets are sent on the connection, from the output port (source) to the input port (destination). There are two types of connections, "push" and "pull", according to whether the transfer of a packet is initiated by the source element or the destination element, respectively. Likewise, the type of a port may be defined (at router initialization time) as push, pull, or agnostic (i.e., neutral). An agnostic port behaves as push or pull if connected to a port of the push or pull type, respectively. The type of a connection is that of its ports. In addition to ports, an element may have a procedural interface accessible to other elements.

Validity rules define correct router configurations: a push output or a pull input port may not be the endpoint of more than one connection; a connection may not be set up between a push and a pull port; if an element acts as a filter between two agnostic ports, these ports must be used in the same way (push or pull). When the router is initialized, the connections are checked for validity, and the ports that were initially agnostic are set to the proper type according to the connections.

Contrary to the usual implementations of "ports", Click ports do not have built-in queues. A queue must be explicitly implemented as a *Queue* element, with a push input port and a pull output port.

A Click configuration is described by a simple declarative language, which allows elements and connections to be defined. Classes are defined for the usual elements, and compound classes may be defined by composing existing classes. A configuration description is used by the Click kernel to create and to initialize a configuration.

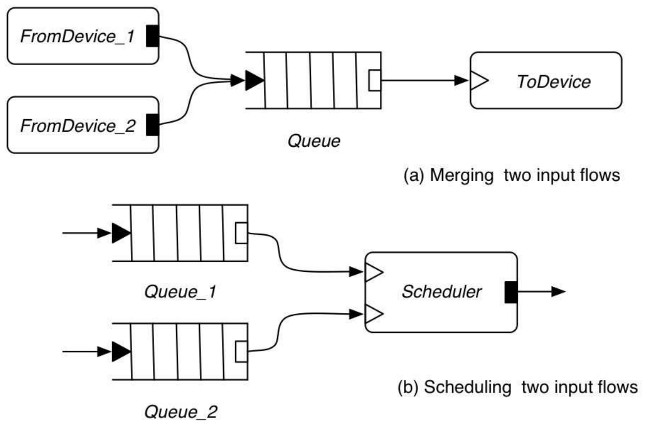


Figure 4.7. Elementary constructions in Click

Figure 4.7 illustrates two elementary configurations in Click. Output ports are represented by rectangles, input ports by triangles; push ports are black, pull ports are white. In configuration (a), packets coming from two devices are merged in a single flow, which is directed to an input device through a queue. In configuration (b), two input flows are input to a scheduler, which successively inputs packets from either flow according to its policy, and sends them on its output port. Note that in (a) the order of the packets entering the input device is determined by the order in which the packets are pushed into the queue, whereas in (b) the order of packets in the output flow is determined by the scheduler.

Elements are the units of CPU scheduling: an element that needs CPU time (e.g., because its `push` or `pull` methods are called) enters a task queue. A single thread is used to run this queue.

The Click framework has been used to construct actual IP routers. A typical router is made of a few tens of elements, and is highly configurable. Adding extensions and redefining scheduling and routing policies involve adding or reordering a few elements. Experience shows that the overhead due to the modular structure of the router is acceptable.

Dream: A Framework for Configurable Middleware

Dream [Leclercq et al. 2005] is a component-based framework dedicated to the construction of communication middleware. It provides a component library and a set of tools to build, configure and deploy middleware implementing various communication paradigms: group communication, message passing, event-reaction, publish-subscribe, etc. Dream builds upon the Fractal component framework (see 7.6).

Dream inherits the advantages of Fractal, among which hierarchical composition with component sharing, explicit definition of both provided and required interfaces, provision of an extensible management interface, which allows fine-grained control on binding and life-cycle at the component level.

Figure 4.8 shows the architecture of a simple Dream system (which is also a component, by virtue of hierarchical composition).

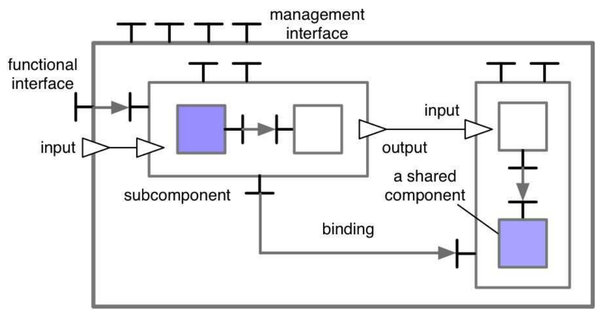


Figure 4.8. System composition in Dream (from [Leclercq et al. 2005])

This figure illustrates the main aspects of composition in Dream: composite components (i.e., components containing other components), component sharing, connections (or bindings, in the Fractal terminology) between components. A binding may connect a required interface to a provided one (for synchronous method call), or an output port to an input port (for asynchronous message passing). Conformity rules (see 7.6) specify compatibility between the interfaces and therefore define legal bindings.

In addition, Dream provides features that facilitate the construction of communication systems:

- A component library, which contains components ensuring the functions most commonly fond in an asynchronous communication system, and components for managing resources (e.g., memory chunks for message management and schedulers for activity management).

- A specific type system, together with tools for type checking. The type system allows the designer of a communication system to attach semantics to messages and to components. The system guarantees that a "well formed" configuration (whose components conform to their types) will not be subject to run time failures.
- Tools for deployment, based on a structural description of the architecture of a system using an Architecture Description Language (ADL).

While the overall composition structure of Dream is close to that of Click, there are a few important differences.

- The use of of a sound underlying component model, Fractal, provides an explicit, well-structured, representation of the architecture, together with flexibility, including run time reconfiguration.
- The Dream type system allows a rigorous specification of the semantics of a system, and enables early detection of incorrect constructions.
- Dream provides flexible resource management facilities, which allow control over quality of service.

Dream has been used for full size experiments, among which a reimplementation of Joram (Java Open Reliable Asynchronous Messaging), a JMS compliant, industrial strength, open source middleware (6.8). This allows the Dream-based Joram to be easily reconfigured, possibly at run time. This benefit comes with a negligible penalty (about 2 percent in execution time and a fraction of a percent in memory footprint).

A Modular Network Layer for SensorNets

Wireless sensor networks, which are being developed for various applications, pose specific communication problems: achieving efficient and reliable communication with the constraint of scarce resources and noisy, time varying links. One proposal [Culler et al. 2005, Polastre et al. 2005] in response to these problems has been a unifying "narrow waist" architecture inspired by that of the Internet, with a single protocol, SP (Sensornet Protocol) providing an abstract layer for building higher level protocols while allowing multiple low-level protocols to coexist. SP differs from IP in two ways: it sits at an intermediate level between the link and network layers (i.e., it allows multiple network-level layers); and it does not impose a single format. Rather, it provides a set of services that abstract properties of the link layer (such as MAC format) and allow resource sharing between multiple link and network layer protocols operating simultaneously.

Building on SP, a modular network layer has been proposed [Ee et al. 2006], with the objective of minimizing redundancy by sharing fine-grain elements between protocols.

To that end, a decomposition of the network layer has been proposed, which identifies a "control plane" and a "data plane" (Figure 4.9). The data plane defines a data path between a dispatcher to an output queue through a forwarding engine. The control plane controls the forwarding engine and the output queue, through two modules: the routing engine and the routing topology. While the dispatcher and the output queue are unique, multiple instances of the other components may coexist.

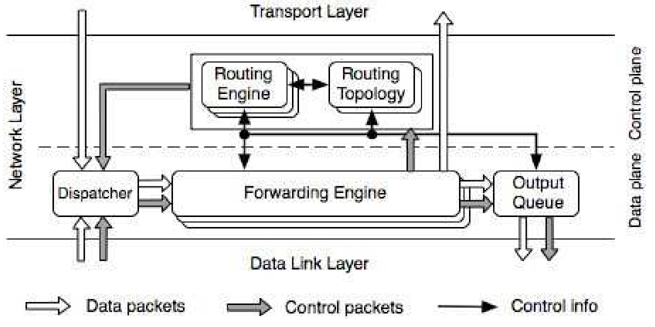


Figure 4.9. Architecture of the Sensornet network layer (from [Ee et al. 2006])

The routing engine determines whether a packet should be forwarded, and, if that is the case, its next hop. The routing topology module exchanges information with its peers on other nodes to determine and to maintain the network topology. The forwarding engine queries the routing engine to obtain the next hop to which a packet must be sent.

In the experiments reported in [Ee et al. 2006], various existing protocols have been decomposed using specific instances of the basic elements described above. Experience shows that the objective of code reuse is actually achieved, with significant gains in memory occupation. As expected, there is a performance overhead with respect to a monolithic architecture. This penalty is considered acceptable in the context of sensor nets.

Conclusion on Component-based Communication Frameworks

The three case studies briefly discussed above illustrate a common trend in component-based communication systems, characterized by a search for common abstractions, an emphasis on architectural description with explicit structure and composition rules, and fine grain decomposition and sharing. The benefits are flexibility, adaptability, and economy (both conceptual and in terms of resources). There is a cost in terms of performance, which may be mitigated in most cases. A further step would be to derive the structure of the communication system from higher-level requirements. An example of research in this direction is described in [Loo et al. 2005]

4.6 Case Study: the Jonathan Communication Framework

Jonathan [Dumant et al. 1998] is a set of frameworks for building Object Request Brokers (ORBs). An overview of Jonathan, and a description of its binding framework, based on the `export-bind` pattern, is presented in 3.4. Here we describe the communication framework.

4.6.1 Principles of the Communication Framework

The Jonathan communication framework follows the general pattern introduced by the x-kernel and described in 4.5, i.e., a protocol graph whose base layer is at the transport level, used through Java sockets. We are not concerned about the lower levels.

Sessions

The main communication abstraction provided by Jonathan is a *session* (4.5), which represents a communication channel. A session supplies an interface for sending and receiving messages; actually two different interfaces (`Session_Low` and `Session_High`) are respectively provided for incoming and outgoing messages. In Jonathan, a protocol is essentially a session manager: it creates sessions, acts as a naming and binding context for these sessions, and provides them with communication resources. Like protocols, sessions are organized in a hierarchy. At the lowest level, a session relies on a basic communication mechanism called a *connection*, which provides an interface to send and to receive elementary messages (sequences of bytes). For instance, in the TCP-IP protocol suite, a connection provides the `IpConnection` interface and encapsulates a socket.

The main communication primitives are message `send` and `receive`. As explained in 4.3.1, they operate in different ways, due to the asynchronous nature of receiving. A read operation (implemented by a `receive()` method on a connection) blocks the executing thread until data is available on the input channel associated with the connection. When data becomes available (a message has arrived), the thread is unblocked, causing the message to be passed up the protocol stack by calling the "lower" interfaces of the sessions, in ascending order. On the other hand, an application process sends an outgoing message by calling the "higher" interface provided by a session. The message is then sent down the protocol stack by calling "higher" interfaces in descending order, down to the call of an `emit` method on the connection. Figure 4.10 gives an overview of this mechanism, which is described in further detail in Section 4.6.2.

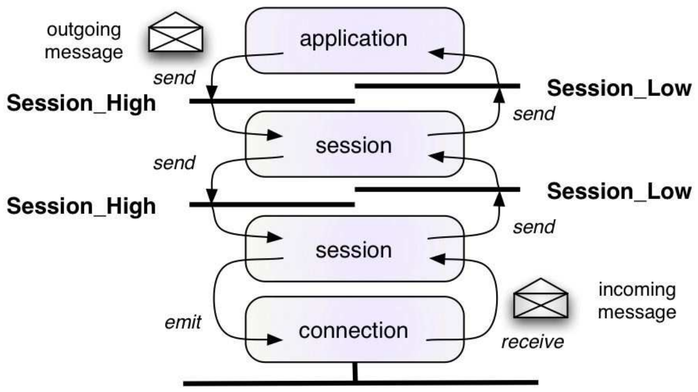


Figure 4.10. Sending and receiving messages

Sessions are set up according to the Jonathan binding framework. On the server side, a *protocol graph* is first constructed by assembling elementary protocols. The protocol graph is a *naming context*, which provides the `export` method. The exported interface (`srv_itf`) is the "lower" interface of a session (of type `Session_Low`), which provides the functionality of the server. The `export` method returns a session identifier (a name for the exported interface), which contains all the information needed to set up a communication with the server (e.g.,for TCP/IP, the IP address of the server and a port number). This information may be transmitted over the network and decoded by a client.

In order to be able to access the interface exported by a server, a client must call the `bind` method provided by a session identifier that designates the server, passing the client application's "lower" interface (`clt_itf`) as a parameter. The session identifier may be obtained from the network (e.g., through a name service), or it may be constructed locally using the server address and port number if these are known. The `bind` method returns an interface `session` of type `Session_High`, which may be used by the client to call the server. Messages from the server are directed to the client application, through the interface `clt_itf` provided as a parameter of the call to `bind`.

A general picture of the `export-bind` mechanism is outlined on Figure 4.11. Many details are omitted; these are provided in Section 4.6.2.

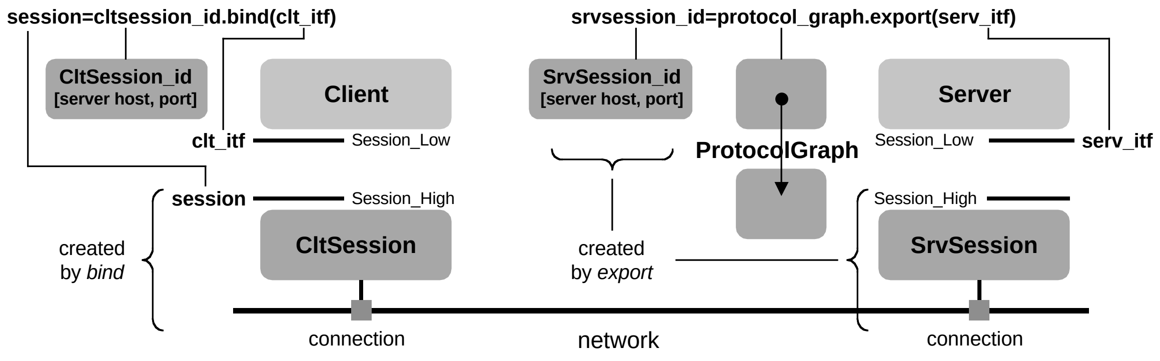


Figure 4.11. The `export-bind` pattern for session setup

Actual communication relies on two services: chunks and (un)marshallers, that are provided, respectively, by the Jonathan resource library and the Jeremie presentation library. We describe these services briefly.

- **Chunks**. A chunk represents a part of an array of bytes. Chunks are linked to form messages that may be sent from an address space to another. Using chunks avoids unnecessarily copying arrays of bytes, and helps recovering these arrays without resorting to garbage collection (thanks to chunk factories).

- **Marshallers and unmarshallers**. Marshallers are used to convert typed data into a standard serialized form suitable for transmission on a network. Unmarshallers perform the reverse function. Thus a `Marshaller` is used as an abstract (i.e., network independent) output device, whose interface provides methods to write data of various types; likewise, an `Unmarshaller` acts as an abstract input device, whose interface provides methods to read data of various types.

Typically, marshallers and unmarshallers are used as follows (this is a simplified example).

> Sending a message composed of an integer `i` followed by a 8-byte string `str` followed by an object `obj`.

```
    Session_High session ...
    StdMarshallerFactory marshaller_factory ...
    ...
    Marshaller m = marshaller\_factory.newMarshaller();
    marshaller.writeInt(i);
    marshaller.writeString8(str);
    marshaller.writeValue(obj);
    session.send(marshaller);
    ...
```

> Receiving the message sent by the above program sequence; the following sequence is supposed to be part of a method having `Unmarshaller unmarshaller` as a parameter.

```
    i=unmarshaller.readInt();}
    str=unmarshaller.writeString8();}
    obj=unmarshaller.readValue();}
    unmarshaller.close();}
    ...
```

Marshallers and unmarshallers are created by marshaller factories. A marshaller factory is usually provided in the bootstrap configuration of Jonathan (see the configuration framework tutorial).

The Communication Infrastructure: Java Sockets

The first example (using Java sockets) does not involve Jonathan at all. It illustrates, at a fairly low level, the export-bind pattern of interaction that is further expanded in the following use cases. Consider a server that provides a service to a single client at a time (multiple clients are considered later on). The server selects a port (port 3456 in this example) and creates a server socket associated with that port. It then waits for client connections by calling `accept()` on the socket. When a client connects to port 3456 (this is done in the `Socket` constructor), `accept()` returns a new socket dedicated to exchanges with the client. The original socket remains available for new connections (if we do not create a new thread per client, only one client connection may be opened at a time).

```
Server

// create a new server socket associated with a specified port
   server_socket = new ServerSocket(3456);
```

```
// wait for client connections:~a ‘‘pseudo-export’’ operation
   Socket socket = server_socket.accept();
// socket is now available for communication with client

Client

// connecting to server: a ‘‘pseudo-bind’’ operation
   Socket socket = new Socket(hostname, 3456);
// socket is now available for communication with server
```

In effect, the `accept()` call in the server program is equivalent to our *export* primitive, while the `connect()` implicitly called in the `Socket` constructor in the client program is equivalent to our *bind* primitive.

Note that the binding process always relies on an information shared by the client and the server (here, the host name and the port number). In the present case, this shared information is hardwired in the code. More elaborate methods are introduced in further examples.

The complete code of an example using this pattern (a simple echo server) may be found in the Sun Java tutorial:
`http://java.sun.com/docs/books/tutorial/networking/sockets/`

4.6.2 The TCP-IP Protocol

In this section, we present the implementation of the TCP-IP communication protocol in Jonathan. This is a typical example of the way communication frameworks are defined and used in Jonathan.

Overview of the TCP-IP Framework

The `libs.protocols.tcpip` package implements the session level, together with the "chunk provider" which allows a session to get input data from a connection. The `libs.resources.tcpip` package implements the connection level. The session and connection levels are described in the following sections.

The Session Level

Since sessions play a central part in the communication framework, it is important to understand the interplay between sessions at different levels. We illustrate this by the example of TCP-IP (Figure 4.12). The general pattern outlined on this figure applies both on the client and on the server side. The main difference is that the server-side sessions are typically created by `export`, while the client-side sessions are created by `bind`.

At the lower level, we have a TcpIp session, which essentially encapsulates a connection to the network. It has two functions:

- to receive messages from the network and to pass them up to the upper level ("application") session;

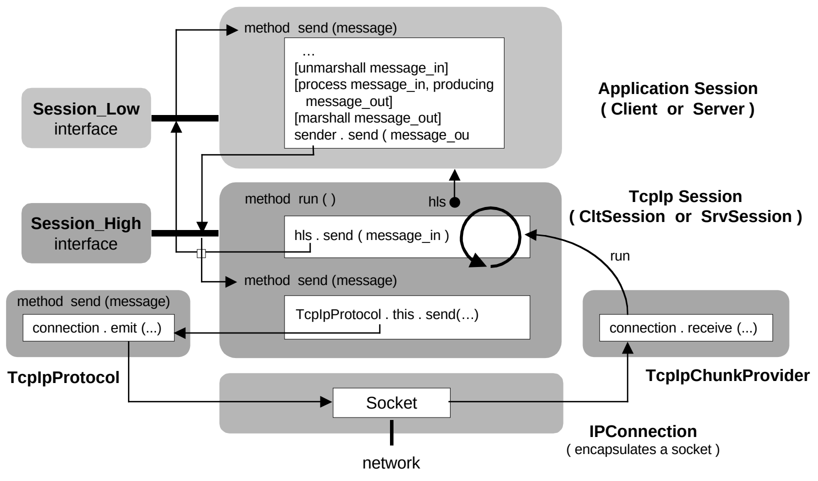


Figure 4.12. Sessions in the TCP-IP Jonathan framework

- to implement an interface (called `Session_High`) allowing the application session to send messages through the network to its "sibling" application session (i.e., client to server and server to client). In this sense, the TcpIp session acts as a surrogate to a (remote) application session.

The TcpIp session has two slightly different forms (`TcpIpProtocol.CltSession` and `TcpIpProtocol.SrvSession`) on the client and server side.

At the upper level, we have an application session, which provides the client or server functionality. The application session

- sends messages on the network by calling the `send` method provided by the TcpIp session in its `Session_High` interface.
- receives messages from a lower level session through the `Session_Low` interface that it implements. However, there is no explicit `receive` operation; instead, the TcpIp session delivers an incoming message to the application session by calling the `send` method of that session's `Session_Low` interface.

It is important to emphasize the difference between the `Session_High` and `Session_Low` interfaces (especially since both interfaces include a method called `send`, which may seem confusing at first sight).

- `Session_High` is used by the application session to send messages "downwards". If `lower` is a variable that designates a TcpIp session in the application session, `lower.send(message)` sends a message down to the network (eventually to the remote application session for which the TcpIp session is a surrogate).
- `Session_Low` is used by the TcpIp session to send messages "upwards". If `hls` (standing for "higher level session") is the variable that designates an application

session in the TcpIp session, `hls.send(message)` sends a (presumably incoming) message up to the application session.

The classes `ServerSession` and `ClientSession` that implement the server and client application sessions of the Echo application have the following general outline.

```
class ServerSession implements Session_Low{
   ServerSession();
   static MarshallerFactory marshaller_factory;
   private int counter;             //internal state of session

   // the server method for accepting requests:
   //    - unmarshaller: the request message
   //    - sender: the local interface to the client

   public void send(UnMarshaller unmarshaller, Session_High sender){
      String theOutput = null;
      String theInput = unmarshaller.readString8();
      theOutput = counter + ":" + theInput;
      unmarshaller.close();
      Marshaller marshaller = marshaller_factory.newMarshaller();
      sender.prepare(marshaller);
      marshaller.writeString8(theOutput);
      sender.send(marshaller);
   }
}


class ClientSession implements Session_Low {
   static MarshallerFactory marshaller_factory;
   BufferedReader reader;           // for terminal input by client
   ClientSession(BufferedReader reader);
   this.reader = reader;}

   // the client method for accepting messages from server
   //    - unmarshaller: the message
   //    - session: the local interface to the server

   public void send(UnMarshaller unmarshaller, Session_High session){
      String fromServer,input;
      System.out.print("Client: "); // prompting client
      System.out.flush();
      input = reader.readLine();
      Marshaller marshaller = marshaller_factory.newMarshaller();
      session.prepare(marshaller);
      marshaller.writeString8(input);
      session.send(marshaller);
      fromServer = unmarshaller.readString8();
      unmarshaller.close();
   }
}
```

The actual programs include, in addition, provision for exception handling and for nice termination of client sessions. They also contain provision for multiple clients, to be explained later on (cf. Section 4.6.2).

Setting up sessions

The mechanism for session setup uses the binding framework based on the `export-bind` pattern (3.3.2).

Both the server and the client start by an initial configuration phase and create an instance of `TcpIpProtocol`. Then each side instantiates a session as follows.

- On the server side, an instance of `ServerSession` (the application session) is created. Then, a protocol graph is created with a single node (the instance of `TcpIpProtocol`). Finally, this graph `export`s the newly created application session: it creates an instance of `SrvSession`, with the `ServerSession` instance as its higher level session, and returns a session identifier that designates the exported session. Here is the code sequence that does this:

```
Server:
// configuring the system: creating factories
// (described in the configuration tutorial)
// creating a protocol instance (a naming context for sessions)
   TcpIpProtocol protocol =
      new TcpIpProtocol(<parameters, to be described later>);

// creating and exporting a new session
   SessionIdentifier session_id =
      protocol.newProtocolGraph(port).export (new ServerSession());
// if no port specified, selects an unused port
```

- On the client side, a new session identifier (`participant`) is created to designate the remote server (In this version, we still assume that the name of the server host and the server port are known by the client). An instance of `ClientSession` (the application session) is created. Finally, the `bind` method is called on the participant identifier: it creates a new instance of `CltSession`, with the `ClientSession` instance as its higher level session, and returns a session identifier that designates the exported session. Here is the code sequence that does this:

```
Client:
// configuring the system: creating factories
// (described in the configuration tutorial)
// creating reader, getting server hostname and port

// creating a protocol instance (a naming context for sessions)
   TcpIpProtocol protocol =
      new TcpIpProtocol(<parameters, to be described later>);

// preparing for connection to server
```

```
      IpSessionIdentifier participant =
         protocol.newSessionIdentifier(hostname,port) ;

   // creating client-side session and connecting to server
      Session_High session = participant.bind (new ClientSession(reader)) ;
   // session is now available for communication with server}
```

From this point on, the core of the program runs in the application programs, i.e., the `ClientSession` and `ServerSession` classes, as described above.

The Connection Level

The interfaces provided by the session level abstract away (in the `send` methods) the low-level message transmission mechanism. This mechanism is defined at the connection level and (in the current implementation) relies on two classes: `JConnectionMgr` defines generic mechanisms for using socket-based connections, and `IPv4ConnectionFactory` provides a specific implementation of these mechanisms. The main abstraction at this level is the connection (instance of `IpConnection`), which encapsulates a socket.

For completeness, we now give a summary explanation of the mechanisms for message input. Recall that `CltSession` and `SrvSession` are the client and server incarnations, respectively, of the generic TcpIp session described above (in the code, both classes derive from a common abstract class, `TcpIpProtocol.Session`). This class extends `Runnable`, i.e., its instances are executed as independent threads activated by a `run()` method, which is called when a message is received. This is done through the `TcpIpProtocol.TcpChunkProvider` class, which encapsulates a socket input stream (through an `IpConnection`), and delivers messages as "chunks" (a `Chunk` is the abstraction provided by Jonathan to efficiently use data of variable length). This class has two main methods, `prepare()` and `close()`, which are respectively called as a prelude and postlude of all input operations performed through an `Unmarshaller` on the input stream. A `TcpChunkProvider` contains a data cache,which is used as follows.

- `prepare()` delivers the contents of the cache (if not empty) and attempts to read further data into the cache from the underlying connection(the input stream);
- `close()` is used to close the chunk provider if it is no longer used; if the cache is not empty, the session thread is reactivated, so the session may read the remaining data.

Thus the chunk provider effectively acts as a data pump that injects incoming messages into the TcpIp session, which in turn sends them to the upper level application session.

Putting it all together

We now describe in detail the internal workings of the `export-bind` operations. An overview of these operations is given in Figure 4.13 which gives a more detailed picture of the process outlined on Figure 4.11.

Calling the `export` method on `ProtocolGraph` has the following effect (s1, c1, etc. refer to the tags that designate the server and client operations on Figure 4.13).

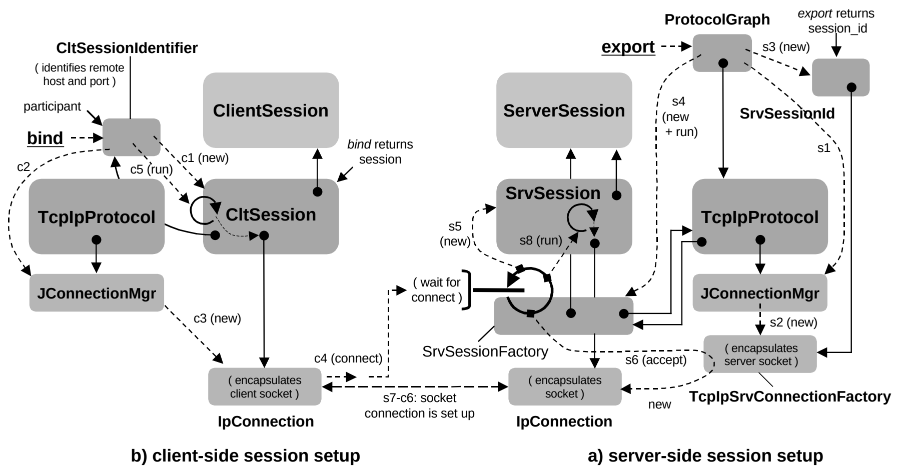


Figure 4.13. Creating client and server sessions

- The `newSrvConnectionFactory(port)` method is called on `JConnectionMgr` (s1). This creates a new instance of a `TcpIpSrvConnectionFactory` (s2), which encapsulates a server socket bound to the port provided as parameter (if 0, an available port is selected).
- A new instance (`session_id`) of `SrvSessionId` is created (s3); it contains the host name of the server and the port number of the server socket.
- A new instance of `SrvSessionFactory` is created (s4); it has references to `session_id`, to the exported `ServerSession` and to the `TcpIpProtocol`.
- A new thread is started to execute the `run()` method of the `SrvSessionFactory`. The first action of this method is to create a new instance of `SrvSession` (s5).
- The `newConnection` method is called on the `TcpIpSrvConnectionFactory`. This method actually calls an `accept()` on the underlying server socket (s6). This is a blocking call. The server now waits for a `connect()` operation from a client.
- When `connect()` is called from a client (see client description below, step c4), a new socket is created and connected to the client socket (s7-c6).
- A new thread is created to execute the `run()` method of `SrvSession` (s8). This in turns starts reading messages from the socket, as explained in the description of connections.

Calling the `bind` method on `CltSessionIdentifier` has the following effect.

- A new instance of `CltSession` is created (c1).
- The `newCltConnection` method is called on `JConnectionMgr` (c2). This creates a new socket (c3), encapsulated in a `Connection`, an implementation of the `IpConnection` interface.

- The socket tries to `connect()` to the remote server, whose hostname and port number are included in the `CltSessionIdentifier` (c4).
- Finally, a new thread is created to execute the `run()` method of `CltSession` (c5). This in turns starts reading messages from the socket, as explained in the description of connections.

Serving multiple clients

Two patterns may be used for serving multiple clients, according to whether the server maintains a common state shared by all clients or a distinct state for each client.

Multiple connections with shared state. The mechanism described above allows several clients to connect to a single server, through the connection factory mechanism. If a new client `binds` to the server, a new connection is created (using the socket `accept` mechanism), as well as a new `SrvSession` instance encapsulating this connection, together with a new thread. However, there is still a unique application session (`ServerSession`) , whose state is shared between all clients (Figure 4.14). This is illustrated in the example programs by adding state to the application session, in the form of an integer variable `counter` that is incremented after each client call. Multiple clients see a single instance on this variable.

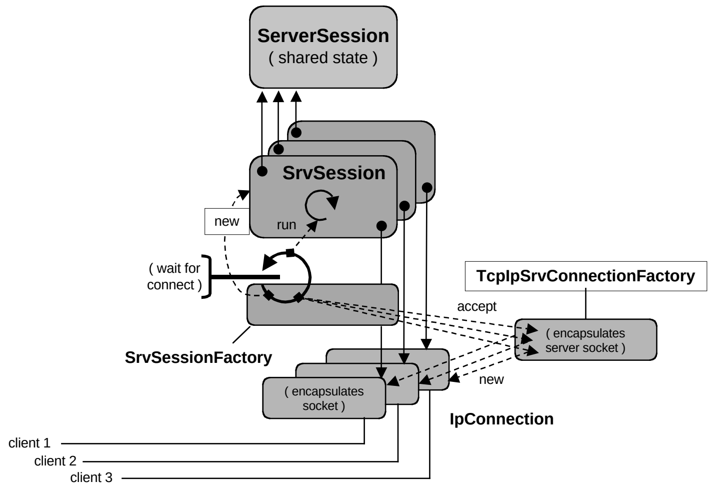


Figure 4.14. Multiple clients sharing a session state

If the application needs a per-client session state, then it is necessary to explicitly manage multiple sessions at the application level. This is done in the following example.

Multiple connections with private state In this example, each client is associated with a distinct application-level session that maintains the client's own version of the state (in this case, the `counter` variable). This is achieved, on the server side by an instance of

SrvProtocol, which has two functions: it acts as a factory that creates a new instance of the client session, OwnSession, when a new client connects; it acts as a demultiplexer that forwards the incoming messages to the appropriate OwnSession according to the identity of the sender. The factory is implemented as a hash table that contains an entry per session. The body of the application (in this case, the incrementation of the counter) is implemented by OwnSession.

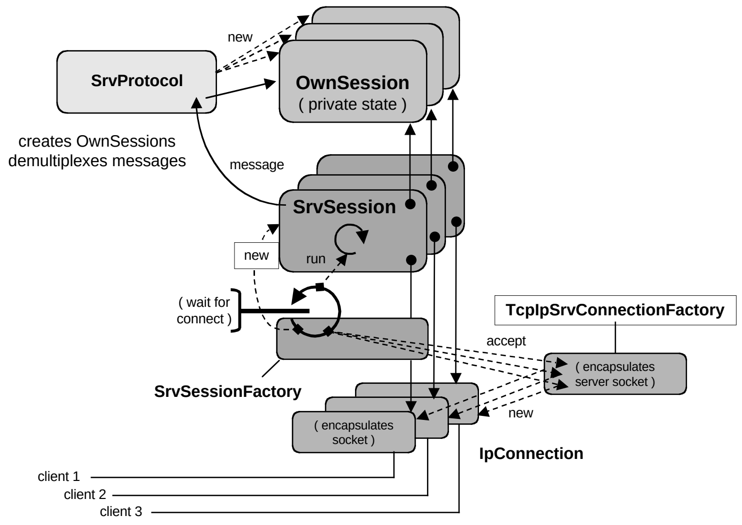


Figure 4.15. Multiple clients, with a server session per client

The client side is identical to that of the previous application.

4.6.3 Other Communication Protocols

In this section, we examine the implementation in Jonathan of two other protocols: the IP Multicast Protocol and the Real Time Protocol (RTP). These implementations conform to the export-bind pattern as described in Section 4.6.1, with local variations. Then we present the Event Channel use case, which combines both protocols. A brief presentation of the GIOP protocol may be found in 5.5.1.

The IP Multicast Protocol

As defined in IETF RFC 1112, "IP multicasting is the transmission of an IP datagram to a "host group", a set of zero or more hosts identified by a single IP destination address". IP addresses starting with 1110 (i.e., 224.0.0.1 to 239.255.255.255) are reserved for multicast. Groups are dynamic, i.e., a host may join or leave a group at any time; a host may be a member of several groups. A host need not be a member of a group to send a message to that group.

A particular class of Java sockets, java.net.MulticastSocket, is used in Jonathan as the base layer for implementing IP Multicast. A multicast socket s may subscribe to a host group g by executing s.joinGroup(g) and unsubscribe by executing s.leaveGroup(g).

In order to send a message `msg` to a group `g`, a datagram must first be created by `DatagramPacket d = new DatagramPacket(msg, msg.length, g, port)`, where `port` is the port to which socket `s` is bound. The datagram is then sent by executing `s.send(d)`.

The Jonathan `MulticastIpProtocol` class manages `MulticastIpSession` sessions. Each session is dedicated to a (`IP Multicast address, port`) network endpoint. A session may optionally be associated with an upper level session. In that case, it may send and receive messages, and a per session thread is used to wait for incoming messages. Otherwise, the session is only used to send messages.

In the IP Multicast protocol, there are no separate client and server roles; therefore there is no need to separate protocol graphs (which export servers) from session identifiers (which are used by clients to bind to servers). A single data structure, `MulticastIpSessionIdentifier`, is used for both functions (it implements `SessionIdentifier` and `ProtocolGraph` and thus provides both `export` and `bind`).

The sessions managed by this protocols are instances of class `MulticastIpSession`, which essentially provides two methods, `send` and `run`. In the current implementation, a single thread is created with each instance and waits on the multicast socket. The constructor is as follows:

```
MulticastIpSession(MulticastIpSessionIdentifier sid,
              Session_Low prev_protocol) throws IOException {
  socket=new MulticastSocket(sid.port());
  socket.joinGroup(sid.InetAddress());
  this.sid=sid;
  this.prev_protocol=prev_protocol;
  if(prev_protocol!=null) {
    reader =new Thread(this);
    cont = true;
    reader.start();
  }
}
```

The reader threads executes the `run()` method which is a loop with the following overall structure (exceptions not shown):

```
while (true) {
  socket.receive(packet); \\ a DatagramPacket
  extract message from packet
  send message to upper session, i.e., prev_protocol;
}
```

The `send(message)` method does essentially this:

```
encapsulate message into a DatagramPacket packet;
socket.send(packet);
```

A `MulticastIpSessionIdentifier` contains three fields: `address`, `port`, and `session`. As explained above, the `export` and `bind` methods are very similar. Both create a new socket with an associated session. They only differ by the type of the returned value (an identifier for `export`, a session for `bind`); in addition, export needs to supply an upper level interface.

```
public SessionIdentifier export(Session_Low hls) throws JonathanException {
   if (hls != null) {
     try {
           session = new MulticastIpSession(this,hls);
           return this;
     } catch (IOException e) {
           throw new ExportException(e);
     }
   } else {
     throw new ExportException("MulticastIpSessionIdentifier: no protocol low interface specif
   }
}

public Session_High bind(Session_Low hls)  throws JonathanException {
   try {
         return new MulticastIpSession(this,hls);
   } catch (IOException e) {
         throw new org.objectweb.jonathan.apis.binding.BindException(e);
   }
}
```

As shown above, this creates a multicast socket using the port associated with the identifier, and spawns the reader thread if an upper (receiving) interface (`hls`) is provided (if not, the socket is only used for sending and does not need a waiting thread).

The RTP Protocol

RTP (Real-time Transport Protocol) is the Internet standard protocol for the transport of real-time data, including audio and video. RTP works on top of an existing transport protocol (usually UDP) and is designed to be independent of that protocol. RTP is composed of a real-time transport protocol (RTP proper), and of an RTP control protocol, RTCP, which monitors the quality of service.

Jonathan provides a partial implementation, `RTPProtocol`, of the RTP protocol (not including RTCP). RTP packets have a 12 byte header (defined by the `RTPHeader` class), which includes such information as sequence number and timestamp.

`RTPProtocol` provides the usual `export` and `bind` operations to its users:

- `export(Session_Low hls)` is borne by a protocol graph built over the underlying protocol, and is called by a client providing `hls` interface for receiving messages. It returns a new `RTPSessionIdentifier`, which identifies an RTP session. It also creates a receiving front end for the `hls` interface, in the form of a decoder, an instance of the internal class `RTPDecoder`. The decoder extracts the header from an incoming message and forwards the message to the upper level interface (here `hls`).

- `bind(Session_Low client)` is borne by an `RTPSessionIdentifier`. Its effect is to bind `client` to a new `RTPDecoder` (for receiving messages), and to return a coder implementing the `Session_High` interface for sending messages. This coder (an instance of the internal class `RTPCoder`) allows the client to prepare a header to be prepended to an outgoing message, incrementing the sequence number and timestamp as needed.

Use Case: Event Channel

Introduction An *event channel* is a communication channel on which two types of entities may be connected: *event sources* and event *consumers*. When a source produces an event, in the form of a message, this message is delivered to all the consumers connected to the channel. The channel itself may be regarded as both an event source and consumer: it consumes events from the sources and delivers them to the consumers.

Two communication patterns may be used:

- the "push" pattern, in which events are pushed by the sources into the channel, which in turn pushes them into the consumers;
- the "pull" pattern, in which events are explicitly requested (pulled) by the consumers from the channel, which in turn tries to pull them from a source (this operation may block until events are produced).

In this example, we use the push pattern. The interface provided by the channel to event sources is that of a representative (a proxy) of the interface provided by the event consumers (Figure 4.16). Particular implementations of the event channel may provide specific guarantees for message delivery, in terms of reliability, ordering, or timeliness.

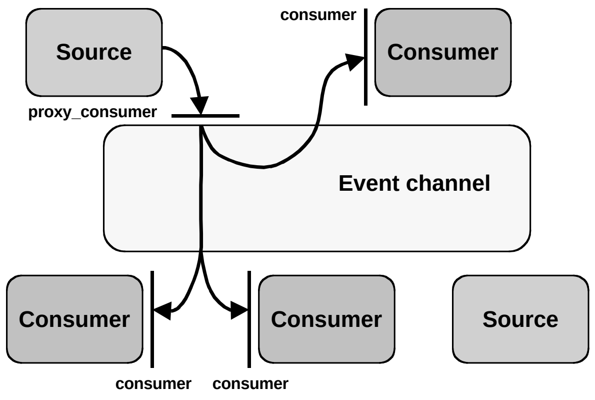


Figure 4.16. An event channel based on the "push" pattern

Jonathan provides two implementations of a simple event channel, using a CORBA implementation (David, see 5.5) and a Java RMI implementation (Jeremie, see 5.4), respectively. Both rely on the same binder and event model; they essentially differ by the communication protocol. This presentation is based on the Jeremie version of the event channel.

The event channel provides the following interface, which defines the methods needed by event sources and consumers to connect to the channel using the "push" pattern:

```
public interface EventChannel extends java.rmi.Remote { // for use by RMI
```

```
  void addConsumer(Object consumer)
    throws JonathanException;  // adds a new consumer to the event channel
  void removeConsumer(Object consumer)
    throws JonathanException;  // removes a consumer (will no longer receive events)
  Object getConsumerProxy()
    throws JonathanException;  // returns a consumer proxy to be used by a source
}
```

Event channels are built by event channel factories. Class `EventChannelFactory` provides, among others, the following method:

```
  public EventChannel newEventChannel(String address, int port, String type)
      throws JonathanException
```

which creates a new event channel built on the supplied host address and port. It also provides a class that implements the `EventChannel` interface defined above.

Using an Event Channel Like in the "Hello World" case described in the binding tutorial, a name server (in Jeremie, a registry) is used to register event channels. In this example, an event source creates the channel, registers it under a symbolic name of the form `<//<registry host><channel name>`, and starts producing events. A prospective consumer retrieves a channel from the name server using its symbolic name, and subscribes to the channel; it then starts receiving messages transmitted on this channel.

The core of the `main` method of `NewsSource`, the program for event sources, is:

```
  EventChannel channel;
  EventChannelFactory channelFactory =
     EventChannelFactoryFactory.newEventChannelFactory(NewsSource.class);
  channel=channelFactory.newEventChannel(address,port,"NewsChannel");
  Naming.rebind("//"+args[3]+"/"+args[2], channel); // //<registry host><channel name>
  System.out.println("Ready...");

  NewsTicker ticker=(NewsTicker)channel.getConsumerProxy();
  NewsSource producer=new NewsSource(ticker);
  producer.produce();// produce is the event generator method of NewSource class
                     // it includes a call to the NewsTicker method: ticker.latestNews
```

The core of the `main` method of `NewsConsumer`, the program for event consumers, is:

```
  System.setSecurityManager(new RMISecurityManager());
  channelName=args[0];
  EventChannel channel=
     (EventChannel) Naming.lookup("//" + args[1] + "/" + channelName);
  if(channel==null) {
     System.err.println("Channel "+ channelName + " not found");
     System.exit(1);
  }
  channel.addConsumer(new NewsConsumer());
```

The `NewsTicker` interface shared by source and consumer classes consists of the method `latestNews(String msgs[])` activated when an event is produced. This method is implemented in the `NewsConsumer` class; it simply displays the incoming messages on the screen:

```
public void latestNews(String msgs[]) {
   for(int i=0;i<msgs.length;i++)
    System.out.println(msgs[i]);
}
```

Event Channel Implementation The implementation of the event channel relies on two components that closely interact: the event channel factory, `EventChannelFactory`, and the event binder, `EBinder`. The role of the factory is to deliver implementations of the `EventChannel` interface, both in the form of actual instances and in the form of stubs. The role of the binder is to provide an interface allowing both sources and consumers to connect to an event channel. The current implementation is hardwired to work with the RTP protocol on top of the IP Multicast protocol.

`EBinder` provides a specific class of identifiers, `EIds`. An `EId` designates an event channel built on a particular IP address and port number used by the underlying IP Multicast protocol. The two main methods are `getProtocolGraph()` and `bind()`, which are used as follows:

- An event source that needs to connect to an event channel designated by `EId channel_id` executes `channel_id.bind()`, which returns a (`Session_High`) session on which the source will send events.
- An event consumer that needs to connect to an event channel designated by `EId channel_id` executes `EBinder.bindConsumer (consumer, channel_id)`, where `consumer` is the (`Session_Low`) interface provided by the consumer to receive events (note that this is a "push" interface, including a method `send`). `bindConsumer` is implemented as follows:

  ```
  ProtocolGraph protocol_graph = channel_id.getProtocolGraph();
  protocol_graph.export(consumer);
  ```

The usual export-bind pattern is again used here; `getProtocolGraph()` returns the protocol graph of the underlying RTP protocol, itself relying on IP Multicast; `bind()` returns a stub, created by a stub factory using the underlying RTP session.

An `EventChannelFactory` is created by an `EventChannelFactoryFactory`, which associates it with an `EBinder`. It implements special instances of stubs and skeletons targeted towards one-way invocation, and provides a specific implementation of `EventChannel`, that works as follows.

An instance of `EventChannel` is created by the constructor:

```
EventChannelImpl(String address, int port, String type,
      EventChannelFactory binder) throws JonathanException  {
  id=(EBinder.EId)binder.getEBinder().newId(address,port,type);
  SessionIdentifier ep=id.getSessionIdentifier();
```

```
293    Context hints = JContextFactory.instance.newContext();
294    hints.addElement("interface_type",String.class,type,(char) 0);
295    proxy=binder.newStub(ep,new Identifier[] {id}, hints);
296    hints.release();
297    this.binder=binder;
298  }
```

On line 291, a new `EId` is created by the `EBinder`, with the given address and port. On line 292, a new session is associated with this `EId`, using the underlying RTP and IP Multicast protocols. Finally, a proxy (stub) is created using this session. This proxy is ready to be delivered to any source willing to send events to the channel, through the following method:

```
328  public Object  getConsumerProxy() throws JonathanException {
329     return proxy;
330  }
```

A consumer connects to the event channel by calling the following method:

```
301    public void addConsumer(Object consumer) throws JonathanException {
302       a: if(binder==null) {
303          if (proxy instanceof StdStub) {
304             Identifier[] ids =
305                ((JRMIRef) ((StdStub) proxy).getRef()).getIdentifiers();
306             Object cid;
307             for (int i = 0; i < ids.length; i++) {
308                cid = ids[i];
309                while (cid instanceof Identifier) {
310                   if (cid instanceof EBinder.EId) {
311                      id = (EBinder.EId) cid;
312                      binder = (EventChannelFactory) id.getContext();
313                      break a;
314                   } else {
315                      cid = ((Identifier) cid).resolve();
316                   }
317                }
318             }
319          }
320          throw new BindException("Unbound channel");
321       }
322       binder.getEBinder().bindConsumer(new OneWaySkeleton(consumer),id);
323    }
```

The key operation here is on line 322: the consumer builds a skeleton that will act as an event receiver interface for it, then binds that skeleton to the event channel using the `bindConsumer` method of the `EBinder`, as described above. Lines 302 to 321 illustrate another (classical) situation in the binding process: if the factory associated with the channel is not known (e.g., because the event channel reference has been sent over the network), the method tries to retrieve it using one of the identifiers associated with the stub, by iteratively `resolving` the identifier chain until the factory (binder) is found, or until the search fails.

4.6.4 Conclusion

Jonathan is based on the ideas introduced by the x-kernel, but improves on the following aspects:

- It follows a systematic approach to naming and binding through the use of the export-bind pattern (3.3.33.3.3)
- It is based on an object-oriented language (Java); the objects defined in the framework are represented by Java objects.

The aspects related to deployment and configuration have not been described. Jonathan uses configuration files, which allow a configuration description to be separated from the code of the implementation.

Although Jonathan itself is no longer in use, its main design principles have been adopted by Carol [Carol 2005], an open source framework that provides a common base for building Remote Method Invocation (RMI) implementations (5.4).

4.7 Historical Note

Communication systems cover a wide area, and a discussion of their history is well outside the scope of this book. In accordance with our general approach, we restrict our interest to networking systems (the infrastructure upon which middleware is built), and we briefly examine their evolution from an architectural point of view.

Although dedicated networks existed before this period, interest in networking started in the mid 1960s (packet switching techniques were developed in the early 1960s and the first nodes of the ARPAnet were deployed in 1969). The notion of a protocol (then called a "message protocol") was already present at that time, as a systematic approach to the problem of interconnecting heterogeneous computers.

As the use of networks was developing, the need for standards for the interconnection of heterogeneous computers was recognized. In 1977, the International Standards Organization (ISO) started work that led to the proposal of the 7-layer OSI reference model [Zimmermann 1980]. By that time, however, the main protocols of the future global Internet (IP, TCP, and UDP) were already in wide use, and became the *de facto* standards, while the OSI model essentially remained a common conceptual framework.

The Internet underwent a major evolution in the mid-1980s, in response to the increase in the number of its nodes: the introduction of the Domain Name System (DNS), and the extension of TCP to achieve congestion control. A brief history of the Internet, written by its main actors, may be found in [Leiner et al. 2003].

The 1990s were marked by important changes. The advent of the World Wide Web opened the Internet to a wide audience and led the way to a variety of distributed applications. At about the same time, the first object-based middleware systems appeared, stressing the importance of application-level protocols. New applications imposed more stringent requirements on the communication protocols, both on the functional aspects (e.g., multicast, totally ordered broadcast) and on quality of service (performance, reliability, security).

In the 2000s, two main trends are visible: the development of new usages of the Internet, exemplified by overlay networks (e.g., [Andersen et al. 2001]) for various functions, from content distribution to peer to peer systems; and the rise of wireless communications [Stallings 2005], which generates new forms of access to the Internet, but also wholly new application areas such as sensor networks [Zhao and Guibas 2004] or spontaneous ad hoc networks. Both trends are leading to a reassessment of the basic services provided by the Internet (whose main transport protocols are still basically those developed in the 1980s) and to the search for new architectural patterns, of which the SensorNet research (4.5.2) is a typical example. Peer to peer and ad hoc networks call for new application-level protocols, such as gossip-based algorithms [Kermarrec and van Steen 2007a].

On the architecture front, the x-kernel [Hutchinson and Peterson 1991, Abbott and Peterson 1993] and related systems opened the way to a systematic approach to the modular design of communication protocols. Initially based on objects in the 1990s, this approach followed the transition from objects to components and is now applied to the new above-mentioned applications areas. The trend towards fine-grain protocol decomposition (as illustrated in 4.5.2) is imposed both by economy of resource usage and by the search for increased adaptability and customization. The strive for a balance between abstraction and efficiency, an everlasting concern, is still present under new forms.

Chapter 5

Distributed Objects

Organizing a distributed application as a set of objects is a powerful architectural paradigm, which is supported by a number of infrastructures and tools. The development of middleware has in fact started with the advent of distributed object systems. This chapter presents the main patterns related to distributed objects, and illustrates them with two systems that are representative of current technology, Java RMI and CORBA. The main software frameworks used in the implementation of these systems are presented, with reference to open source implementations.

5.1 Distributing Objects

Organizing a distributed application as a (dynamically evolving) set of objects, located on a set of sites and communicating through remote invocations (2.2.2), is one of the most important paradigms of distributed computing, known as the distributed objects model. In this section, we first present an overview of this model (5.1.1). We next introduce the main interaction scheme between objects, the remote object call (5.1.2). We finally present the user's view of this mechanism (5.1.3).

5.1.1 Overview

The distributed objects model brings the following expected benefits.

- Application designers may take advantage of the expressiveness, abstraction, and flexibility of an object model (as described in 2.2.2).
- Encapsulation allows an object's implementation to be placed on any site; object placement may be done according to specific criteria such as access locality, administration constraints, security, etc.
- Legacy applications may be reused by encapsulating them in objects, using the Wrapper pattern (2.3.3).
- Scalability is enhanced by distributing processing power over a network of servers, which may be extended to accommodate an increasing load.

Note that, in the remote objects model, objects are the units of distribution, i.e. an individual object resides on a node in its entirety (Figure 5.1).

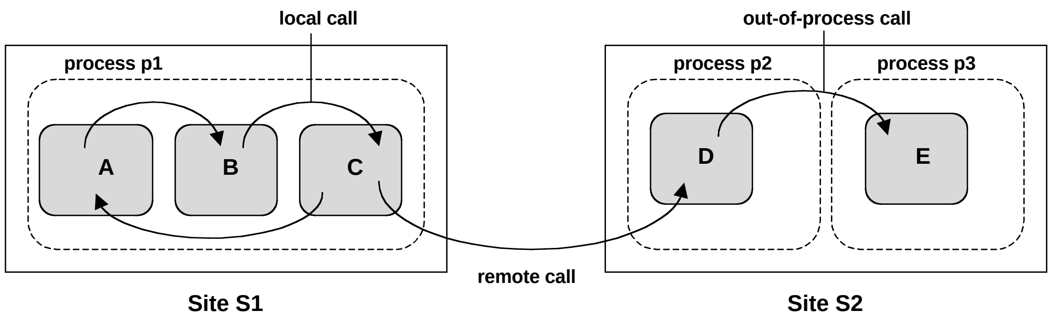


Figure 5.1. Distributed objects

Other models for distributing objects have been proposed, such as:

- The *fragmented objects* model, in which an object may be split in several parts, located on different nodes, and cooperating to provide the functionality of the object. An example of a fragmented object is a distributed binding object (3.3.2). An example of a system using this model is Globe [van Steen et al. 1999]. This model will not be considered further in this book.
- The *replicated objects* model, in which several copies, or replicas, of a given object may coexist. The motivation for replicated objects is to increase availability and to improve performance. However, the replicas of an object must be kept consistent, which entails additional cost. Replicated objects are examined in Chapter 11.
- The *migratory* (or *mobile*) objects model, in which an object may move from one node to another one. Object mobility is used to improve performance through load balancing, and to dynamically adapt applications to changing environments.

These models may be combined, e.g. fragmented objects may also be replicated, etc.

A distributed application using remote objects is executed as a set of processes located on the nodes of a network. An object's method is executed by a process or a thread (in some models, objects may also be shared between processes), and may include calls to other objects' methods. For such inter-object method calls, three situations may occur (Figure 5.1).

- The calling and called objects are in the same process (e.g. objects A and B): this is a *local invocation*.
- The calling and called objects are executed by different processes on the same site (e.g. objects D and E): this is an *out-of-process invocation*.
- The calling and called objects are on different nodes (e.g. objects C and D): this is a *remote invocation*.

Local invocations are done like in a non-distributed object system. Non-local forms of invocation rely on an *object request broker* (ORB), a middleware that supports distributed objects. This term has been introduced for the CORBA architecture, but it applies to other object systems as well. An ORB has the following functions.

- Identifying and locating objects.
- Binding client to server objects.
- Performing method calls on objects.
- Managing objects' life cycle (creating, activating, deleting objects)

In the rest of this section, we give a first outline of the operation of a non-local invocation. More details on the internals of an ORB are given in Sections 5.2 and 5.3.

5.1.2 Remote Object Call: a First Outline

An application using remote objects is typically organized using the client-server model: a process or thread executing a method of a client object sends a request to a (possibly remote, or out-of-process) server object in order to execute a method of that object.

The overall organization of a method invocation on a remote object, shown on Figure 5.2, is similar to that of an RPC, as described in Chapter 1. It relies on a stub-skeleton pair. In contrast with RPC, the stub and the skeleton are objects in their own right. Take the example of the call from object C to the remote object D. The stub for D, on client C's site, acts as a local representative, or proxy, of object D. It therefore has the same interface as D. It forwards the call to D's skeleton on D's site, which performs the actual method invocation and returns the results to C, via the stub.

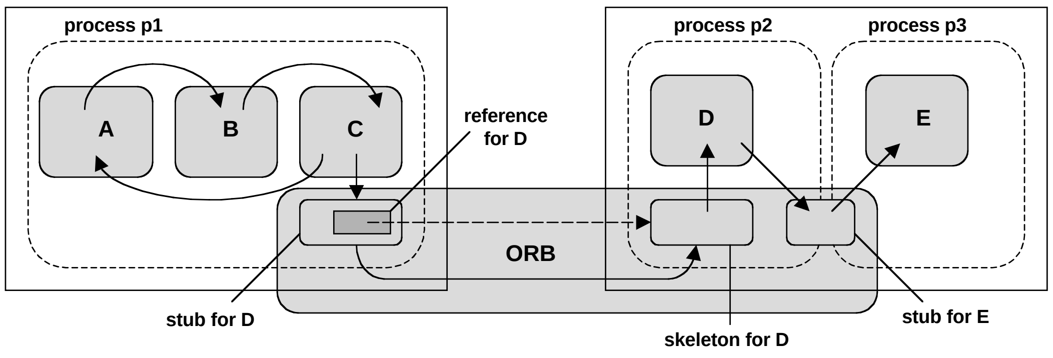


Figure 5.2. Performing non-local calls

In order to be able to forward the invocation, D's stub contains a reference to D (more precisely, to D's skeleton). A *reference* to an object is a name that allows access to the object (cf. 3.1); this name is therefore distribution-aware, i.e. it contains information allowing the object to be located (e.g. network address and port number). Object references are further developed in 5.2.1.

An out-of-process call on the same node (e.g. from object D to object E) could in principle be performed as a remote invocation. However, it is possible to take advantage of the fact that the objects are co-located, using e.g. shared memory. Thus an optimized stub-skeleton, bridging the client and server address spaces, is usually created in that case.

Let us go into the details of a remote method invocation. The client process invokes the method on the local stub of the remote object (recall that the stub has exactly the same interface as the remote object and contains a reference to that object). The stub marshalls the parameters, constructs a request, determines the location of the remote object using its reference, and sends the request to the remote object (more precisely, to the object's skeleton). On the remote object's site, the skeleton performs the same function as the server stub in RPC: unmarshalling parameters, dispatching the call to the address of the invoked method, marshalling returned values, and sending them back to the stub. The stub unmarshalls the returned values and delivers them to the client process, thus completing the call. This is represented on Figure 5.3.

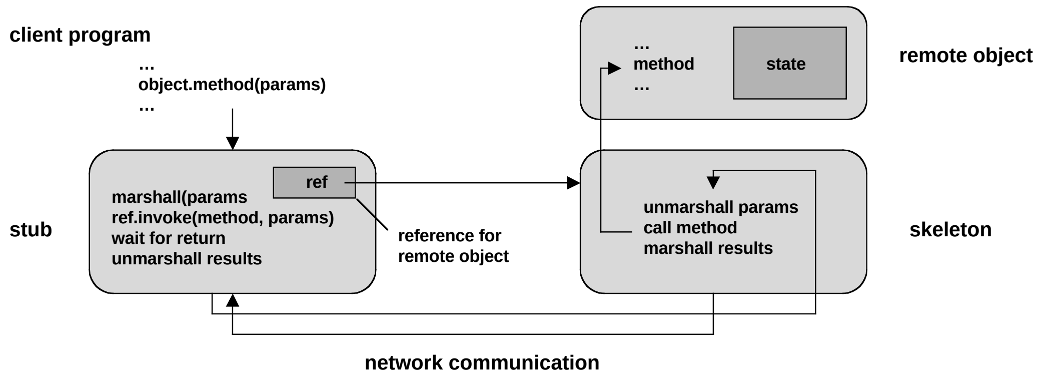


Figure 5.3. Invoking a remote object

An important difference with RPC is that objects are dynamically created (details in next section).

5.1.3 The User's View

Remote object infrastructures attempt to hide distribution from the user, by providing access and location transparency. However, some aspects of distribution remain visible, e.g. object creation through factories.

According to the encapsulation principle, an object, be it local or remote, is only accessible through an interface. Object interfaces are described by an *Interface Description Language* (IDL). The function of an IDL for objects is similar to that of the IDL used in RPC systems. A set of IDL descriptions is used

- to describe the interfaces of the objects being used by an application, thus helping the design process, allowing consistency checks, and providing useful documentation;
- to serve as input for stub and skeleton generation, both static (before execution) and dynamic (during execution).

There is no single format for an IDL. The syntax of most IDLs is inspired by that of a programming language. Languages that include interface definitions, such as Java and C#, define their own IDL.

An example of a general purpose IDL is the OMG IDL, used for programming in CORBA. This IDL may be "mapped" on various programming languages by using appropriate generation tools. For example, using the IDL to C++ mapping tools, stubs and skeletons in C++ may be generated from IDL description files. Client and server executable programs may then be generated, much like in an RPC system (1.3.3).

The main distribution-aware aspect is object creation and location. Creating a remote object cannot be done through the usual object instantiation mechanism, which involves memory allocation and is not directly applicable to a remote node. Creating a remote object is done through an *object factory* (2.3.2), which acts as a server that creates objects of a specified type. The reference of the created object is returned to the client (Figure 5.4 (a)).

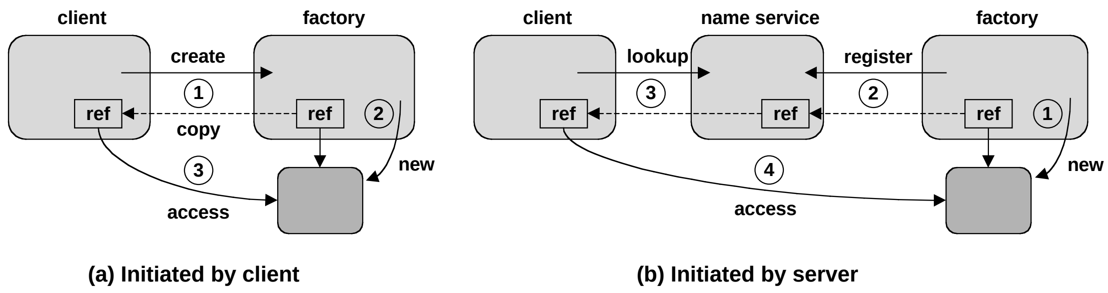


Figure 5.4. Object creation

An object may also be created at the server's initiative. In that case the server usually registers the object (more precisely, a reference to the object) in a name service, to be later retrieved by the client (Figure 5.4 (b)). This is an instance ot the general binding mechanism described in 3.3.4.

As a conclusion, a remote object may only be accessed by a client program through a reference, which in turn may be obtained in several ways: as a return parameter of a call, through an object factory, through a name service. The first two ways imply access to existing objects that in turn have to be located; thus, ultimately, retrieving an object relies on a name service. Examples are described in the case studies.

5.2 Remote Object Call: a Closer View

A first outline of the execution of a remote object call is given in Section 5.1.2. A few points need to be further clarified in this scheme.

- What information should be contained in an object reference?
- How is the remote object activated, i.e how is is associated with a process or thread that actually performs the call?
- How are parameters passed, specially if the parameters include objects?

These points are considered in the rest of this section.

5.2.1 Object References

In Chapter 3, we mentioned that the two functions of a name (identification and access) have conflicting requirements, which leads to use different forms of names for these functions. We consider here the means of providing access to an object; an information that fulfills this function is called an *object reference*.

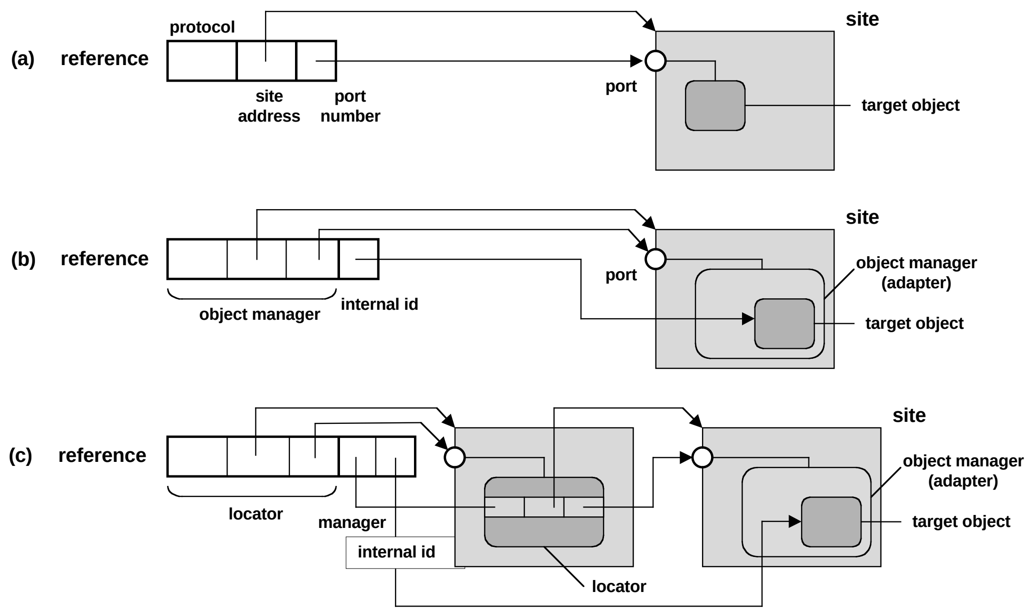


Figure 5.5. Object references

Recall that an object may only be accessed through its interface. Therefore an object reference must also refer to the object's interface. Since an object may be remote, the reference must provide all the information that is needed to perform a remote access to the object, i.e. the network location of the object and an access protocol.

Figure 5.5 shows three examples of reference implementations. In the simplest case (a), the location is a network address (e.g. host address and port number). However, this scheme lacks flexibility, since it does not allow the target object to be relocated without changing its reference. Therefore, indirect schemes are usually preferred. In example (b), the reference contains the network address of a server that manages the object (an object adapter), together with an internal identification of the object on this adapter – more on this in Section 5.2.3. To provide additional flexibility (e.g. allowing a reference to remain valid after the shutdown and restart of a server), an additional level of indirection may be introduced. In example (c), the reference contains the network address of a locator for the adapter, together with the identity of the adapter and the internal object identification. This allows the adapter to be transparently relocated on a different site, updating the locator information without changing the reference.

Recall that a stub, i.e. a proxy (2.3.1) for a remote object, contains a reference to the object it represents. Actually, the stub itself may be used as a reference. This has the advantage that the reference is now self-contained, i.e. it may be used anywhere to invoke the remote object. In particular, stubs may be used for passing objects as parameters. This aspect is further developed in Section 5.2.4.

Object references may be obtained through several mechanisms.

- When an object is created, the object factory returns a reference to the object;
- The `bind` operation on the name of an object (Chapter 3), if successful, returns a reference to the bound object;
- As a special case of the above, looking up an object in a name server or trader returns a reference to the object (if found).

Two points should be noted regarding object references.

The validity domain of a reference. A reference is not usually universally valid in space and time.

In a closed system relying for instance on a local area network using a fixed protocol suite, object references may use a restricted format, in which some elements are implicit (for example the protocols used). Such references may not be exported outside the system. However, if references are to be passed across interconnected systems using different protocols or representations, then their format must accommodate the specification of the variable elements. For example, the Interoperable Object Reference (IOR) format, used in CORBA systems (5.5), allows for communication between objects supported by different ORBs.

In addition, a reference to an object is only valid during the lifetime of the object. When an object is removed, any reference to it becomes invalid.

Reference vs identity. Recall that an object reference is not intended to identify an object, but only to provide a means of accessing it. Therefore a reference cannot in general be used as an identifier (e.g. different references may give access to the same object). Object identification must be dealt with separately (e.g. an object may carry a unique identifier as part of its state, and a method may be provided to return this identifier[1]).

One could imagine including a unique identification for an object in a reference. However, this would defeat the primary purpose of a reference, which is to give access to an object that provides a specified functionality. To understand why, consider the following scenario: Suppose *ref* is a reference for object O; it contains the network address and port number of a location server, plus a key for O on this server. The key is associated with the actual network location of O (e.g. again network address and port number). Suppose the server that supports O crashes. A fault tolerance mechanism is activated, which updates the location server with a new copy of O (say O_1). While O and O_1 are different objects, as regards identity, they are equivalent from the client's point of view (provided consistency is indeed ensured), and the reference *ref* may at different instants give access to either of them.

[1]note that, in this case, we need access to the object in order to check its identity.

5.2.2 The Object Invocation Path

Remote object invocation is actually somewhat more complex than described in Section 5.1.2, because of two requirements.

- Abstraction. The actual service described by the remote invocation interface may be provided by a variety of concrete mechanisms, which differ by such aspects as activation mode (whether a new process or thread needs to be created), dynamic instance creation (whether the target object already exists or needs to be created at run time), persistence (whether the target object survives through successive executions), etc. This results from the encapsulation principle, as presented in 2.2.2: a service described by an abstract interface is implemented by an object whose specific (concrete) interface depends on the details of the service provision mechanism. In other words there is a distinction between a server (a site that provides a set of services to remote clients), and a *servant* (an object, managed by a server site, that actually implements a specific service).
- Portability and Interoperability. A distributed application may need to be ported to various environments, using ORBs provided by different vendors. An application may also involve communication between different ORBs. Both situations call for the definition of standards, not only for the interface between client and server, but for internal interfaces within the ORB.

As a result, the overall remote invocation path is organized as shown on Figure 5.6. This is a general picture. More details are provided in the case studies.

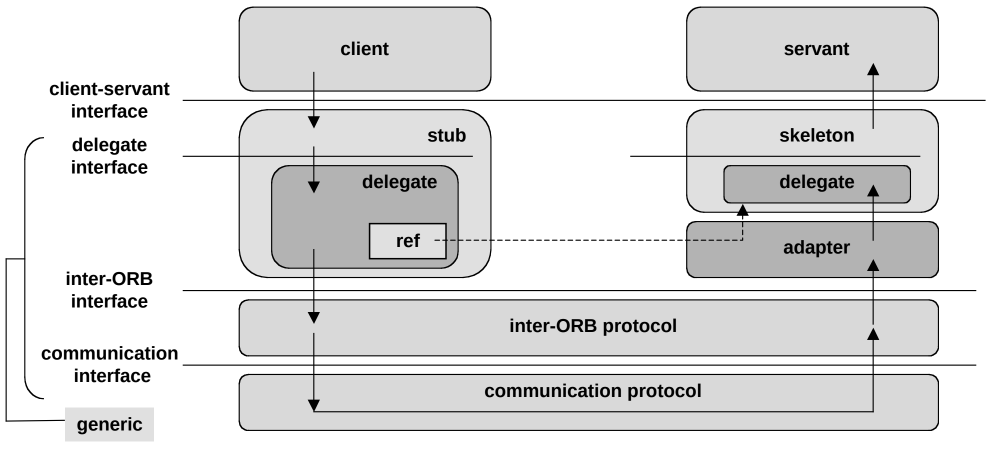


Figure 5.6. The invocation path and its main interfaces

The motivations for this organization result from the above stated requirements.

On the client side, a new interface is defined inside the stub. While the "upper" interface of the stub is application-specific (since it is identical to that of the remotely called object), this internal interface is generic, i.e. application-independent. This means that both the interface entry points and the format of the data may be standardized.

Typical methods provided by the delegate interface are `create_request` (constructing a invocation request in a standard form) and `invoke` (actually performing the invocation).

Likewise, the "lower" interface of the stub, connecting it to the network, is generic. An example of a generic operation is `SendRequest`, whose parameters are the reference of the called object, the name of the method, the description of the parameters of the called method. An example of a standard is GIOP (General Inter-ORB Protocol), further described in Section 5.5.1.

On the server side, the interface transformation between service (as described by a generic ORB interface) and servant (a specific, application-dependent interface) is again done through a server delegate, part of the skeleton. The skeleton itself (and thus the servant) is located through a piece of software called an *object adapter* (2.3.3). While delegates are part of the internal organization of a stub, and are never explicitly seen by a user, adapters are usually visible and may be directly used by applications.

5.2.3 Object Adapters

An adapter performs the following functions.

- to register servant objects when they are created or otherwise installed on the server;
- to create object references for the servants, and to find a servant object using its reference;
- to activate a servant object when it is called, i.e. to associate a process with it in order to perform the call.

There are three main ways of installing a servant object on a server. The first way consists of creating the servant, using an object factory. The second way consists of importing the servant from another site (assumed it is available there), by moving or copying it to the server site. The third way consists of constructing the object from its elementary parts, i.e. a state and a set of functions (methods) operating on the state. This latter mode is used when an existing (or *legacy*) application, not written in object style, must be reused in an object-oriented setting. In that case, the legacy application needs to be "wrapped up", i.e. its contents needs to be encapsulated to make only visible an object-style interface. This is another instance of interface transformation, which is again performed by an adapter (this explains why adapters are also called *wrappers*).

Several different adapters may coexist on a server, for example to provide different policies for process activation. A servant object may now be identified by providing an identification for its adapter together with an internal identification of the object within the adapter. This information is part of the object's reference.

The operation of a remote object invocation may now be refined as shown on Figure 5.7.

Starting from the stub as above, the call is directed to the adapter, providing the internal identification of the servant object within the adapter. The adapter must locate the object (more precisely, the object's skeleton), activate the object if needed, and forward the call to the skeleton. The call is then performed as described in Section 5.1.2.

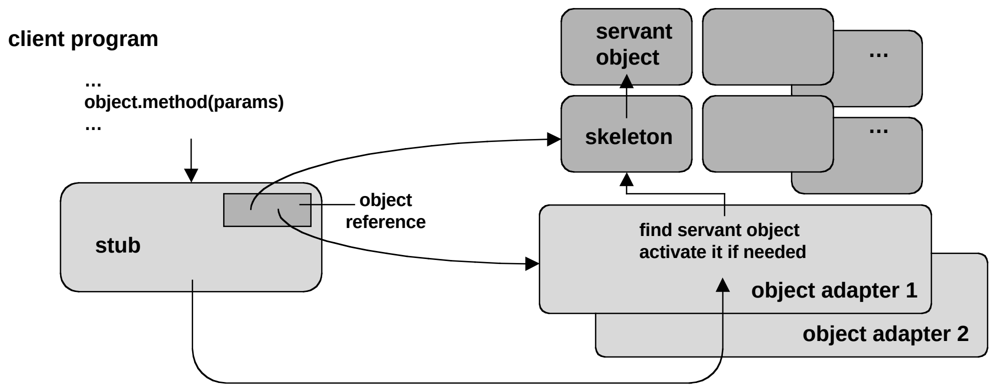


Figure 5.7. Invoking a remote object through an adapter

5.2.4 Parameter Passing

Passing parameters in a remote object system poses two kinds of problems.

- Transmitting values of elementary types through a network. This is the "marshalling-unmarshalling" problem.
- Passing objects as parameters. This raises the issue of the passing mode (reference vs value).

The marshalling problem is similar to that found in RPC systems (1.3.2). It is solved by creating marshallers and unmarshallers for the elementary types, using a serializable format (a byte sequence) as the intermediary form.

We now consider passing objects as parameters. Since the standard way of calling a method on a remote object is through a reference to it, the usual mode of passing an object as a parameter is by reference (Figure 5.8 (a)). The reference may have any of the formats presented in 5.2.1, including a stub. It needs to be in a serializable form to be copied on the network.

If the object being passed is on the site of the calling method (Figure 5.8 (b), then any access to that object involves a remote invocation from the called site to the calling site. If this access does not modify the object, one may consider passing the object by value, i.e. sending a copy of it on the called site (Figure 5.8 (c). However, this implies that the state of the object may indeed be marshalled and copied on the network[2]. Choosing the mode of passing for a (read-only) local object is a trade-off between the cost of marshalling and sending the copy of the object and the cost of remotely invoking the object from the called site, which depends on the size of the object's state and on the frequency of access. Some aspects of this trade-off are discussed in [Spiegel 1998].

[2]For example, in Java, an object must implement the `java.io.Serializable` interface to allow its state to be marshalled.

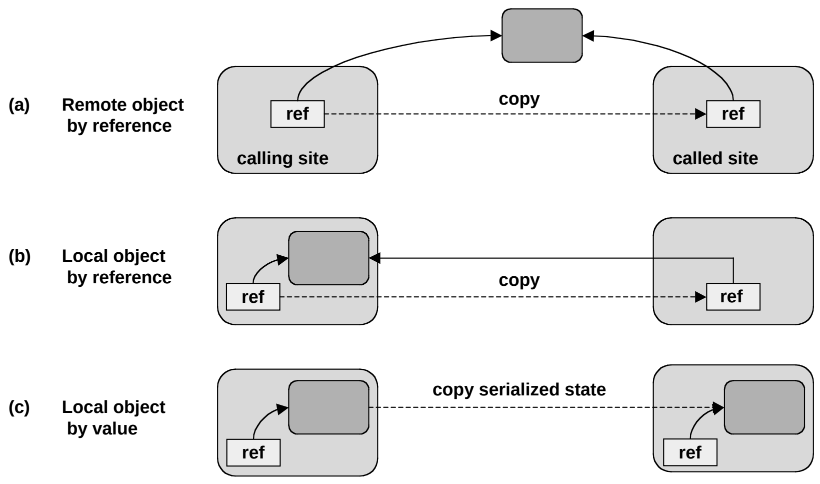


Figure 5.8. Passing an object as a parameter

5.3 Inside an Object Request Broker

In this section, we present a detailed view of the internal operation of an object request broker, emphasizing the binding and communication aspects. While this description is inspired by the organization of the Jonathan ORB, we attempt to present a generic view of the structure and operation of an ORB. Specific examples are presented in Sections 5.4 and 5.5.

A thorough treatment of the patterns involved in remote invocation may be found in [Völter et al. 2004].

5.3.1 The Mechanics of Object Invocation

The use of generic (i.e. application-independent) interfaces in the invocation path to a remote object has been motivated in 5.2.2. The main generic interface has been specified by the OMG in the General Inter-ORB Protocol (GIOP). While this specification is independent of the underlying transport layer, its dominant implementation, called IIOP (Internet Inter-ORB Protocol), is built over TCP/IP.

The initial purpose of GIOP was to allow different CORBA implementations to interoperate, leaving each ORB vendor free to choose an internal transport protocol. However, in practice, GIOP (essentially under the IIOP form), is also used for the internal implementation of ORBs (both CORBA and others such as Java RMI).

We do not intend to describe the full GIOP specification (see [OMG 2003]). We only identify the main elements that are needed to understand the basic mechanics of object binding and invocation.

The GIOP specification covers three aspects.

- A common format for the data being transmitted, called Common Data representation (CDR);

- A format for the messages used for invoking remote objects;
- Requirements on the underlying transport layer.

The IIOP specification defines a mapping of GIOP on the TCP/IP transport protocol. In particular, it defines a general format for object references, called Interoperable Object References (IOR).

Eight message types are defined for object invocation, The two most important are `Request` (from client to server), and `Reply` (from server to client). The other messages fulfill auxiliary functions such as aborts, error detection, and IOR management.

A `Request` message includes a reference for the target object, the name of the invoked operation, the parameters (in marshalled form), and, if a reply is expected, an identifier for a reply holder (the endpoint on which the reply is expected). After sending the message, the calling thread is put to wait. At the receiving end, the servant is located using the reference and the request is forwarded to it.

A `Reply` message, only created if the invoked operation returns a value, contains that value, in marshalled form, together with an identification of the reply holder. When this message is received by the client, the waiting thread is activated and may retrieve the reply from the reply holder.

Note that the `Request` message may either be created by a stub, itself generated prior to the invocation, or created dynamically by directly putting its parts together at invocation time. Here we only consider the first case; dynamic request creation is examined in Section 5.6.

An outline of the invocation path may be found on Figures 5.9 and 5.10, which describe the client and server sides, respectively. The operation of the transport protocol used by the GIOP layer is not detailed (see Chapter 4).

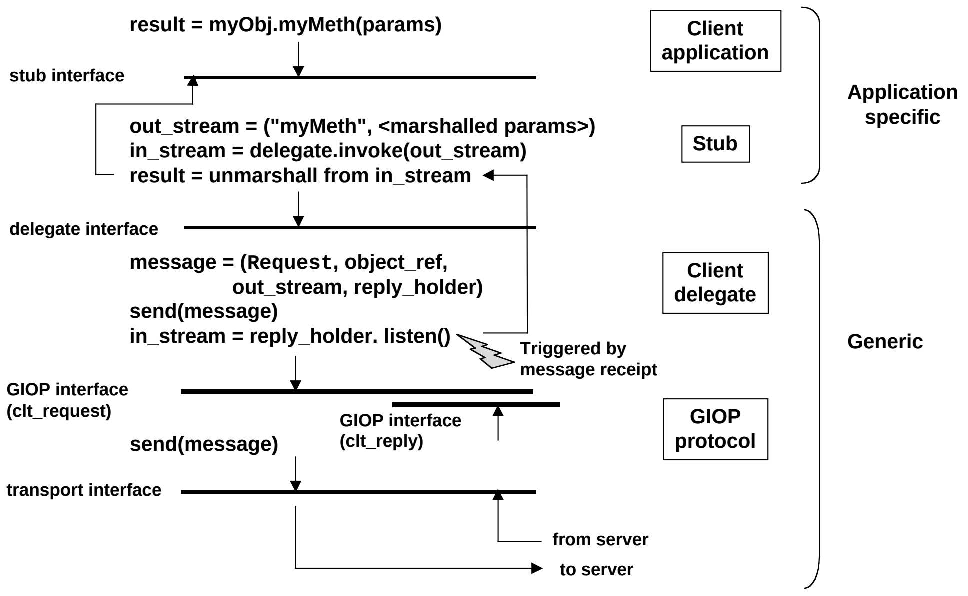

result = myObj.myMeth(params)
stub interface
out_stream = ("myMeth", <marshalled params>)
in_stream = delegate.invoke(out_stream)
result = unmarshall from in_stream
delegate interface
message = (Request, object_ref, out_stream, reply_holder)
send(message)
in_stream = reply_holder. listen()
Triggered by message receipt
GIOP interface (clt_request)
GIOP interface (clt_reply)
send(message)
transport interface
from server
to server
Client application
Stub
Client delegate
GIOP protocol
Application specific
Generic

Figure 5.9. Object invocation: client side

Above the GIOP layer, the invocation path goes through a stub (at the client end) and a skeleton (at the server end). The upper interface of the stub and the skeleton is application specific. Inside the stub and the skeleton, it is convenient to define an additional generic (application-independent) interface, to improve code reusability, leaving the application-specific part to a minimum. Therefore, in several ORB organizations, the stub and the skeleton are separated into an (upper) application-specific part (the stub or skeleton proper) and a (lower) application-independent part called a *delegate.* The interface of the delegate reifies the application-dependent interface, giving an explicit representation of the operation name and the (marshalled) parameters.

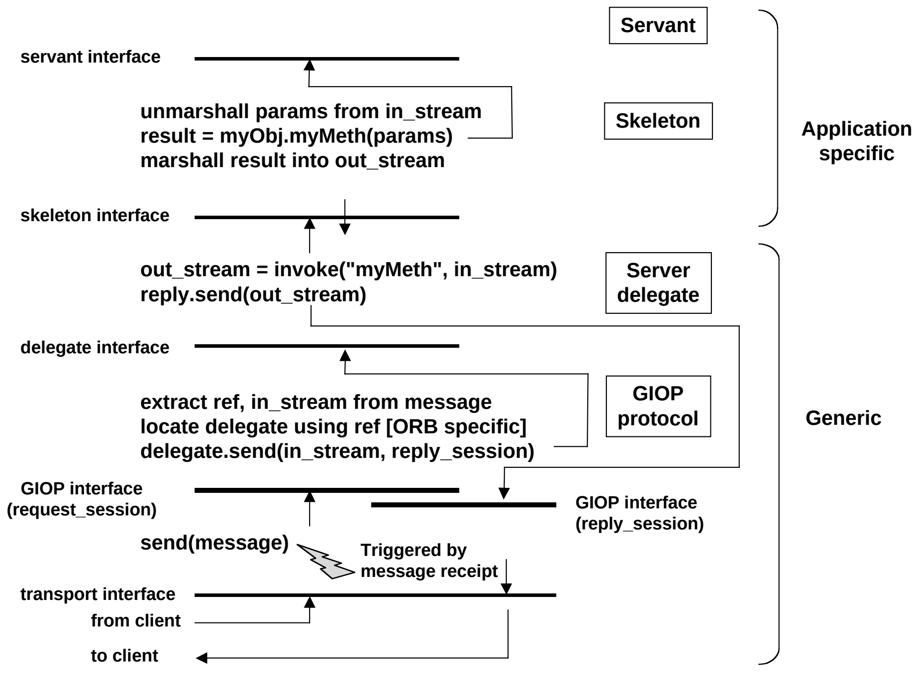

Servant
servant interface
unmarshall params from in_stream
result = myObj.myMeth(params)
marshall result into out_stream
Skeleton
Application specific
skeleton interface
out_stream = invoke("myMeth", in_stream)
reply.send(out_stream)
Server delegate
delegate interface
extract ref, in_stream from message
locate delegate using ref [ORB specific]
delegate.send(in_stream, reply_session)
GIOP protocol
Generic
GIOP interface (request_session)
GIOP interface (reply_session)
send(message)
Triggered by message receipt
transport interface
from client
to client

Figure 5.10. Object invocation: server side

An important aspect is locating the servant at the server's end, using the object reference sent in the `Request` message. This an ORB-specific issue, which is linked to the organization of object references. For example, Java RMI directly uses a [host address, port number] reference, while CORBA goes through an object adapter. More details on servant location are provided in the case studies.

5.3.2 Binding in an ORB

In the previous section, we have described the (statically generated) invocation path in an ORB. The components of this path (stub, skeleton, delegates, transport protocol endpoints) collectively form a binding object between the client and the servant.

Classes for the stub and skeleton of a remotely used object are generated from a description of the interface of this object. This description is expressed in an Interface Description Language (IDL), as explained in Section 5.1.3.

Instances of these classes are created before invocation, using appropriate stub, skeleton and delegate factories. The main information to be provided is the object reference. This is done using the `export-bind` pattern.

- `export`: a servant object is created, directly or through a factory; it is registered by the server (possibly using an adapter), thus providing a reference. This reference is then registered for future use in a name service. Parts of the binding object (the skeleton instance, and possibly the delegates) are also created at this time.
- `bind`: the client retrieves a reference for the servant, either through the name service or by any other means such as direct transmission from the server. It uses this reference to generate an instance of the stub, and to set up the path from client to server by creating the end points (sessions) of the communication path (see Chapter 4).

This process is described on Figure 5.11.

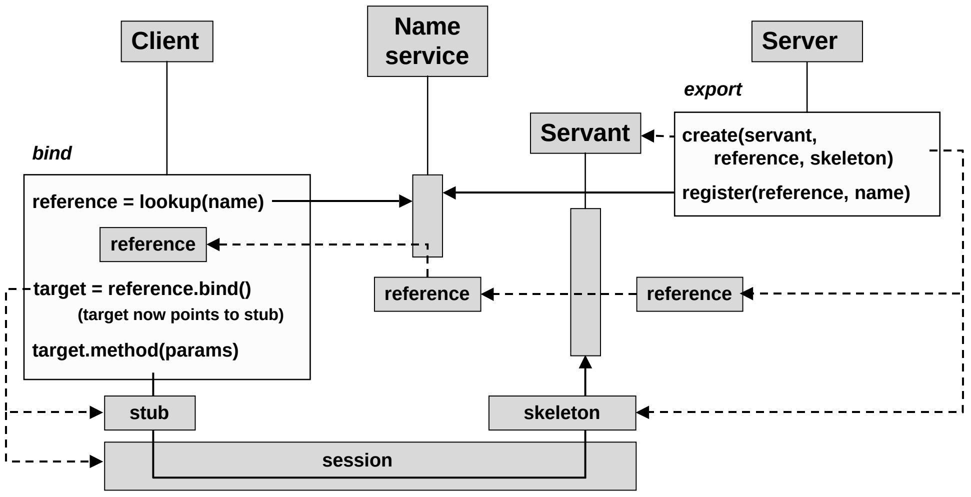


Figure 5.11. Binding for remote invocation

Specific instances of this process are described in more detail in the case studies.

5.3.3 Introduction to the Case Studies

We illustrate the internal working of an ORB with two case studies, based on open source implementations of Java RMI and CORBA, two widely used middleware systems based on the remote objects model.

We do not intend to give a detailed description of these systems, but to show the application of common design patterns. The two implementations are "personalities" built on top of the Jonathan kernel (3.4). Recall that Jonathan provides a framework for communication and binding. The core of both ORB implementations essentially consists of a binding factory, or binder (3.3.2), which itself relies on lower level tools and other binders:

- a set of tools for stub and skeleton generation;
- a binder for the specific remote invocation model;
- a binder for the communication protocol used between client and server;
- various auxiliary tools for managing elementary resources.

A global view of the structure of the ORBs is given on Figure 5.12. Some details such as the use of common resource managers (schedulers, storage allocators) are not shown. A more detailed view is provided for each case study.

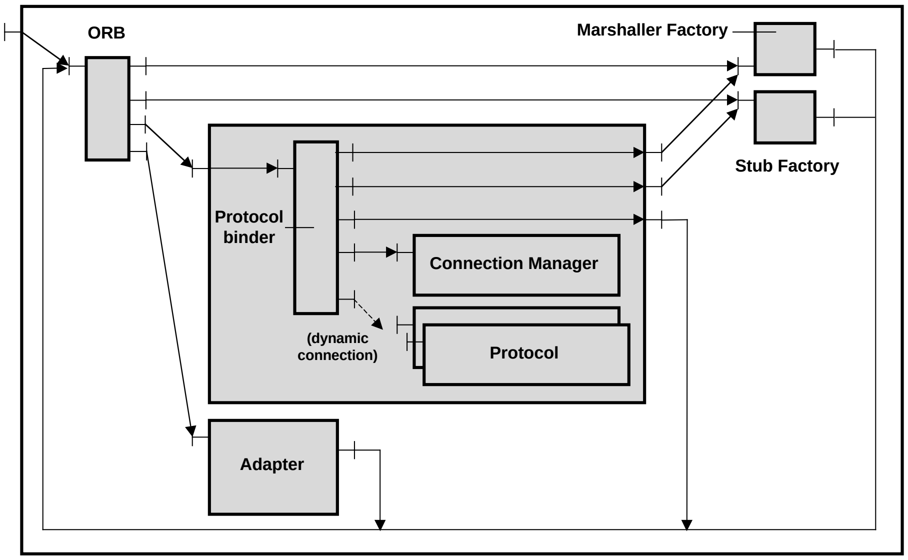


Figure 5.12. The common structure of Jonathan ORBs

The structure of the ORB is described in terms of "components" using a self-descriptive graphical formalism (the arrows connect "required" to "provided" interfaces – see Chapter 7 for more details).

5.4 Case Study 1: Jeremie, an Implementation of Java RMI

We present the Jeremie implementation of Java RMI in the Jonathan framework. After a brief introduction (5.4.1), we illustrate the principle of application development in Java RMI, through a simple example (5.4.2). We finally describe the inner working of the implementation (5.4.3).

5.4.1 Introducing Java RMI

Java Remote Method Invocation (RMI) [Wollrath et al. 1996] implements the remote objects model for Java objects. It extends the Java language with the ability to invoke a method on a remote object (a Java object located on a remote site), and to pass Java objects as parameters in method calls.

Since the Java language includes the notion of an interface, there is no need for a separate Interface Description Language (IDL). Stub and skeleton classes are generated from a remote interface description, using a stub generator (`rmic`).

Programming with remote objects is subject to a few rules of usage, as follows.

- A remote interface (the interface of a remote object) is defined like an ordinary Java interface, except that it must extend the `java.rmi.Remote` interface, which is nothing but a marker that identifies remote interfaces.
- A call to a method of a remote object must throw the predefined exception `java.rmi.RemoteException`.
- Any class implementing a remote object must create a stub and skeleton for each newly created instance; the stub is used as a reference for the object. This is usually done by making the class extend the predefined class `java.rmi.server.UnicastRemoteObject`, provided by the RMI implementation. Details are provided in Section 5.4.3.

Objects may be passed as parameters to methods. Local objects (residing on the caller's site) are passed by value, and must therefore be serializable. Non-local objects are passed by reference, i.e. a stub for the object is transmitted.

Thus, in accordance with the engineering principle discussed in 1.3.4 (see also [Waldo et al. 1997]), the Java RMI programming model does not attempt to be fully transparent, i.e. some modifications must be made to a centralized application when porting it to a distributed environment.

As mentioned in 5.1.3, a remote object system relies on a naming service. In Java RMI, this service is provided by a registry, which allows remote objects to be registered under symbolic names. The data that is registered is actually a reference for the remote object, i.e. a stub.

The registry may be located on any node. It is accessible on both the client and server node through a local interface called `Naming`. The interaction between `Naming` and the actual registry is described in 5.4.3.

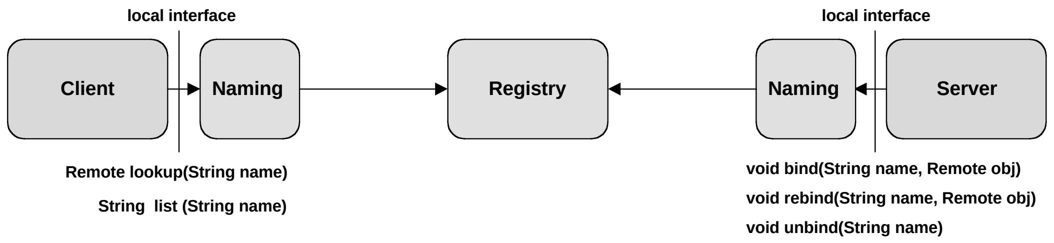

local interface
local interface
Client
Naming
Registry
Naming
Server
Remote lookup(String name)
String list (String name)
void bind(String name, Remote obj)
void rebind(String name, Remote obj)
void unbind(String name)

Figure 5.13. The RMI naming registry interface

The symbolic names have the form: `rmi://[host name][:portname]/local name` (the items between brackets are optional). A server registers references in the registry using `bind` and `rebind` and unregisters them using `unbind`. The client uses `lookup` to search the registry for a reference of a given name. The `list` method is used to list a registry's contents. The use of the registry is illustrated in the next section.

5.4.2 Developing an RMI Application

We present the main steps in developing an RMI application. Since emphasis here is on the internal working of the RMI system, we do not attempt to discuss a realistic application; we use the minimal "Hello World" example.

The centralized version of this example is presented below (the programs are self-explanatory).

```
// Hello Interface
     public interface Hello {
       String sayHello();}
     };
```

```
// Hello Implementation
    class HelloImpl implements Hello {
      HelloImpl() {   // constructor
        };
      public String sayHello() {
        return "Hello World!";
        };
    }
```

```
// Hello Usage
     ...
     Hello hello = new HelloImpl ();
     hello.sayHello();
```

In the distributed version, the client and the server run on possibly different machines. In order to allow this new mode of operation, two main problems must be solved: the client must find the location of the `hello` object (the target object of the invocation); the client must access the target object remotely.

Here is an outline of the distributed version (some details are omitted, e.g. catching exceptions, etc.). First the interface.

```
public interface Hello extends Remote {
   String sayHello() throws RemoteException;
}
```

The server program contains the implementation of the `Hello` interface and the main program. Objects are located through a naming service (the registry), which is part of the RMI environment. The server creates the target object and registers it under a symbolic name (5.4.1). The `rebind` operation does this, superseding any previous associations of the name. Note that the URL prefix of the name is `jrmi` (for the Jeremie version of RMI)

```
class HelloImpl extends UnicastRemoteObject implements Hello {
    HelloImpl() throws RemoteException {
    };
  public String sayHello() throws RemoteException {
      return "Hello World!";
  };
```

```
public class Server {
  public static void main (...) {
    ...
    Naming.rebind("jrmi://" + registryHost + "/helloobj", new HelloImpl());
    System.out.println("Hello Server ready !");
  }
}
```

The client program looks up the symbolic name, and retrieves a stub for the target object, which allows it to perform the remote invocation. The client and server must agree on the symbolic name (how this agreement is achieved is not examined here).

```
...
Hello obj = (Hello) Naming.lookup("jrmi://" + registryHost + "/helloobj");
System.out.println(obj.sayHello());
```

The registry may itself be a remote service (i.e. running on a machine different from that of the client and the server). Therefore, both the client and server use a local representative of the registry, called `Naming`, which locates and calls the actual registry, as described in 5.4.1. This allows the registry to be relocated without modifying the application.

The interaction diagram shown on Figure 5.14 gives a high-level view of the global interaction between the client, the server, and the registry. A more detailed view is presented in the following sections.

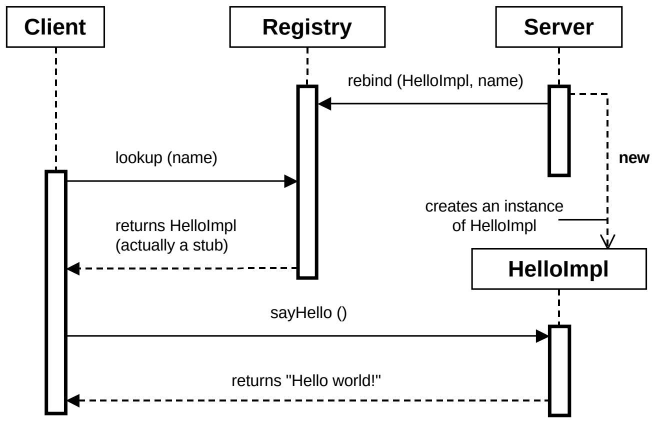


Figure 5.14. The "Hello World" application: overview

The actual execution involves the following steps.

1. Generate the stub and skeleton classes, by compiling the `Hello` interface definition on the server site, using the stub compiler provided by Jeremie.
2. Start the registry (by default the registry is located on the server site, but the system may be configured to make the registry reside on a different site).

3. Start the server.
4. Start the client.

5.4.3 The Inner Working of Jeremie

The execution of the above application proceeds as follows: on the server side, export the servant object and generate the actual stub and skeleton objects (the stub compiler has only generated the corresponding classes); on the client side, set up the binding, i.e. the actual connection between client and server, and perform the call. We examine the detail of these operations in turn.

Exporting the Servant Object

Recall that the servant object's class `HelloImpl` inherits from the standard class `UnicastRemoteObject`. The goal of this extension is essentially to enhance the creation of a remote object: when `new` is called on such a class, a skeleton and a stub are created in addition to the actual instance (using the classes generated by `rmic`), and the stub is returned as the result of `new`, to serve as a reference for the object. Thus the instruction

```
Naming.rebind("jrmi://" + registryHost + "/helloobj", new HelloImpl());
```

in the server program creates a new instance of `HelloImpl` (the servant), together with a skeleton and a stub for this servant, and returns the stub. It then registers the stub in the naming registry under a symbolic name.

Technically, the creation phase is performed by the `UnicastRemoteObject` (Figure 5.15), which exports the new servant object `impl` by calling a servant manager through an `exportObject` method. This manager (called `AdapterContext` on the figure) is essentially a driver for an object adapter; it first gets an instance of a stub factory from the ORB, and uses it to create a skeleton `skel`. It then registers `skel` into an adapter (essentially a table of objects), in which the skeleton is associated with a `key`. The `key` is then exported to the ORB, which delegates its operations to an `IIOPBinder`. This binder manages the IIOP protocol stack; it encapsulates the `key` into an object `ref`, of type `SrvIdentifier`, which includes a [`host`, `port`] pair (the port number was passed as a parameter to `exportObject`; if missing, an available port is selected by the binder). Thus `ref` is actually a reference for the servant `impl`. This reference is then used to create a stub (again using the stub factory), which is finally returned.

The stub is then registered, which concludes the export phase. Note that the stub must be serializable in order to be transmitted over the network.

This framework is extensible, i.e. it identifies generic interfaces under which various implementations may be plugged in, such as the adapter interface and the binding protocol interface, which are shown on the figure.

Setting up the Binding

The client looks up the servant in the registry, using its symbolic name:

```
   Hello obj = (Hello) Naming.lookup("jrmi://" + registryHost + "/helloobj");
```

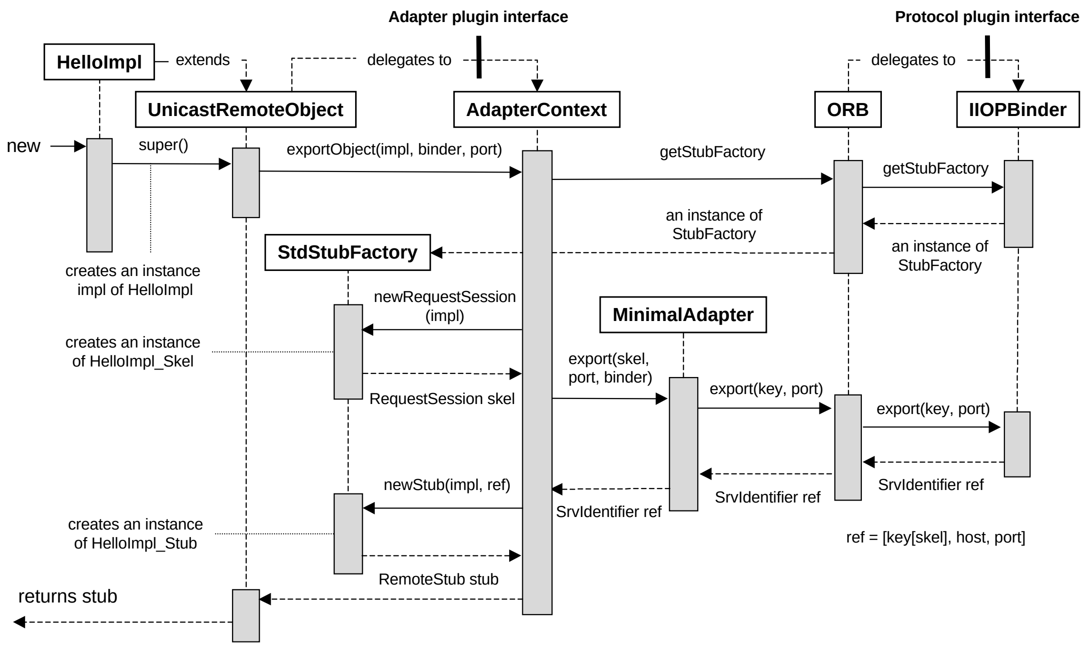


Figure 5.15. Remote Method Invocation: creating the stub and skeleton

This operation apparently retrieves a "servant", but what it actually does is setting up the binding, by first retrieving a stub and then binding this stub to the skeleton, and therefore to the actual servant. We now present the details of this operation (Figure 5.16).

The stub object stored in the registry is returned as a result of the `lookup` operation. This stub is still unbound, i.e. the calling path to the remote servant is not set up. The binding operation relies on the mechanisms of object transmission in Java:

- when an externalizable object is read from an `ObjectInputStream` by the `readObject` method, this method is superseded by a specific `ReadExternal` method;
- after an object has been de-serialized, a `readResolve` method is called on this object.

The first mechanism is used when the stub delegate (here called `RefImpl`) is read. Recall that the delegate contains a reference to the remote servant, including its IP address, port number and key in the adapter. The call to `ReadExternal` creates a client endpoint (called `CltId` on the figure), managed by the `IIOPBinder`. The second mechanism invokes the `bind()` method on this endpoint, which in turn sets up the binding, i.e. the path to the remote object, using the `IIOPBinder` (to set up a TCP/IP session), and a stub factory (to put the final stub together).

Performing the Call

Finally, the call is performed by the operation

```
System.out.println(obj.sayHello());
```

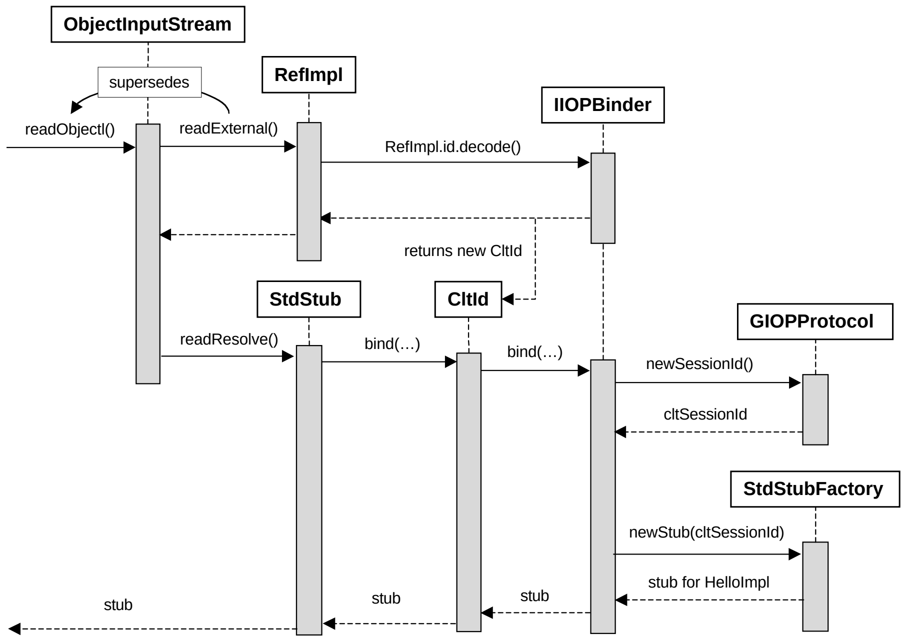


Figure 5.16. Remote Method Invocation: retrieving the stub on the client side

which actually invokes the `sayHello()` method on the stub. The skeleton is located using the reference contained in the delegate, the remote call is performed using the IIOP protocol, and the skeleton dispatches the invocation to the `sayHello` method of the servant. The return value is transmitted using the reverse path.

The Mechanics of Registry Invocation

Recall that the registry may be located on any node and is accessible through a local `Naming` interface. In order to reach the actual registry, `Naming` calls a `LocateRegistry`, giving the symbolic name under which the registry servant has been initialized (e.g. `RegistryImpl`).

`LocateRegistry` uses a binder (here `JIOP`) to retrieve a stub for `RegistryImpl` (Figure 5.17). The binding process is identical to that described above (i.e. using `IIOPBinder`), and details are not shown.

When the registry server is initialized on a host, it creates a `RegistryImpl` servant on a specified port. A precompiled stub class for this servant (`RegistryImpl_Stub`, needs to be present on each site using the registry.

5.5 Case Study 2: David, an Implementation of CORBA

We present the David implementation of CORBA in the Jonathan framework. After a brief introduction (5.5.1), we illustrate the principle of application development in CORBA,

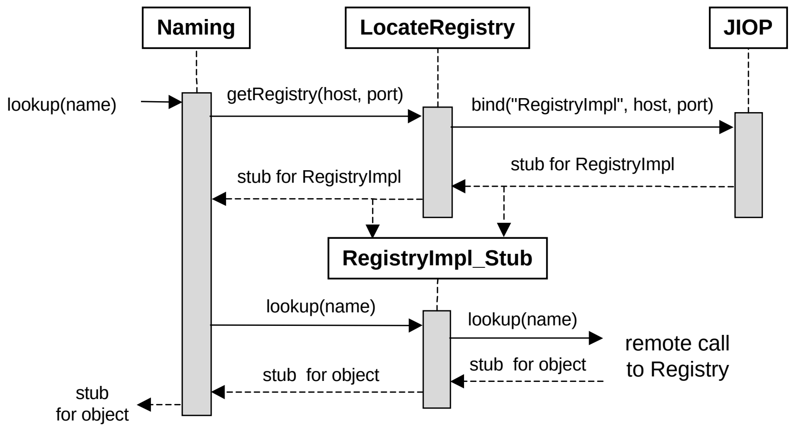


Figure 5.17. Invoking a registry method

through a simple example (5.5.2). We finally describe the inner working of the implementation (5.5.3).

5.5.1 Introducing CORBA

The Object Management Group (OMG) is a consortium created in 1989 with the goal of making and promoting standards in the area of distributed applications development. CORBA (Common Object Request Broker Architecture) is one of these standards, covering the area of middleware and services for applications based on distributed objects.

CORBA is intended to allow the cooperation of heterogeneous applications, using different languages and operating environments. To that end, the standard defines a common way of organizing applications based on the remote objects model, a common Interface Definition Language (IDL), and the main components of an object request broker architecture. In addition, a number of common services are specified for such functions as naming, trading, transactions, persistence, security, etc. These services are accessible through IDL interfaces.

The global organization of an ORB, as defined by CORBA, is represented on Figure 5.18.

The usual invocation path from a client application to a method provided by a remote servant is through a stub and a skeleton generated from an interface description. CORBA defines a generic IDL, from which stubs and skeletons may be created for different languages, using the appropriate tools. The mapping of the entities defined in the IDL to constructs in a particular language is defined by a set of binding rules. Such bindings have been defined for all usual programming languages. Thus for instance a client written in C++ may call a servant written in Java: the client stub is compiled using tools based on the IDL to C++ binding, while the skeleton is compiled using tools based on the IDL to Java binding.

The ORB itself provides interfaces to both clients and servants, in order to fulfill a number of basic functions such as a primitive name service and a tool to map object references to strings and vice-versa, in order to facilitate the interchange of references.

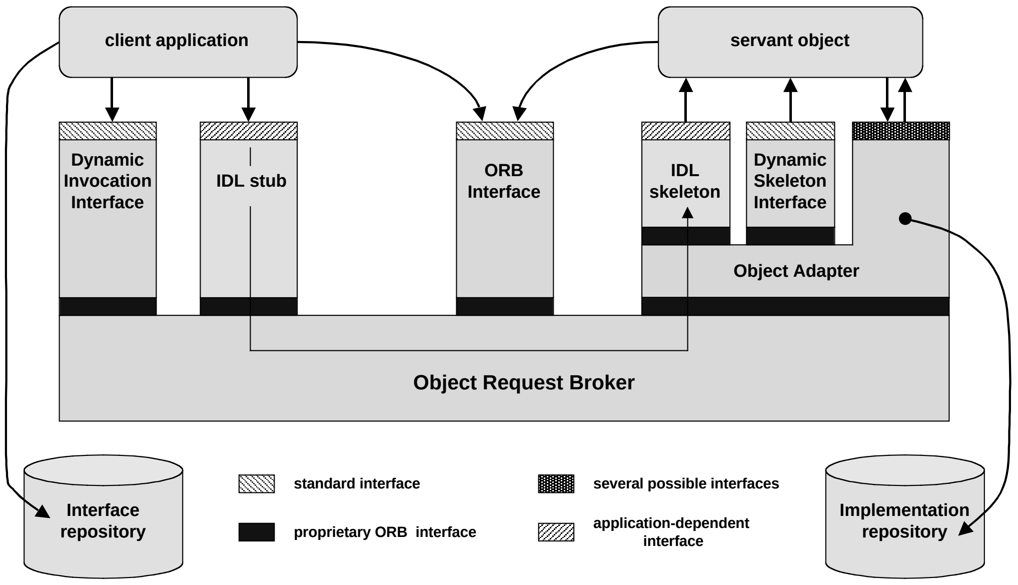

client application
servant object
Dynamic Invocation Interface
IDL stub
ORB Interface
IDL skeleton
Dynamic Skeleton Interface
Object Adapter
Object Request Broker
Interface repository
Implementation repository
standard interface
several possible interfaces
proprietary ORB interface
application-dependent interface

Figure 5.18. The global organization of CORBA

CORBA uses object adapters on the server side. The motivation for an adapter has been given in 5.2.3. The adapter manages servant objects, using an implementation repository. The use of adapters in CORBA is further developed in 5.6.

Finally, CORBA provides facilities for dynamically constructing the elements of an invocation path, by allowing a client application to discover a servant interface at run time (using an interface repository), and to create requests without using stub generation tools. These aspects are examined in 5.6.

5.5.2 Developing a CORBA Application

We use the same application (HelloWorld) as in Java RMI. The centralized version has been described in Section 5.4.2.

The IDL description of the interface of the remote object is:

```
interface Hello {
   string sayHello();
};
```

Since we intend to write the client and server programs in Java, this description must be compiled using an IDL to Java compiler, to generate the stub and skeleton files (`_HelloStub.java` and `_HelloImplBase.java`, respectively). The compiler also generates auxiliary programs (holders and helpers), whose function will be explained when necessary.

Here is the the program of the server.

```
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CosNaming.NameComponent;
import org.objectweb.david.libs.helpers.IORHelpers;
import idl.*;

class HelloImpl extends _HelloImplBase {

   HelloImpl() {}
   public String sayHello() {
      return "Hello World!";
   }
}

public class Server {
   public static void main (String[] args) {
      try {
         ORB orb = ORB.init(args,null);
         Hello hello = new HelloImpl();
         orb.connect(hello);
         IORHelpers.writeIORToFile(orb.object_to_string(hello),"hello_ior");
         org.omg.CORBA.Object ns_ref =
            orb.resolve_initial_references("NameService");
         NamingContext ns = NamingContextHelper.narrow(ns_ref);
         ns.rebind(new NameComponent[] { new NameComponent("helloobj","") },hello);
         System.out.println("Hello Server ready");
         orb.run();
      } catch (Exception e) {
         System.err.println("Hello Server exception");
         e.printStackTrace();
      }
   }
}
```

As may be expected, the program of the servant object is identical to that used in Java RMI. The difference lies in the server program.

The first instruction `ORB orb = ORB.init(args,null)` initializes the ORB, which is thereafter accessible as an object `orb`. Then an instance `hello` of the servant is created. The instruction `orb.connect(hello)` makes the servant known to the ORB for further use, by exporting it to an adapter.

The following instructions register the servant (actually a reference to it) in the CORBA name service. A reference to this service must first be retrieved, which is done using a primitive name service (`resolve_initial_references`) provided by the ORB. Note the use of the `narrow` operation performed by a "helper" program: the primitive name service returns references to type `Object`, and these references need to be cast to the actual type of the object. However, a language cast operation cannot be used here, because the references are not language references, but remote references managed by the ORB.

Finally, the `orb.run()` operations puts the server thread to sleep, waiting for client invocations.

Here is the client program.

```
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CosNaming.NameComponent;
import idl.*;

public class Client {
   public static void main(String[] args) {
      try {
         ORB orb = ORB.init(args,null);
         org.omg.CORBA.Object ns_ref =
            orb.resolve_initial_references("NameService");
         NamingContext ns = NamingContextHelper.narrow(ns_ref);
         org.omg.CORBA.Object obj_ref =
            ns.resolve(new NameComponent[] { new NameComponent("helloobj","") });
         Hello obj = HelloHelper.narrow(obj_ref);
         System.out.println(obj.sayHello());
      } catch (Exception e) {
         System.err.println("Hello Client exception");
         e.printStackTrace();
      }
   }
}
```

The ORB is initialized like in the server program. Then a reference for the servant is retrieved using the naming service. This reference must again be cast to its actual type, using the appropriate helper. Finally the remote operation is invoked.

CORBA uses a standard object reference format (Interoperable Object Reference, or IOR), defined by the IIOP specification. While these references are opaque (i.e. their contents is hidden), they may be used through primitives provided by the ORB. Thus, in the server program, the registration of the servant in the name service might be replaced by:

```
IORHelpers.writeIORToFile(orb.object_to_string(hello),"hello_ior_srv");
```

This instruction writes in the `hello_ior_srv` file a “stringified” form of the IOR. This string may be sent to the client (e.g. by mail), and copied in a file `hello_ior_clt`. The reference to the servant may then be retrieved in the client program by:

```
hello = orb.string_to_object(IORHelpers.readIORFromFile("hello_ior_clt"));
```

Executing the application involves exactly the same steps as in Java RMI: generating the stub and skeleton classes by compiling the IDL interface description; starting the name service; starting the server; starting the client.

CORBA allows additional facilities: dynamic request generation, adapter programming. These aspects are examined in Section 5.6.

5.5.3 The Inner Working of David

The compilation of the IDL interface description generates several files for each servant. For instance, from the `Hello` interface description, the `Idl2Java` compiler creates the Java source files for the following entities: `HelloHelper`, a class which carries auxiliary operations such as `narrow` (the reference casting operation); `HelloHolder`, a class which carries read and write operations of objects of type `Hello` on streams; `HelloOperations`, the interface of objects of type `Hello`; `_HelloStub`, the stub class; `_HelloImplBase`, the base class from which the servant object derives (this class extends a predefined `Skeleton` class).

The overall working of the CORBA ORB is similar to that described for Java RMI. The main differences are the use of the adapter on the server side, and the reflective operations described in 5.6.

The first step is to obtain a reference on the ORB as an object. This is done by the operation `ORB orb = ORB.init()`, which creates an ORB as an instance of a singleton class (a class with a single instance). This ORB provides a number of primitive operations, some of which are described in the rest of this section.

Exporting the Servant Object

The servant object's class `HelloImpl` inherits from the `_HelloImplBase` generated class, which itself inherits from `Skeleton`. This extends the servant's class constructor: when a new servant is created (an instance `hello` of `HelloImpl`), a new server delegate (5.3.1) is created and associated with that servant.

The operation `orb.connect(hello)` exports the new servant to the ORB. The server delegate is registered in the adapter (which returns a key to subsequently retrieve the servant), and then exported to the IIOP binder (which provides host address and port). At this stage, the IOR (containing host, port, and key) is ready, and it is actually registered in the servant itself (here, as an instance of `SrvIdentifier`). This is shown on Figure 5.19.

The next step consists in registering the servant in the name server. The name server must first be located, which is done as follows through a primitive naming service provided by the ORB:

```
org.omg.CORBA.Object ns_ref =
    orb.resolve_initial_references("NameService");
NamingContext ns = NamingContextHelper.narrow(ns_ref);
```

Retrieving a reference for the name server uses an internal association table (a context) managed by the ORB, in which a reference for the server has been initially registered. From this reference, a stub for the name server is generated, using the binding mechanism explained in the next subsection. The function of `narrow` (reference cast) has been explained in 5.5.2.

The servant is then registered in the name server:

```
ns.rebind(new NameComponent[] { new NameComponent("helloobj","") },hello);
```

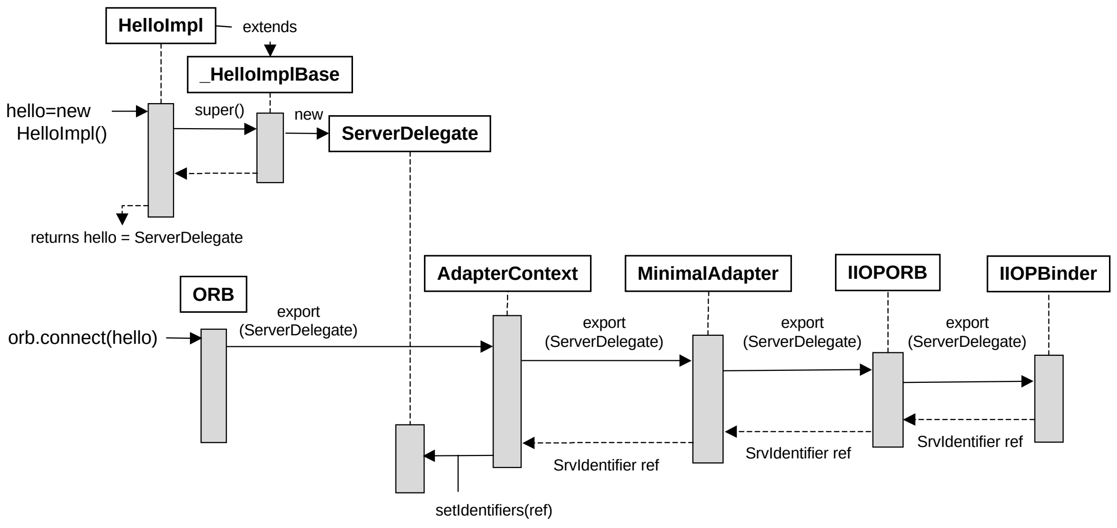


Figure 5.19. Exporting a servant in CORBA, phase 1

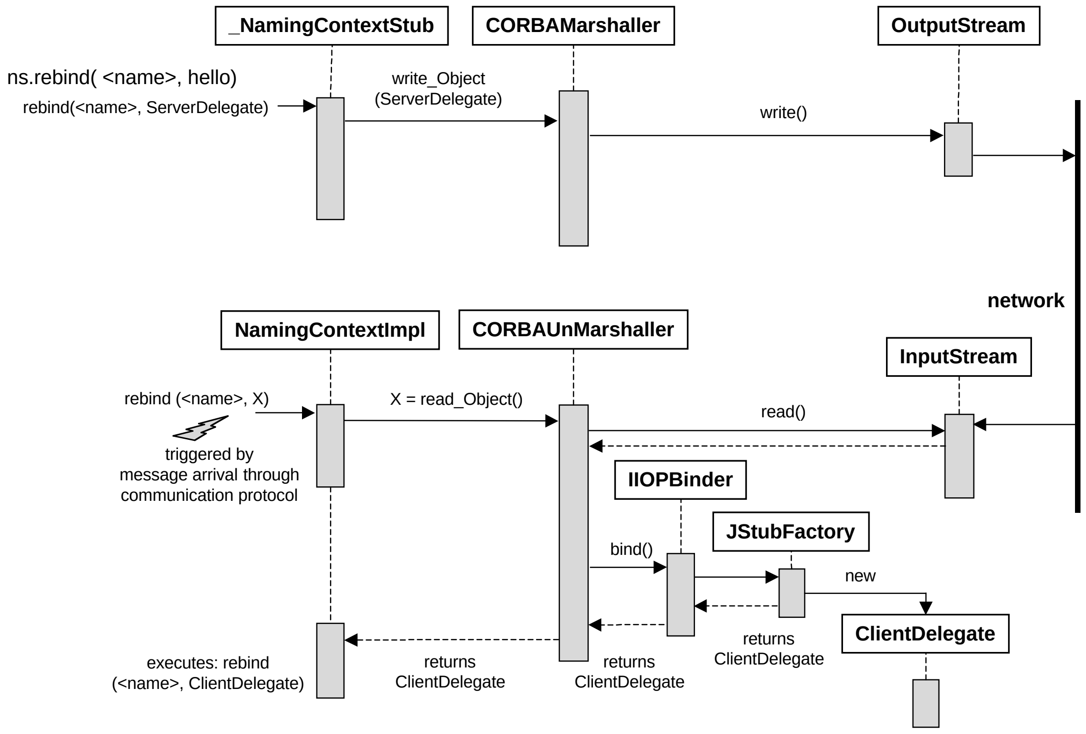


Figure 5.20. Exporting a servant in CORBA, phase 2

The effect of this operation is the following (Figure 5.20).

The reference of the servant (i.e. the `SrvIdentifier`) is sent to the name server, using the encapsulation mechanism described in 3.4.3. In this process, the identifier is unmarshalled from an `ObjectInputStream`, using the `ReadObject` operation. This operation invokes `bind` on the decoded identifier, in the `IIOPBinder` context, which in turn invokes a `JStubFactory`. This latter constructs a `ClientDelegate` (5.3.1), an access point to the servant using a generic (invoke) interface. This delegate is registered in the name server, to be later retrieved by the client, as described below.

Completing the Binding

After initializing the ORB and locating the name server (like above), the client binds to the servant by calling the name server:

```
ns.resolve(new NameComponent[] { new NameComponent("helloobj","") });
Hello obj = HelloHelper.narrow(obj_ref);
```

Like in the registration phase, the `ClientDelegate` is transmitted in an encapsulated form and unmarshalled through `ReadObject`. The `IIOPBinder` now constructs a stub using this delegate, using the mechanism illustrated in the lower-right quarter of Figure 5.20. The binding is now complete, including the communication path (session) between the client and the server.

Performing the Call

The call may now proceed using the object invocation scheme described in 5.3.1 and illustrated in Figures 5.9 and 5.10. The mechanics of passing objects as parameters or results by reference is again illustrated by the operation of the `Naming.rebind` primitive (Figure 5.20, i.e. recreating a delegate on the receiving site as the core of the object's stub.

5.6 Complements and Extensions

Currently not available (should cover reflective features, semi-synchronous invocations, etc.)

5.7 Historical Note

The historical evolution of object middleware has been outlined in the Historical Note section of the Introduction Chapter (1.5).

Chapter 6

Coordination and Events

Many applications involve a number of activities that cooperate to achieve a common goal and need to interact in order to coordinate their evolution. In this chapter, we are interested in *asynchronous* interaction based on three related mechanisms: events, i.e. state transitions that trigger reactions, message queuing, and shared data-spaces. We present the main paradigms of asynchronous coordination of loosely coupled activities and their implementation in middleware. These notions are illustrated by a few case studies.

6.1 Introducing Coordination

6.1.1 Motivation and Requirements

In a broad sense, *coordination* refers to the methods and tools that allow several entities to cooperate towards a common goal. A coordination model provides a framework to organize this cooperation, by defining three elements [Ciancarini 1996]: a) the *coordination entities* whose cooperation is being organized, e.g. processes, threads, various forms of "agents" (see 6.6), organizations, people, etc.; b) the *coordination media* through which the entities communicate, e.g. messages, shared variables, or more elaborate mechanisms built above the basic communication layers; and c) the *coordination rules*, which define the interaction primitives and patterns used by the cooperating entities to achieve coordination.

Besides the notion of cooperation, coordination also has a function of integrating previously independent activities. In the words of [Gelernter and Carriero 1992]: "A coordination model is the glue that binds separate activities into an ensemble". In that sense, coordination is related to composition. The main difference is that coordination models are associated with loose coupling, while most component-based models (Chapter 7) imply a notion of strong coupling.

We are specifically interested in application domains that are subject to the following requirements.

- *Loose coupling.* The entities involved in the cooperation should be allowed to evolve independently, and coordination should therefore not impose strong constraints on their behavior.

- *Dynamic evolution.* The cooperating entities are allowed to join and leave the application freely, during execution. Late binding is therefore mandatory.
- *Large scale.* The number of cooperating entities may potentially become very large and they may operate over a wide geographical range; the coordination algorithms and structures should therefore be scalable with respect to these two aspects.
- *Heterogeneity.* The cooperating entities may use heterogeneous hardware, operating systems, run time environments, and security policies; they may be administered by various authorities.

Examples of such application domains include monitoring of industrial installations, supervision of networking equipment, weather forecast using distributed sensors, stock tracking, distributed auctions. In these applications, systems must react to variations in the environment, and notifications of changes must be propagated to a dynamically changing set of recipients, which may themselves react by generating other notifications. Devices and services may be dynamically added or removed. These environments usually do not have a central controlling authority. Another relevant area is that of ubiquitous and mobile systems, which involve spontaneous, usually unpredictable, patterns of communication between heterogeneous objects such as mobile phones, portable or wearable devices, sensors and actuators embedded in mobile equipment, etc. An asynchronous, loosely coupled communication model is well adapted to these requirements.

6.1.2 Terminology

Before introducing the plan of the chapter, a word on terminology is in order, because the terms that designate the main concepts of coordination are heavily overloaded and therefore subject to misuse.

An *event* is a detectable state transition that locally occurs at a definite point in time in a specified environment. Example of events, in the environment of a process, are: the change of the value of a variable (the contents of a memory location); the occurrence of an interrupt, e.g. a timeout expiration signal or a mouse button click.

In a more general sense, in the context of a distributed system, an event is the notification of a local state transition (an event in the above sense) to a set of (usually remote) receiving entities. The entity in which the state transition occurred is an event producer, or *event source*; the entities that receive the notification are event consumers, or *event sinks*. An event notification may be initiated by the source (*push* mode) or by the sink (*pull* mode); many coordination schemes use a combination of these two modes.

Note that the reception of a notification by a consumer is a (local) event, in the original sense, for that consumer; and that event may in turn be notified to other consumers. Also note that the term "notification" designates both the action of signaling an event to consumers and the data structure used for this signaling.

The reception of an event notification by an event sink causes the execution of a specified *reaction*. The form of this reaction, and the association between an event and a reaction, depend on the specific coordination model and on the framework that implements it. The terms *callback* and *handler* are often used in relation to the implementation of reactions to events.

A *message* (as defined in Chapter 4) is an information transmitted by a sender to one or several receiver(s). A message may be considered at different levels of abstraction, from the physical signal to more elaborate formats. In the context of coordination systems, a message usually has a composite structure that includes a contents proper and meta-information whose format depends on the specific messaging system.

From the above definition, events (more precisely event notifications, which propagate the occurrence of a local event) are specific instances of messages. The difference between events and messages is one of usage rather than an intrinsic one; and a number of basic patterns (e.g. the selection of receivers) are common to both mechanisms. One usually refers to "events" when emphasis is on the occurrence and signaling of the initial transition (the local event), and to "messages" when emphasis is on the information contents being transmitted. However, in many cases, the two terms are not clearly distinguished. Products that provides coordination primitives using either form of communication are collectively called *Message-Oriented Middleware* (MOM).

In the context of coordination systems, the term *agent* refers to an entity that coordinates its activity with that of other similar entities, and which has the following characteristics: it is active, i.e. it encapsulates at least one thread of execution; it is self-contained, i.e. it includes a minimal set of resources needed for its execution; it may communicate both with its environment (i.e. the run time system that supports its execution) and with other agents; its behavior is determined both by its own initial program and by its interactions; it may be fixed or mobile (in the latter case, it may migrate from one environment to another one). The term "agent" is mainly used in conjunction with a specific support system, which defines its interaction primitives and patterns.

In the rest of this chapter, we present the principles of these coordination mechanisms and the middleware that implements them. The common patterns for coordination are reviewed in Section 6.2. Event-based communication is the subject of Section 6.3. Message-based communication and middleware are presented in Section 6.4. Coordination through shared objects is discussed in Section 6.5. Some instances of agents are described in Section 6.6. Finally, Section 6.8 is a case study of a message-oriented middleware.

6.2 Patterns for Coordination

The three patterns that follow are directly relevant to coordination. The main common objective is to favor uncoupling between the cooperating entities, which leads to various forms of indirect, asynchronous interaction. Since these patterns apply to several coordination models, and since they may be implemented using various techniques, we present them in an abstract form, concentrating on the communication aspects. Some indications on the implementation techniques and on the non-functional properties are given in the case studies.

6.2.1 Observer

Observer is the basic pattern that underlies coordination. In its primitive form [Gamma et al. 1994], it involves one "observed" object and an unspecified number of independent "observer" objects. The problem is for the observers to be kept aware of any change occurring in the observed object.

This situation typically occurs when a given abstract entity (e.g. the contents of a spreadsheet) has several different representations, or images (e.g. a table, a histogram, a pie chart, etc.). Whenever that entity is modified, all its images need to be updated to reflect these changes; these images, and the processes that maintain them, are independent of each other, and new forms of images may be introduced at any time.

In the proposed solution, the observers register their interest with the observed, which therefore ke a list of these observers. When a change occurs, the observed asynchronously notifies all the registered observers. Each observer can then query the observed to get aware of the changes. This pattern therefore uses both the push and the pull modes.

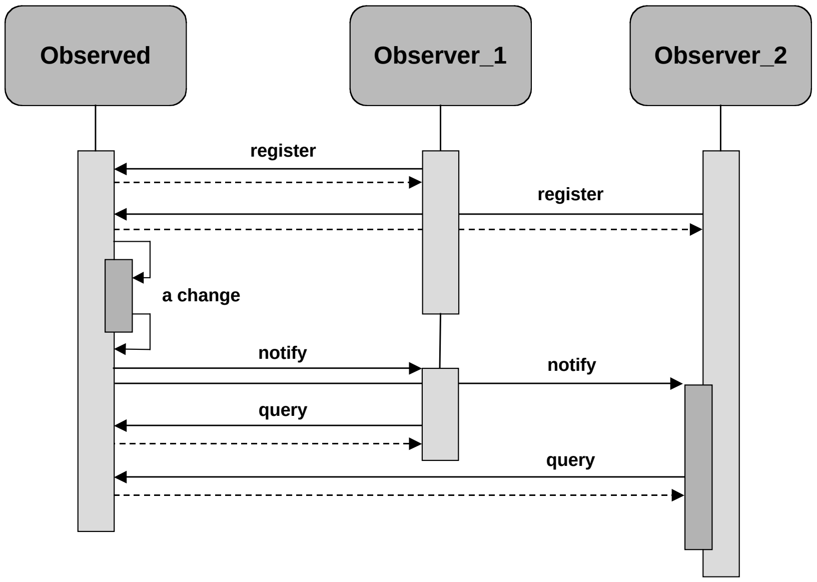


Figure 6.1. The OBSERVER pattern

This solution favors uncoupling, not only between the observers, which are not aware of each other, but also between the observed and the observers. Once an observer has been informed of a change, it may query the observed for a specific piece of information and it may choose the moment of the query.

This pattern has the following limitations.

- The observed entity bears a heavy load, since it has to provide and to implement the registration interface, to keep track of the observers, and to answer the queries. This is detrimental to performance and to scalability.
- While favoring uncoupling, the two-stage process of notification and query is not selective and may lead to unnecessary exchanges (e.g. an observer may not be interested in a specific change, but has to query the observed in all cases).

The following pattern attempts to remedy these drawbacks.

6.2.2 Publish-Subscribe

The PUBLISH-SUBSCRIBE pattern is a more general form of OBSERVER. It defines two "roles", Publisher (event source) and Subscriber (event sink). A given entity may take up either role, or both.

Events generated by publishers are selectively notified to subscribers. The selection is achieved by defining event classes, i.e. sets of events that satisfy some conditions, to be discussed later. A subscriber may register its interest for a class of events by *subscribing* to that class. When a publisher generates an event E, all subscribers that previously subscribed to an event class to which E belongs receive an asynchronous notification of the occurrence of E. Reacting to this notification is locally arranged by each subscriber (e.g. by associating a specific handler with each class of events, etc.).

We present the pattern in an abstract form by defining a *mediator* as an entity that is responsible for registering subscriptions, receiving events, filtering events according to class definitions, and routing them to the interested subscribers (Figure 6.2). Depending on the specific system, the mediator may be implemented in various ways, e.g. by a centralized server, by distributed cooperating servers, or possibly collectively by the publishers (like in OBSERVER). These organizations are discussed and illustrated by examples in Section 6.3.

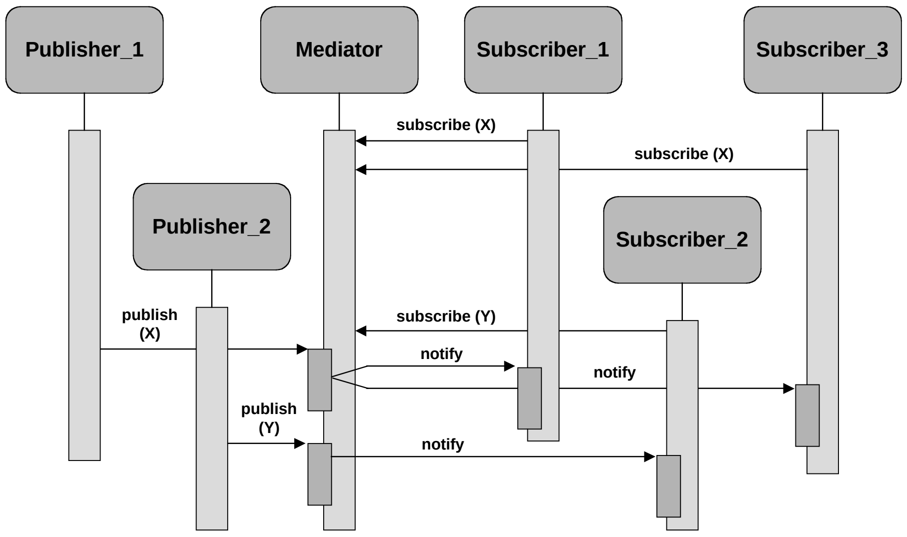


Figure 6.2. The PUBLISH-SUBSCRIBE pattern

There are two main ways of defining event classes.

1. *Topic-based classification.* This scheme relies on the association of events with named topics, or subjects. Topics essentially define a name space (3.1), which may be organized according to the needs of a specific application or application domain. The topics need not be disjoint, i.e. an event may be associated with several topics. Like for any name space, a number of topic organizations may be defined, e.g. flat space, hierarchy, graph, etc., and all the facilities associated with name manipulation

(pattern matching using wildcards, etc.) may be used. The structure of the topic space may conveniently mimic that of other currently used name spaces, e.g. file systems or URLs.

A variation of the topic-based classification is *type-based classification* [Eugster et al. 2000]. In this scheme, events are filtered according to their type, which is a specific attribute. This implies that the publish-subscribe system should be integrated in a typed programming language. Event notifications are then defined as typed objects, with the benefits of typing (safety, compile time checking). Note however that some languages allow dynamic type changes, which offsets these benefits to favor flexibility.

2. *Content-based classification.* This classification scheme introduces an additional degree of flexibility, since events are classified according to their run-time properties, and not to a statically defined topic organization. The "contents" of an event is the data structure used for notification, or meta-data associated with this structure. Event filtering is now an associative process, i.e. matching a specified template against the contents. Depending on the organization of an event's contents, the template may take various forms, e.g. a string, one or several name-value pair(s), etc.

Content-based classification is more general than topic-based classification, since the latter may be implemented by means of the former while the reverse is not true. However, implementing a scalable content-based filtering is a difficult task (see example in 6.3).

The Publish-Subscribe pattern has the potential of achieving the uncoupling of the cooperating activities:

- due to the indirect interaction scheme, the activities have no knowledge of each other; they do not directly interact for synchronization, and do not need to be active at the same time; they may be dynamically created and removed, and may freely enter or leave the application.

- due to the event filtering mechanism, the number of notifications is reduced: an activity is only notified of the events that it considers relevant; content-based filtering allows this selection to be dynamic.

However, actually achieving these potential benefits is strongly dependent on implementation. For instance a centralized implementation may reduce the mutual independence of activities; or an inefficient implementation of a filtering algorithm may offset the advantage of selectivity. Implementation issues are examined in Section 6.3.

A survey of various aspects of Publish-Subscribe is given in [Eugster et al. 2003].

6.2.3 Shared Dataspace

The Shared Dataspace pattern has been introduced in the Linda language [Carriero and Gelernter 1989, Gelernter and Carriero 1992], under the name of *generative coordination.* It defines a form of communication between uncoupled processes, through a common, persistent information space organized as a set of *tuples.* In the original model, a tuple is a record, i.e. an ordered set of typed fields. A process may insert a new tuple into the space, lookup the space for a tuple matching a specified template (e.g. by specifying the values of some fields) and then read the tuple and possibly remove it from the space. Tuples are persistent, i.e. a tuple remains in the space until explicitly removed. This is a pure pull model in the sense that changes in the tuple space are not explicitly notified to processes. A process whose query fails remains blocked until a tuple matching the query is inserted into the the space.

In the original model, processes are themselves elements of the tuple space. When a process is created, it is inserted in the tuple space; after the process finishes execution, it turns into a passive tuple, which is a representation of its state at the end of the execution.

This pattern has several limitations, which we present together with some proposed remedies.

- There is a single dataspace. Multiple dataspaces are useful for improving modularity, for exploiting locality, and for implementing isolation and protection. Various forms of multiple dataspace organizations (independent, nested) have actually been introduced in several implementations.
- When several tuples match its template, a query returns one of these tuples, and the choice is nondeterministic. When several processes query the space with the same template, there is no way for either of them to find all the tuples that match the template.

  This problem is solved by introducing an additional primitive, *copy-collect*, which allows a process to get *all* the tuples that match a given template.
- The original model is based on a pure pull mode and therefore lacks reactivity. Several implementations have added a *notify* primitive, which signals the introduction of a tuple matching a template to interested processes.

The shared dataspace pattern is useful in applications in which the resolution of a problem is partitioned among a number of "worker" processes, which use the tuple space to allocate work and to share results. Another application area is resource discovery (3.2.3), a form of trading in which resources are looked up according to various criteria, and may be dynamically added and removed.

6.2.4 Extra-Functional Properties

Recall (2.1.3) that extra-functional properties are those which do not explicitly appear in the interface of a service. Several such properties, listed below, are useful additions to a coordination system.

- Persistence. An event or a message may be transient, i.e. it is lost if no receiver is ready to accept it, or it may be persistent, i.e. it is conserved by the system until explicitly removed. Persistence is implemented in most coordination systems because it favors uncoupling, since there is no synchronization constraint between a sender and a receiver.

- Ordering. Message ordering is discussed in **??**. Imposing strong ordering constraints goes against uncoupling. However, such guarantees as causal ordering are unavoidable whenever messages and events are used to maintain such constraints as consistency of replicated information. Designing scalable ordering algorithms is a challenging task because it involves global synchronization, and because large scale communication systems such as the Internet do not provide any ordering guarantees. One approach consists in partitioning the set of coordinating entities and trying to derive global ordering properties from local ones by an adequate choice of the partition (see e.g. [Laumay et al. 2001]).
- Transactions. Executing reactions to events as transactions may be required to preserve application-specific invariants, or to prevent an observer from seeing an inconsistent state of the system.

6.2.5 Comparing the Coordination Patterns

The Publish-Subscribe and Shared Dataspace patterns illustrate two dual approaches to coordination of loosely coupled entities. The first one is based on messages, the second one on shared information. It is interesting to compare their expressive power. The main result in this area has been proved in [Busi and Zavattaro 2001], and may be summarized as follows.

- The Shared Dataspace model may be reduced to (i.e. simulated by) the Publish-Subscribe model.
- The Publish-Subscribe model may not be reduced to the original Shared Dataspace model. However, the reduction becomes possible if the dataspace model is extended with a primitive such as *copy-collect* (6.2.3).

Therefore the choice between the two forms of coordination is not mainly based on expressive power, but on the adequation of the pattern to the main orientation of the problem (event-driven vs. data-driven), and on the availability of efficient and scalable implementations.

6.3 Event-based Communication

In this section, we examine the main implementation aspects of event-based communication based on the Publish-Subscribe pattern. Efficient implementation is critical for achieving the benefits of uncoupling. The most important issue is *scalability*, with respect to both the number of cooperating entities and geographical size.

We first examine three main aspects: organizing the mediator, implementing filtering, implementing notification routing. Since the mediator implements the filtering and routing algorithms, the issue of mediator organization is closely related to the design of these algorithms. For presentation, we find it more convenient to discuss architectural aspects first. In addition, some properties, such as fault tolerance and locality, directly depend on the architecture and can be examined in that context.

This section concludes with a brief review of a few case studies.

6.3.1 Event Mediator Architecture

We assume that the event system actually uses a mediator. In other words, we do not consider systems using the OBSERVER pattern, because the requirement of uncoupling precludes direct access from the consumers to the producers[1].

The mediator is the entity that receives events from publishers and delivers notification to subscribers. Implementing the mediator as a single server has the usual drawbacks of centralized architectures: the server is a single point of failure and a performance bottleneck. This solution is limited to small scale system on local area networks.

Event mediators for large scale systems are therefore organized as multiple cooperating servers. There are several ways of designing the server architecture.

1. The first solution is a hierarchical organization, in which the servers are organized into a tree. Coordinating entities (publishers and subscribers) may be connected to any server in the hierarchy. If a server does not have the information to process a request (the distribution of information among the servers depends on the algorithms), it sends it to its parent server.

   While the communication pattern is straightforward, this solution has two main drawbacks: it puts a heavy load on the higher levels of the hierarchy, and it is sensitive to server failures, since a failed server isolates all its descendants. The usual remedy for these drawbacks, server replication, is not easily applicable because the information update rate is typically high, and therefore replica consistency is expensive to maintain.

2. The second solution is an acyclic peer to peer organization, in which the communication pattern between servers is an acyclic undirected graph. The main benefit over the hierarchical organization is a better distribution of the load. The absence of cycles in the communication pattern can again be exploited by the filtering and notification algorithms (e.g. algorithms based on a spanning tree are easily implemented).

3. The third, more general solution is a general peer to peer organization, using an unrestricted graph communication pattern. This solution has no limitations and may be used to improve fault tolerance by redundancy. However, the communication graph has no special properties.

These organizations are discussed in more detail in [Carzaniga et al. 2001].

As usual, hybrid solutions combining the advantages of the above "pure" communication schemes are found in practice. An interesting solution is a clustered servers organization in which inter-cluster communication follows an acyclic pattern, while intra-cluster communication is unrestricted. This organization ke the advantage of acyclicity for inter-cluster (global) communication and has the following additional benefits.

- Exploiting locality. In many applications, the coordinated entities are grouped according to organizational or geographic criteria, and the communications within a

[1]An example of an event model using direct access is the Java Distributed Event model used in JavaBeans, a system essentially used in a centralized environment.

group are more frequent than those between groups. Server clusters can be organized so as to reflect this structure, making intra-group communication more efficient.

- Improving fault tolerance. Server availability is increased by using some nodes of a cluster as replicas. Replica consistency must only be maintained within a cluster.
- Allowing efficient causal delivery. Causal delivery of notifications is discussed in 6.2.4. As shown in [Laumay et al. 2001], partitioning the servers in acyclically connected clusters allows a scalable implementation of causal delivery.

6.3.2 Event Filtering

Event filtering (or matching) is the process of finding the subscriptions that match a published event, so as to select the recipients of notification. The problem is different for the two forms of subscription described in 6.2.2.

- For topic-based subscription, the topics are statically known, before the event is published. The problem reduces to looking up an element in a (possibly distributed) table. A structured organization of the topics (e.g. hierarchical) allows for faster search. In any case, scalable lookup algorithms based on hash-coding are known.
- For content-based subscription, the situation is different, because the contents associated with an event is only known at publication time. Designing scalable algorithms for this case is still a research issue.

Here we only consider content-based subscription. A naive algorithm would match the contents of a publish message against all registered subscriptions, and would therefore run in linear time with respect to the number of subscriptions. For Internet-scale systems, this is considered inefficient, and the goal is to design sub-linear matching algorithms.

Such an algorithm, used in the Gryphon system [Gryphon 2003], is described in [Aguilera et al. 1999]. It is based on pre-processing of subscriptions. The motivation is that the rate of change of subscriptions is slow with respect to the rate of event publishing, which makes pre-processing feasible. Pre-processing creates a matching tree, in which each node describes a test on some subscription attribute, and the edges outgoing from this node represents the results of this test. The leaves of the tree represent individual subscriptions. Thus an event is matched against successive nodes of the tree, starting from the root, by performing the prescribed tests. Several paths are possible, i.e. an event may match several subscriptions.

The matching algorithm is used on each server, in a multi-server mediator system. The problem of routing events in the network of servers is examined in the next section.

6.3.3 Notification Diffusion

In the trivial case in which there is no filtering (or a subscription that matches all events), the problem of routing reduces to multicasting a notification to all subscribers. This problem is adequately solved, even on a large scale, by protocols such as IP Multicast (see Chapter 4).

The solution is again easy for topic-based subscription. A multicast group is associated with each different topic; new subscribers join the relevant groups, and events published under a topic are notified to the associated group, again using a group multicast protocol.

With content-based subscription, in a publish-subscribe system implemented as a network of servers (6.3.1), the question is how to best exploit the topology of the network. There are two extreme solutions, assuming that each publisher or subscriber sends its request to one server in the network.

- Propagating each subscription to all servers, and performing the matching at the server on which the event is published. This produces a list of matching subscribers, to which a notification is multicast.

- Registering each subscription on the server that receives it, and propagating all published events to all servers. Matching is then performed on the servers on which subscriptions have been received.

The first solution is acceptable in small scale systems (recall that the rate of subscriptions is usually much lower that that of event production). Neither solution scales well, however. Therefore more elaborate solutions have been proposed.

In Gryphon, a server which receives an event uses the matching tree pre-computed from subscriptions (6.3.2), in order to selectively propagate notifications to its neighbors (on a spanning tree covering the network of servers). The notifications are only sent on those links that are part of a path to at least one matching subscriber. The algorithm is described in [Banavar et al. 1999].

Siena uses routing algorithms based on two general requirements, inspired by the design principles of IP Multicast: a) to route notifications in one copy as far as possible (i.e. to start replicating it only as close as possible to its destinations); and b) to apply filtering as close as possible to the event sources. Thus a subscription sets up a routing path to be used later to propagate notifications, in the form of a tree that connects the subscriber to all servers in the network. Notifications that match the subscription are then routed to the subscriber following the routing path in the reverse direction. This may be again optimized if event sources give advance information, in the form of *advertisements*, on the classes of events that they will publish.

6.3.4 Examples

Currently not available.
The examples should include:
Gryphon [Banavar et al. 1999] [Aguilera et al. 1999]
Siena [Carzaniga et al. 2001] [Rosenblum and Wolf 1997]
Infobus [Oki et al. 1993] JEDI [Cugola et al. 2001]

6.4 Message Queuing

6.4.1 Message Queuing Principles

We now consider a coordination mechanism based on message queuing. With respect to the general paradigm of message passing as presented in Chapter 4, the specific feature of this mechanism is *indirect* communication: messages are not directly sent to receivers, but to named queues, from which the receivers can retrieve them. As a consequence:

- There is no need for synchronization between senders and receivers. The receiver of a message need not be active when the message is sent, and the sender need not be active when the message is received.
- Messages are persistent; a message is conserved by the system until explicitly removed from a queue.

These properties favor uncoupling between senders and receivers.

A message queue is a named, persistent container, in which messages are placed and conserved until they are removed. Message queuing achieves a function similar to that of topic-based subscription, i.e. organizing messages according to specified centers of interest.

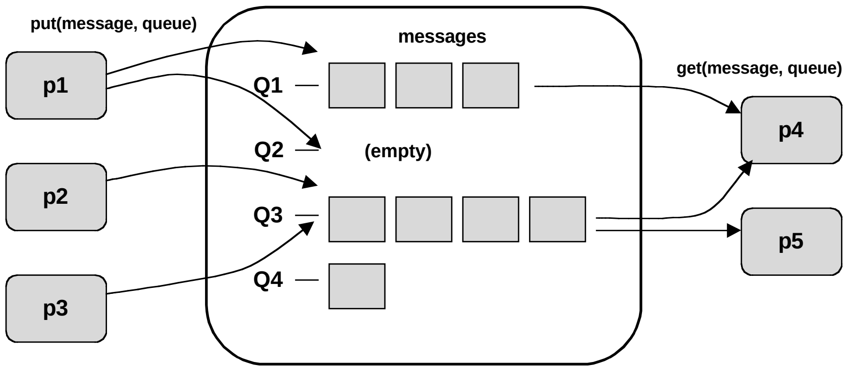


Figure 6.3. Message queuing

A message queuing system provides primitives to put messages in queues and to retrieve messages from queues (Figure 6.3). The latter operation is provided in many variants: it may be blocking or non blocking, and it may remove the message from the queue or leave it there (e.g. to be read by another receiver).

In its pure form, a message queuing system is pull only; it may be augmented with notifications, i.e. a receiver may subscribe to a queue name, and be notified when a new message is put into the designated queue.

6.4.2 Message Queuing Implementation

The main design principle of message queuing implementation is location transparency: senders and receivers see the message queues as logical entities, and the mapping of the queues to physical locations is invisible. This is consistent with the above mentioned analogy between message queues and topic-based publish-subscribe.

A message queuing system may be implemented by a central server that manages the queues, or (more usually) by a network of message brokers.

6.4.3 Examples

Currently not available.

6.5 Coordination through Shared Objects

Currently not available.

6.6 Active Objects and Agents

Currently not available.

6.7 Web Services Coordination

Currently not available.

6.8 Case Study: Joram, a Message-Oriented Middleware

Currently not available.

6.9 Historical Note

Asynchronous events are the earliest coordination mechanism, since the first systems were developed at a low level, directly using the hardware interface, and relied on the hardware interrupt mechanism. In the absence of high-level structures and primitives, these systems were fragile and error prone; due to the transient nature of interrupts, many errors were difficult to reproduce and therefore hard to correct.

As a consequence, the first attempts towards higher level coordination mechanisms aimed at masking the transient and asynchronous nature of interrupts, through such devices as "wake-up waiting switch" and other forms of interrupt memorization. In the mid-1960s, the concepts of sequential process and synchronization were established and led to the invention of semaphores and monitors.

Apart from low-level interrupt handling, which was still used in real-time settings, message passing remained the main asynchronous coordination device in the 1970s. Messages and processes were recognized to have a similar expressive power [Lauer and Needham 1979]. However, most efforts were directed towards the development of tools based on synchronous communication. Attempts towards developing asynchronous communication models, (e.g. [Hewitt 1977] in the context of artificial intelligence), did not meet wide application needs at that time.

In the 1980s, the emergence of new application domains favored the use of asynchronous models based on an event-reaction scheme: graphical user interfaces (GUI), tool integration in software development environments [Reiss 1990], triggers in databases. This period was also marked by a better understanding of the notions of event ordering, causal dependence, and fault tolerance. In the mid 1980's, research on distributed objects introduced the notion of a shared distributed information space. Coordination based on shared dataspaces [Gelernter 1985] was also developed at that time, in a different context.

In the early 1990s, the fast development of local area networks (LANs) stimulated the development of coordination tools based on these networks, such as publish-subscribe systems [Oki et al. 1993] and message queues. In the late 1990s, new application needs such as system administration and networking equipment supervision increased the need for scalable asynchronous coordination. The current challenge is to develop efficient and reliable event distribution systems that can be deployed on the global Internet and that can be augmented with additional properties such as event ordering and transactional processing.

Chapter 7

Composition

Composing an application out of independent, reusable pieces has been a key challenge since the early days of software engineering. This chapter examines some aspects of software composition. While there is still no universally accepted definition of a software component, many projects have explored the requirements for a composition framework, and the industry has developed several technologies for building distributed applications out of components. This chapter starts with a motivation of the need for components, by pointing out the limitations of objects as application building blocks and examining the requirements of component models and infrastructures. It goes on with a discussion of the main paradigms that apply to software composition and to the description of software architecture. It then presents the main patterns used in current middleware for component support, and some examples of their use. The chapter conclude with two case studies of middleware frameworks: Fractal, an innovative proposal founded on a rigorous model; and OSGi, a service-oriented, component-based environment widely adopted by the software industry.

7.1 From Objects to Components

Developing an industry of reusable software components has been a elusive goal, since the early vision formulated in [McIlroy 1968]. The notion of a software application being built as an assemblage of parts seems natural and attractive, but its implementation raises a number of questions. What constitutes a "part"? How is a part specified and implemented? How is it connected with other parts? How is the compound application described? How can it evolve? These questions and related ones are the subject of *software architecture* [Shaw and Garlan 1996], a developing area of software engineering that covers such topics as architecture description languages, configuration management, and dynamic reconfiguration.

There is no single, agreed-upon definition of a software component. This is hardly surprising, for two main reasons: the different kinds of "components" found in the research literature or in industrial products have been designed in response to different sets of requirements; and many aspects of software composition, both fundamental and practical, still need to be clarified. Before concentrating on requirements and specific issues, we start with a few definitions.

A frequently cited definition is that of [Szyperski 2002]:

> A *software component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

We complement this definition with the following ones, borrowed from [Heineman and Councill 2001] (we have modified their definition of "software component infrastructure").

> A *software component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.
>
> A *component model* defines specific interaction and composition standards. A *component model implementation* is the dedicated set of executable software elements required to support the execution of components that conform to the model.
>
> A *software component infrastructure*,[1] or *framework*, is a software system that provides the services needed to develop, deploy, manage and execute applications built using a specific component model implementation. It may itself be organized as a set of interacting software components.

While these definitions are quite general and need to be refined (which we do in the following sections), they are complementary. Both agree on the role of a component as a unit of independent deployment; [Szyperski 2002] stresses explicit dependencies and contract-based specifications (contracts are implied by the notion of an interface, see 2.1.3), while [Heineman and Councill 2001] point out the need for a component model, a composition standard, and a component infrastructure.

More precisely, several levels of component models may be considered (Figure 7.1).

An *abstract model* defines the notion of a component and the related entities, together with their mutual relationships (naming, binding, inclusion, communication). This model may or may not have a mathematical foundation.

A *concrete model* defines an actual representation for the above entities and relationships. For example, communication and binding may be specified in terms of interfaces, method calls, signals, etc. A given abstract model may have several different concrete representations (for example, communication may be based on message passing or on procedure calls; binding may be static or dynamic). A concrete model may be complemented with a *type system*, which allows properties to be specified and verified for binding and communication, thus increasing the safety of applications.

In turn, a concrete model may have specific implementations for different programming languages. Examples of abstract, concrete, and language-specific models are provided in this chapter.

[1]the original quote says " A *software component infrastructure* is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications".

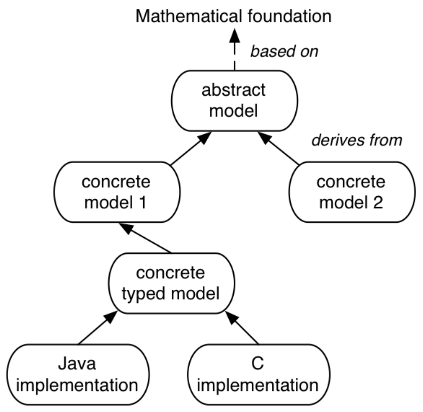


Figure 7.1. Component models

Object-based frameworks, as presented in Chapter 5, are a first step towards a software composition model. They provide encapsulation, i.e., separation between interface and implementation, and mechanisms for software reuse, in the form of inheritance and delegation. However, they suffer from a number of limitations, which have motivated the current developments towards a component model.

- There is no formal expression of the resources *required* by an object, be they interfaces of other application objects or system-provided services; the interface of an object only defines *provided* services.
- There is no global, architectural view of an application. An application may be described as a collection of objects, but there is no explicit description of the structural relationships between these objects.
- There is no provision for the expression of non-functional (i.e., not application-specific) requirements, such as performance or security, although some attempts have been made towards that goal in the form of extensions to interface descriptions.
- There is only limited provision for such aspects as deployment (the installation and initialization of the application on different sites) and administration (e.g., monitoring, reconfiguration).

Two main consequences follow: (a) some parts of the life cycle are not adequately covered, and application evolution is problematic, in the absence of global description facilities; and (b) application developers must explicitly take care of such system-related services as persistence, security and transactions, in addition to the application-specific functionalities. This has the following drawbacks.

- The application code is difficult to understand and to maintain, because the parts related to system services are intertwined with those specific to the application.
- Work is duplicated. The developments linked to system services have to be redone for each application, which is detrimental to overall quality since resources are diverted from application-specific work.

In order to overcome the above limitations, the goal of component-based systems[2] is to enhance reusability by shifting work from development to integration, to provide generic support for common services in order to allow the developers to concentrate on the application's needs, and to facilitate application maintenance, evolution and administration.

The need for a conceptual basis for software composition has been perceived for a long time. However, different models have been developed in response to the various requirements of software design, configuration, and maintenance. The need for a common, unified model suited for all phases of the software life cycle is now recognized (see [van der Hoek et al. 1998]). Our presentation aims at exposing such a common view.

The rest of this chapter is organized as follows. In Section 7.2, the main requirements for a component model are developed. The main paradigms of software composition are introduced in Section 7.3, and the elements of a component model are presented in Section 7.4. Section 7.5 describes the main patterns and frameworks found in current infrastructures for component systems. The final sections present case studies of recent systems.

7.2 Requirements for Components

The requirements for a software development system based on components are motivated by the needs of application designers and users, which cover all phases of the software life cycle. We separate the requirements of the component model proper from those of the supporting infrastructure. However, the two sets of requirements are not independent: some features of the model need support from the infrastructure, while some functions of the infrastructure rely on aspects of the model (e.g., administration uses evolution and adaptation). This analysis is inspired, with some variations, by [Bruneton et al. 2002].

7.2.1 Requirements for a Component Model

The requirements cover the following aspects: encapsulation, composability, global description, reusability and evolution.

Encapsulation. The first requirement for a component model, encapsulation, stems from the need for independent composition. It has already been formulated in Chapter 5. Encapsulation means that the only way to interact with a component is through one or several well-defined interface(s) provided by the component. An interface specifies a set of access points, to be used according to prescribed rules (e.g., read-only for an attribute, synchronous call for a procedure, etc.). No element of the internal structure of the component should be revealed, other than those which are part of the interface, and the user of the interface should not rely on any assumption about its implementation. The set of interfaces provided by a component actually defines the services that this component makes available to other components.

[2]Components may be defined at different levels of granularity. In the context of middleware systems, components are usually large grained entities, which provide fairly complex functionality. On the other hand, most components used in GUI (Graphical User Interface) composition systems are simple entities such as buttons, cursors, or other graphical widgets. While the main concepts of composition are independent of the size and complexity of the components, the support infrastructures are different.

Composability. The second requirement is composability, the capability for a component to be assembled with other ones (the result of this assembly process is called a *configuration*). This requirement may be further refined as follows.

- *Explicit dependency.* Since a component may only be accessed through a specified set of interfaces, it may only be assembled with components that use some of these interfaces according to the prescribed rules. Component A *depends on* component B if A needs to use an interface provided by B in order to provide its own service. These dependencies should be made explicit, i.e., each component should declare the interfaces that it requires, in order to check that the composition process preserves completeness, i.e., there is at least one provided interface that conforms to each required interface[3].

  The specification of explicit dependencies allows the notion of conformance to be extended to the components themselves, by specifying the conditions under which a component may be replaced by another "equivalent" component (7.4.2).

- *Closure and containment.* The composition process needs to be as flexible as possible, i.e., it should impose minimal constraints as to the semantics of composition. It also should be orthogonal to the process of designing individual components, i.e., the design of a component should not depend on the structure of the assemblies that will include this component. For these reasons, it is desirable that the composition operation be closed, i.e., that an assembly of components (a configuration) be itself a component. Such a component is said to be *composite*, and to *contain* the components that compose it.

- *Sharability.* A consequence of the containment property is that a component may be part of several different configurations. In that case, the component can be either duplicated or shared. Duplication entails the need for consistency management in order to maintain the logical image of a single component; therefore sharability, the ability for a component to be shared by several configurations, is a desirable property.

In summary, the following relationships have been identified: between components: *depends on*, *is part of*, *contains*; between interfaces or between components: *conforms to*; between component and interface: *provides*, *requires*. A component model should provide specific, precise definitions of these relationships.

Global Description. The third requirement is the need for global description capability, i.e., a high-level description of the overall structure of the application in terms of compositional entities such as components and connectors (more details in Section 7.3). This description should respect the encapsulation principle, i.e., it should not be concerned with the implementation details of the individual components.

The notation used may take different forms, e.g., one of the varieties of architecture description languages (ADLs). The composition relations defined above need to be made explicit in a global description.

[3]recall (cf 2.1.3) that an interface *I1* is said to *conform* to an interface *I2* if if an object that implements all methods specified in *I1* may be used anywhere an object that implements all the methods specified in *I2* may be used.

Reusability. Components should be *reusable*, in order to capitalize on the resources invested in their development. Reuse entails some degree of variability to adapt to different requirements. Therefore there is a need for a form of component prototypes, or templates, which allows for well localized modifications while preserving the essential features of the component's design.

Evolution. An application made up of an assembly of components should be able to *evolve* in response to changing user needs and operating context. The evolution capability should apply both to individual components and to the configuration as a whole. It may take different forms.

- Individual components should be *adaptable*. The adaptation of a component should respect its interfaces (a change in an interface concerns the overall configuration because of its potential impact on other components). Adaptation usually entails introspection capabilities (the ability of a component to get information on its own state and mode of operation), and possibly the availability of a control interface, which allows the behavior of the component to be modified[4].
- The overall structure of the application should be *reconfigurable*. Reconfiguration may be static (i.e., it only takes place when the application is not executing) or dynamic (i.e., it may be performed on a running application). Reconfiguration may take various forms: modifying the interface of components, modifying the connections between components, replacing a component by another one, adding and removing components (this involves modifying the connection structure as well).

Adaptation and reconfiguration are further examined in Chapter 10.

7.2.2 Requirements for a Component Infrastructure

The requirements apply to application management and to common services.

Application Management The capabilities of a component infrastructure should not be limited to application development, but should also apply to application management, which covers the following aspects.

- *Deployment*. Deployment is the process of making an application available for use, by creating or selecting the relevant parts, installing them on the selected sites, providing them with the required resources and environment, starting them and providing their functional and administration interfaces to users.
- *Monitoring*. Monitoring implies a data collection function, in order to be able to measure performance and load factors, to maintain an up to date image of the current configuration, and to detect alarm conditions and failures.

[4]Note that reusability may be viewed as a special case of adaptability, only entailing static changes (e.g., regenerating a new variant of a family of components), as opposed to the dynamic evolution capabilities required by adaptability in the broad sense.

- *Self-management.* The goal of self-management is to keep the application available and up to its requirements, in spite of undesirable events such as failures, load peaks or attacks. This may imply evolving the application (e.g., through dynamic reconfiguration) to meet changing requirements and operating conditions.

These various aspects of application management are examined in Chapter 10.

Common Services. Common services are concerned with properties usually known as *extra-functional* (2.1.3). This term refers to properties that are not directly expressed in the components' interfaces; but this situation is bound to change, as interface descriptions may eventually evolve to cover these properties as well. It is more appropriate to characterize extra-functional properties as being of a general nature, not directly related to the specific functions of an application. Such properties are supported by the environment in the form of reusable services, which are common to all applications, although they may be adapted to each application's specific needs. Usual instances of common services are the following.

- *Persistence.* Persistence is a property that allows data to survive its creator process or environment; it is usually (not necessarily) associated with database management. Persistence is the subject of Chapter 8.

- *Transactions.* The transactional execution of a sequence of actions guarantees atomicity, consistency, isolation and durability (the so-called ACID properties). Transactions are usually associated with the management of persistent data. Transactions are the subject of Chapter 9.

- *Quality of Service* (QoS). Quality of Service covers a range of properties, including performance factors (e.g., for multimedia rendering), security, availability. These aspects are examined in Chapters 12, 13, and 11, respectively.

A component infrastructure should provide common services to the applications. The application developer should be able to specify the required services, but should not be concerned with the actual provision of these services to the application. In other terms, the implementation of the services and the insertion of the needed calls in the code of the application should be in charge of the infrastructure. This is an application of the general principle of separation of concerns (1.4.2).

No current component-based system actually satisfies all of the above requirements. For instance, the component models used by current industrial component systems fail to satisfy the closure and containment requirements (e.g., in Enterprise JavaBeans, an assembly of several beans is *not* a bean). Few component models support shared components.

7.3 Architectural Description

This section is an introduction to the main paradigms of software composition, from an architectural point of view, i.e., with focus on structural properties, not on implementation aspects. This presentation is intended as a general framework for the definition of component models, not as a description of a specific model. A given model does not necessarily embody all aspects of this framework.

It is important to note that the elements described in this framework not only exist as design and description elements, but remain visible as run-time structures. In other words, *the notion of a component is preserved at run time as an identifiable entity.* There are two main reasons for this.

- Adaptability. Dynamic adaptation (component evolution, reconfiguration) entails the need for the components to be identified at run time.
- Distribution. Distributed components, by construction, keep their identity at run time, specially if mobility is allowed.

This is contrary to the situation of many environments based on modules or objects, in which these entities only appear at the source code level and are not present as such in the run time structures.

7.3.1 Compositional Entities

A compositional architecture defines the global organization of a software system as an assembly of parts. It can be described in terms of components (the parts), connectors (the devices that connect the parts together), and composition rules. While many architectural models have been proposed, these notions are common to all models.

- A *component* performs a specified function and can be assembled with other components; to that end, it carries a description of its required and provided interfaces, in addition to a specification of its function. The precise form of these interfaces, e.g., procedures or events, parameter description and typing, etc., depends on the specific component model. The only way to use a component is through its provided interfaces.
- A *connector* is a device whose function is to assemble several components together, using their required and provided interfaces. A connector has two functions: *binding* (in that capacity, a connector is an instance of a binding object, as defined in Chapter 3); and *communication*. In the simplest case, when the connected components reside in a single address space, a connector may only consist of a pointer or a pointer vector. When the connected components are located on different nodes, the connector implements a communication protocol and possibly stubs for format conversion.
- A set of components linked together by connectors is called a *configuration*. A configuration may or may not be itself a component, depending on whether the component model has the closure property (7.2.1).
- Composition rules specify the allowed ways of assembling a configuration out of components and connectors. Examples of aspects covered by composition rules include visibility (what components and interfaces are visible from a given component)

and conformance (under what conditions a provided interface can be connected to a required interface).

Note that the difference between components and connectors is one of function, not of nature: a connector is a special instance of a component that acts both as a binding object and as a communication channel between other components. It is a matter of convenience to make a distinction between the "ordinary" (or functional) components, which implement the functions of an application, and the communication components, or connectors, whose function is to ensure communication (and possibly interface adaptation) between the functional components. The main benefits of making this distinction are to isolate the issues related to communication (another instance of separation of concerns), and to exhibit a number of generic communication patterns, as described in Section 7.3.3.

A taxonomy of connectors has been proposed in [Mehta et al. 2000]. According to it, a connector provides a service that combines four aspects: communication (data transmission), coordination (transfer of control), conversion (interface adaptation), and facilitation (provision of extra-functional services to facilitate the above interactions). The taxonomy defines the main types of connectors (e.g., procedure call, event, stream, etc.) and specifies, for each type, the dimensions that characterize it (e.g., parameters, invocation mode, etc. for a procedure call).

With the compositional approach, an application developer constructs an application by integrating existing and new components, using the appropriate connectors. Therefore tools are needed to generate connectors and to adapt and combine existing ones. Binding factories (3.3.2) provide such tools for the most common cases (e.g., stub generators). Specific tools for adapting and combining connectors use the techniques for software adaptation described in 2.4, as illustrated in [Spitznagel and Garlan 2001].

7.3.2 Architecture Description Languages

After defining the entities that allow a complex system to be composed out of parts, the next step is to describe the global structure of a compound system, using these entities as building blocks. A number of notations, known as *Architecture Description Languages* (ADL) have been proposed to that effect. This section is a brief review of this area.

ADLs: Objectives and Main Approaches

An ADL is a formal or semi-formal notation that describes the structure of a system made of an assembly of components, with the two following main goals.

- It provides a common global description of the system, which may be shared by designers and implementers and is a useful complement to the system's documentation.
- It can be associated with various tools that use it as an input or output, e.g., tools for graphical visualization and composition, verification, simulation, code generation, deployment, reconfiguration.

As is apparent from the second item above, a number of different aspects may be represented in an ADL, in addition to the structural aspects proper (i.e., the parts and their connections). It is therefore not surprising that little consensus has been achieved on what aspects of the architecture should be represented. Likewise, there is no general agreement on the degree of formality that an ADL should provide.

Considering the wide range of potential uses of an ADL, three approaches may be taken.

1. Defining a single, general purpose ADL, with sufficient flexibility to capture all potentially relevant aspects.
2. Defining a set of specialized ADLs, each tailored to answer a specific need.
3. Defining a core ADL, together with extension mechanisms to adapt the language and its associated tools to changing requirements.

The first approach has been followed by Acme [Garlan et al. 2000], with the objective of providing a generic format for the interchange of architectural designs. Acme also serves as an architecture description language in its own right, and provides a certain degree of extensibility through properties (in the form of arbitrary name-value pairs) attached to its elements.

The second approach has given rise to a variety of specialized languages and tool-sets, which may be classified in two main groups according to the services provided by the tools:

- Assistance for system design and analysis. Examples include Rapide [Luckham and Vera 1995], used for event-based simulation and Wright [Allen 1997], used for system analysis based on a formal description of components and connectors.
- Assistance for actual system implementation and evolution. Examples include Darwin [Magee et al. 1995], used for the construction of hierarchically structured systems (Darwin also allows proving properties using a process calculus), Knit [Reid et al. 2000], used for building component-based systems software, and Koala [van Ommering et al. 2000], a language derived from Darwin, used for managing product line variability in a consumer electronics environment.

While each language may be well adapted to its purpose, the different languages are mutually incompatible and lack extensibility. A form of specialization is to design an ADL as an extension to a particular programming language, thus ensuring conformity between architectural description and implementation. An example is ArchJava [Aldrich et al. 2002], an extension to Java.

The third approach seems to be the most promising one, since the extension capabilities are not limited by a priori decisions. The XML metalanguage is generally used as a base for extensible ADLs ([Dashofy et al. 2002] discuss the benefits of this approach). Examples of such extensible ADLs are xADL [Dashofy et al. 2005] and the Fractal ADL (7.6.3).

ADL Design Issues

As many research groups developed ADLs in the 1990s, experience was collected on the main issues arising in the design of these languages. A summary of this experience, together with a classification and comparison framework for ADLs, has been published in [Medvidovic and Taylor 2000]. We give a summary of their main conclusions.

An ADL is based on the three notions (component, connector, configuration) introduced in 7.3.1. As mentioned earlier, there is no difference in nature between connectors and components; therefore the main design issues are the same for both entities, and are related to the requirements outlined in 7.2. These issues are *interfaces* (the interaction points between a component or connector and the outside world); *types* (formally expressed, verifiable properties); and *semantics* (the description of the entity's behavior, which may or not be formal). Since in the current state of the art the expressiveness of specifications and type systems is limited, two additional issues are *constraints* (properties that characterize a system's acceptability, and which are not captured by the type system) and *non-functional properties* (properties that cannot be derived from the specification of an entity's behavior expressed in terms of its interfaces). These last two aspects are also applicable to configurations.

Specific issues for configurations are *understandability*; *compositionality* (allowing a system's structure to be represented at different levels of detail, by grouping subparts in a single component); *refinement and traceability* (enabling the correct and consistent refinement of an architecture into an executable system, and tracing modifications across the levels of refinement); *heterogeneity*, *scalability* and *evolvability*; and *dynamism* (the ability to modify a system's architecture while the system is running, and to model those changes in the architecture's description).

Finally, a group of issues are related to tool support. This aspect refers to the ability of an ADL to serve as a common representation used by a set of tools. The main functions identified for an ADL-based tool-set are the following:

Active specification (preventing or detecting errors by analyzing a system's description); multiple views (providing different representations e.g., textual and graphical, of a system's description); analysis (establishing syntactic and semantics correctness, simulating the behavior, verifying conformance to constraints); implementation generation; support for dynamic modification.

At the time of the above survey (2000), the main deficiencies identified in the capabilities of the existing ADLs were in the following areas: support of non-functional properties; architectural refinement and constraint specification; and support for architectural dynamism.

Evolution of ADLs: Trends and Prospects

The main trends in the evolution of Architecture Description Languages may be summarized as follows.

- *Extension mechanisms.* As said in the beginning of this section, the most promising approach in the design of ADLs is to extend the principle of modular decomposition to the ADL itself. Thus an ADL may be defined in a modular way, by providing the needed extensions to a common core. XML is widely used as a support notation, since the XML schema mechanism provides a simple way of building modular extensions.

- *Dynamic ADLs.* A dynamic ADL is one that is executed at run time and causes the system's structure to evolve, contrary to a static ADL, which describes an immutable system architecture. While many existing ADLs do support some degree of dynamism (e.g., by allowing component creation at run time), these dynamic capabilities are limited, and the range of dynamic evolution is usually restricted to predefined schemes. For instance, ArchJava uses the dynamic facilities of Java (essentially the `new` construct), but the binding of the newly created components must follow a predefined connection pattern.

Future dynamic ADLs are expected to allow unplanned system evolution. This could be achieved by integrating scripting and workflow facilities into the language, and by using run time mechanisms such as event-condition-action (ECA) rules, as described in Chapter 6. Events could be triggered from inside the system (e.g., exceptions), or from outside (external events, administrator's commands, time, etc.).

A convenient supporting mechanism for dynamic reconfiguration is reflection (2.4, 7.5.5), since the reconfiguration operations acting on the basic entities may be described at the meta level. Examples may be found in [Morrison et al. 2004], [Layaida and Hagimont 2005].

- *Towards more formality?* Early ADLs were essentially designed on an empirical base and had no formal support. Providing such a support increases safety, by allowing the designers to specify and to verify system properties. A few ADLs have a formal base, which is usually derived from a process algebra, as for example in [Bernardo et al. 2002] and [Morrison et al. 2004]. This trend should continue in future ADLs, as their increased power and complexity call for an greater degree of safety and control.

The above discussion was limited to the technical aspects of ADLs. However, other aspects such as domain specificity and business concerns are taking an increasing importance. See 7.3.4 for a discussion of this point.

7.3.3 Examples of Connection Patterns

We illustrate the main notions related to composition with simple examples, using graphical descriptions. While most of the existing notations are equivalent (i.e., they have the same expressing power), no universally accepted standard has yet emerged, as explained in Section 7.3.2. The notation that we use freely borrows elements from the ODP model (as illustrated in [Blair and Stefani 1997]), from the OMG notation for the CORBA Component Model [CCM], and from the Fractal component framework [Bruneton et al. 2002].

Client-Server Systems

The first example is that of a client-server system using a synchronous request-response communication scheme such as RPC or Java RMI. The client requires a service, defined by an interface (for this example, an interface is a set of methods, defined by their signatures). The server provides an interface that conforms to that required by the client. The client and server interfaces are linked by a connector, as illustrated on Figure 7.2a.

Another view of the same system is represented on Figure 7.2b, which shows the connector as a component. Note that the connector has interfaces, which must conform to the corresponding interfaces of the client and the server.

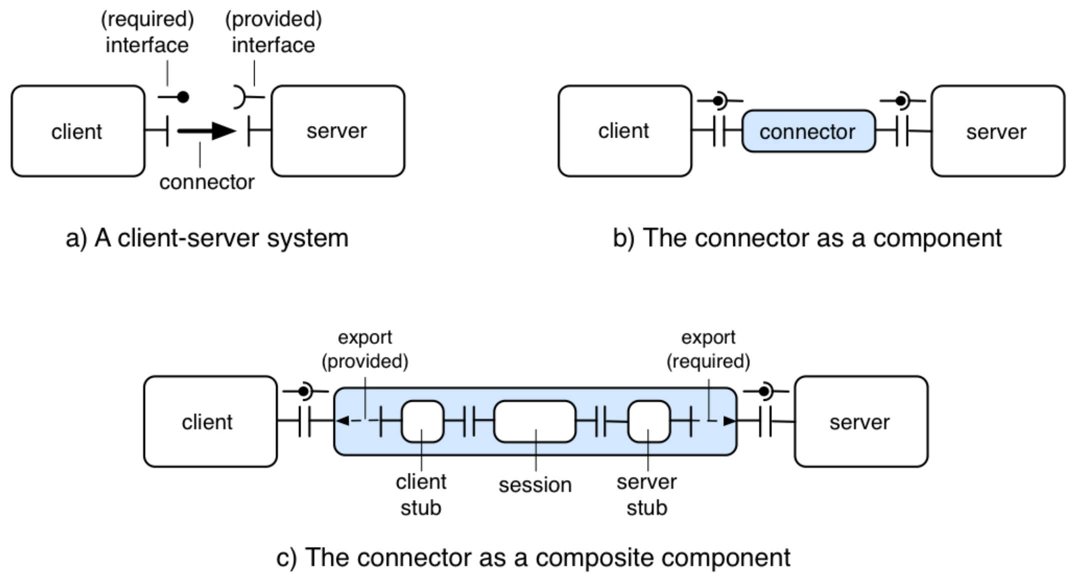


Figure 7.2. A client-server system

A more detailed view (Figure 7.2c) shows the connector as a composite component, i.e., an assembly of three subcomponents: the client stub, the server stub, and a session object used for remote communication (this latter can again be further decomposed, see Chapter 4). Note that the provided interface of the connector is connected to the provided interface of the client stub by an "export" connector (noted as a dotted arrow); in this case, this connector does nothing apart from making the interface visible, but it could, for instance, perform an interface adaptation, i.e., exporting a different (conformant) interface. Similarly, the required interface of the connector is exported from the server stub.

Other sorts of interfaces are illustrated in the next example, which describes a multimedia client-server system (Figure 7.3). There are two main differences with the previous example.

- Several clients may be connected to a server.
- The server provides to the clients a continuous stream of multimedia data, subject to timing constraints. In addition, asynchronous signals are exchanged between the clients and the server, in both directions. These signals are used for synchronization (starting and stopping the stream, controlling the data rate).

The multimedia connector encapsulates the various functions needed to exchange and to synchronize multimedia data between the server and the clients. This is a fairly complex system, and it would be useful to have a means of controlling its operation, by acting on its internal mechanisms. We come back to this question in Section 7.5.5.

Coordination-based systems

Coordination-based systems have been examined in Chapter 6. Such a system may again be represented as a set of clients linked by a connector. Two typical examples are a

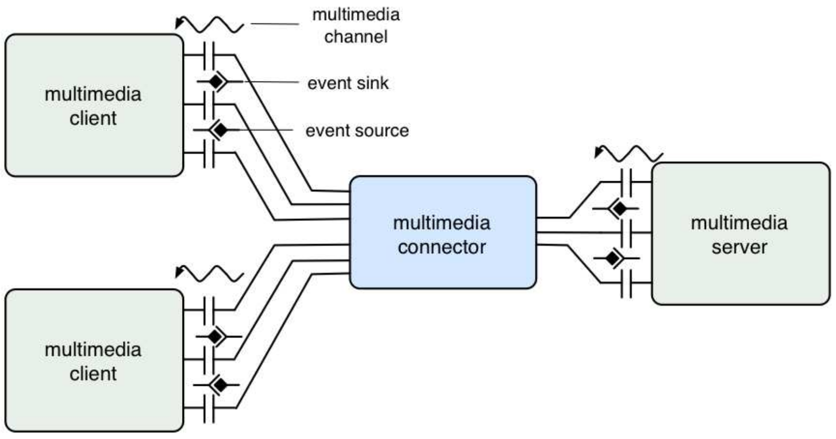


Figure 7.3. A multimedia system

mediation system, such as the one described in Chapter 1, and a cooperative document edition system. In each case, a variant of the *publish-subscribe* communication paradigm is used. The minimal interface includes the synchronous *publish* and *subscribe* operations, and an asynchronous *notify* signal that is sent to subscribers when an event to which they subscribed occurs. A generic form of a coordination system is represented on Figure 7.4. Note that a client may act as a subscriber, as a publisher, or as both.

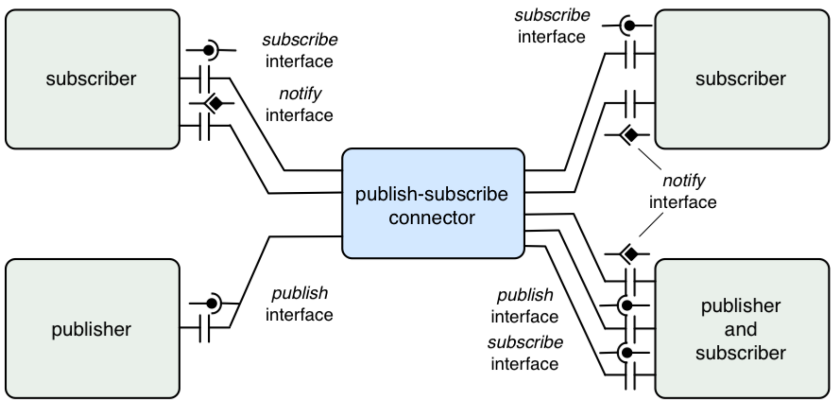


Figure 7.4. A coordination-based system

The coordination connector may encapsulate a variety of mechanisms, including a message bus or a system based on a shared workspace, examples of which include Linda or Jini.

Both the multimedia and the publish-subscribe system must accommodate a varying number of participants, which implies a mechanism for dynamic connection and disconnection. There are three possible approaches.

1. creating a connector with a fixed number of interfaces of each type, thus setting an upper limit to the number of connections of each type.

2. allowing a connector interface to be multiplexed between several participants.
3. allowing dynamic creation of interfaces of a given type. For instance, if a new subscriber is admitted to the publish-subscribe system, a new subscriber interface is created.

Solution 1 is overly rigid. The other solutions raise the issue of multiplexing, which is done at different levels in solutions 2 and 3. Solution 3 combines architectural simplicity (one-to-one interfaces) and flexibility; see an example in 7.6.1.

7.3.4 Software Engineering Considerations

Up to now, we have examined the technical aspects of component-based software architecture. However, other points of view are equally important. In [Medvidovic et al. 2007], the authors define three "lampposts", or sources of insights, under which software architecture, and specially ADLs, may be examined. These are *technology*, *domain*, and *business*.

The technology approach is mainly guided by the tools and environments that may be supported by an ADL, specially as regards the relationships between a system's description and its implementation.

The domain approach takes into account the specific concerns, constraints and properties of a particular application domain. This approach has led to the development of domain-specific languages, and may be extended to architecture descriptions. The main difficulty is to capture in a rigorous notation the properties that characterize a domain, as the knowledge of these properties is part of the domain's specialists' experience and is not always explicitly expressed.

The business approach is concerned with organizational aspects of software development. These aspects include the product strategy (defining various product lines targeted to specific market segments, and how these products are mutually related), and the processes used by the software development organization to create, manage and evolve its products. Here again the difficulty lies in making these elements explicit, and in identifying which elements may be captured by a formal description such as provided by an ADL.

In conclusion of their analysis, [Medvidovic et al. 2007] argue that the next generation of ADLs should provide better support for the domain and business aspects, but that it is unlikely that such a wide range of concerns could be adequately captured by a single notation (even with multiple views such as provided by UML 2.0). Therefore a wide coverage should be best achieved using extension techniques.

7.4 Elements of a Component Model

Recall (7.1) that a component model defines specific standards for the various properties of a component, while a component framework provides the services needed to actually use component implementations conforming to the model supported by this framework. In this section, we attempt to identify those properties of an abstract component model that do not depend on a specific supporting framework. Since components cannot be used independently from a framework, such separation is not always easy. Framework-specific aspects are examined in Section 7.5.

There is currently no single universally accepted component model. In this presentation, we do not attempt to cover the various aspects of a model in their full generality. The main restrictions apply to the dynamic behavior of components; for instance, we assume that the type of a component (defined in 7.4.2) does not change during its lifetime, although some models allow the implementation of a component to be dynamically modified.

We examine the following points: naming and visibility (7.4.1); component lifecycle (7.4.4); component typing and conformance (7.4.2); component binding and configuration (7.4.3); component control (7.4.5).

7.4.1 Naming and Visibility

The general principles of naming (Chapter 3) apply to components. The basic notion is that of a naming context, an environment for name definition and resolution. Usually, a component *Comp* has a name *name* in a naming context *NC* in which it was created. Then, if *ctx* is a universal name of *NC* (i.e., a name valid in all contexts, such as an URL), *ctx:name* is a universal name for the component *Comp* (such a name is often called a *reference* for *Comp*).

One specific feature of naming for components is that a component may itself take the role of a naming context. In this context, external components may be designated by local shorthand names. These names are mapped to references using a local table. In addition, if the model allows composite components, the included components are also designated by local names. Included components may or may not be made visible from outside the composite component, according to the visibility rules define by the model. Components that are shared between several naming contexts have a name in each of these contexts, and these names are mapped to the single reference of the shared component.

A typical use of a naming context is to group a family of things that are related together, like in the above case of the internal components of a composite component. Other examples include Java packages (for components models based on Java) and OSGi bundles (7.7).

7.4.2 Component Typing and Conformance

In object models such as considered in Chapter 5, there is a distinction between a class (a generic description) and instances of this class (objects that conform to this generic description). Likewise, we make a distinction between a “component template” (a generic description of a family of components) and individual component instances generated from this template. Contrary to objects, this distinction is not usually embodied in a programming language, at least in the current state of the art.

Still following the analogy with object models, we define a notion of type for a component (the type is associated with the template). The type is defined by the set of interface descriptions, both required and provided. Instances created from a template share the type of this template.

The notion of conformance defined for interfaces may be extended to components. In a configuration composed of a set of components, consider a single component, which is bound to the rest of the configuration through a set of required and provided interfaces. Each required interface I_r of the component is bound to a provided interface S_p of the configuration, while some or all provided interfaces I_p of the component are bound to a required interface S_r of the configuration. The following relations must hold for all provided and required interfaces of the component (recall that $\sqsubseteq$ denotes the subtyping relation):

$$T(I_p) \sqsubseteq T(S_r) \text{ and } T(S_p) \sqsubseteq T(I_r)$$

Any component may be substituted to the given component, as long as the above relations hold. Note that the substitute may have more provided interfaces (but no less required interfaces) than the original component.

Therefore the conformance relationship between two components *C1* and *C2* may be defined as follows:

C2 conforms to *C1* (written $T(C2) \sqsubseteq T(C1)$) if the following conditions hold:

- *C2* has at least as many provided interfaces as *C1*, and for each provided interface $I1_p$ of *C1*, there is a provided interface $I2_p$ of *C2* such that $I2_p \sqsubseteq I1_p$.
- *C2* has as many required interfaces as *C1*, and for each required interface $I1_r$ of *C1*, there is a required interface $I2_r$ of *C2* such that $I1_r \sqsubseteq I2_r$.

By definition of conformance, if $T(C2) \sqsubseteq T(C1)$, then *C2* (more precisely, any instance whose type is $T(C2)$) may be used in any place where an instance having type $T(C1)$ is expected.

The definition of conformance may be extended to semantic aspects, if one can specify the behavior of a component in a formalism that allows equivalence to be defined, such as finite state machines.

7.4.3 Binding and Configuration

A configuration is an assembly of components. As said in 7.3.2, a configuration may be described by a notation (ADL) which specifies the components that form the configuration and their interconnection. Typically, an ADL contains statements of the form:

A.Required_interface **is connected to** *B.Provided_interface*

where *A* and *B* are components that are part of the configuration and are described by declarative statements in the ADL. Not all component systems, however, use an ADL, and the above statement is then implicit (i.e., it has to be inferred from the program of the components).

Actually producing a working configuration implies a binding phase, in which the connections between the components are created in the form of binding objects (or connectors). Depending on the component model and on its implementation, the binding may be static (preliminary to the execution) or dynamic (performed during execution, e.g., at first call), as explained in Chapter 3.

Again depending on the component model, a configuration may or may not be itself a component. In the latter case, the configuration may be activated through a specified entry point (the equivalent of a *main* procedure), which is part of the provided interface of one of its components; there may be several such entry points. If a configuration is itself a (composite) component, then the interfaces of that component must be specified, using the interfaces of its contained components.

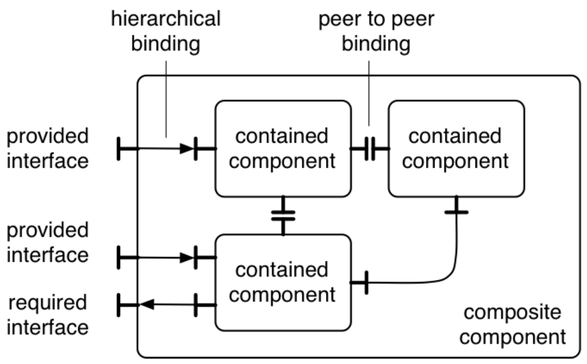


Figure 7.5. Bindings in a composite component

As shown on Figure 7.5, there are two kinds of interface bindings in a composite component.

- "peer to peer" bindings, between the components that are contained in the composite component; these are similar to the bindings found in non-hierarchical component systems;
- "hierarchical", or "export", bindings, which associate provided or required interfaces of the composite component with interfaces of the contained components.

Both types of bindings are controlled by the manager of the composite component (7.4.5).

A final point regards multiple connections to an interface. As said in 7.3.3, one way of multiplexing an interface *Int* of a component is to dynamically create a new instance Int_k of this interface at each binding of *Int* to another interface. Associating an execution unit (process or thread) with the new interface may be done by one of the methods described in 1.7. Dynamic interface instantiation is done by the component manager.

This mechanism bears some analogy with the creation of sockets when a new client connects to a server socket, which may be considered as a low-level mechanism for dynamic interface instantiation (see 3.3.2). Note that dynamic interface instantiation is also applicable to required (client) interfaces.

7.4.4 Component Lifecycle

The lifecycle of a system describes its evolution from creation to deletion. We first consider the simple case of a single, primitive component. We then examine the problems posed by starting and stopping a configuration.

We only present a generic framework for defining the main lifecycle aspects of a component model. Existing component systems propose many variations and extensions of this scheme.

The Lifecycle of a Primitive Component

Creating instances of a primitive component described by a certain template may be done through a factory (2.3.2) associated with the template. The factory is associated with a naming context: it delivers a name in this context (a reference) for the newly created component. The factory may then act as a manager of the instances that it has created. It therefore supports lookup and remove operations, using component references as parameters. More elaborate lookup mechanisms using higher level names are usually available in component frameworks.

The template itself may be created by a template factory, using a description of the template's type (the set of its required and provided interfaces) and implementation.

Once created, a component is installed, i.e., made accessible in the system through a reference. A component usually depends on other resources (e.g., services provided by the infrastructure or by other components. These dependencies must be resolved through a binding process (see 3.3), which may be static (before execution) or dynamic (at run time). A (partially) resolved component may then be activated. An active component may be used through its provided interfaces; it may be stopped (it then ceases all activity until restarted).

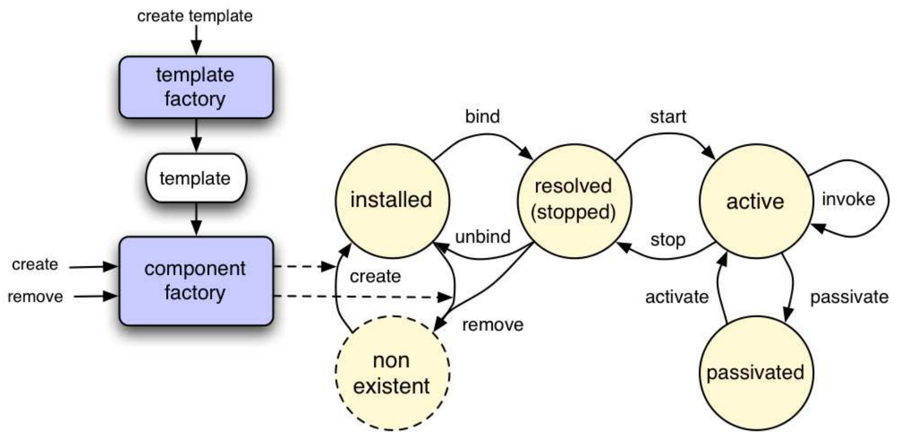


Figure 7.6. A component's lifecycle

Some component models (e.g., Enterprise JavaBeans) define a "passivated" state, in which the internal state of the component is frozen, and its representation is saved. A component may not be used while in passivated state. The motivation for component passivation is twofold[5].

- To capture the state of the component at a given point of its execution, by freezing its evolution. The representation of the passivated component is the analog of a

[5]Note that the first motivation is intrinsic to the component model, while the second one is of a technological nature, i.e., it would disappear if there were no resource limitations.

serialized Java object. It may be passed as a parameter, or used for debugging, replication, or persistent saving.

- To spare resources (specially memory) during execution, by passivating a temporarily unused component and saving its state to persistent storage. When the component is needed again, its state is restored and the component is reactivated.

The state diagram of a component's lifecycle is represented on Figure 7.6.

Starting and Stopping a Configuration

The problem of starting and stopping a configuration, as opposed to a primitive component, is a complex one, because of the dependencies between components.

A component A *directly depends* on a component B if the correct execution of A relies on the correct execution of B. In terms of bindings, A *directly depends on* B if a client interface of A is bound to a server interface of B. More generally, the *depends on* relation is defined as the transitive closure of *directly depends on*.

When starting a configuration (an assemblage of components), the following invariant must be preserved: if component C_i *depends on* C_j, then C_i may only be started if C_j in the active state. Circular dependencies, if any, should be treated as a special case (e.g., by defining sub-states within a component, corresponding to the activation of different threads executing different operations). A configuration usually defines one or several entry points, i.e., methods in specific components to be called to start the configuration (the equivalent of a `main()` method in a C or Java application).

Stopping a configuration is more complex, since a component that is in the stopped state may be re-activated by a call from another, still active, component. To take this situation into account, one defines the notion of quiescence: a component is *quiescent* if (i) the component is stopped (no thread executes one of its methods), and its internal state is consistent; and (ii) it will remain stopped, i.e. no further calls to methods of this component will be issued by other components. The configuration is stopped when all its components are quiescent.

Quiescence detection is examined in 10.5.3 (about reconfiguration), since quiescence is also a prerequisite for dynamic reconfiguration. *directly depends on*.

7.4.5 Component Control

The notion of a *component manager*[6], i.e., an entity that exercises control over a component or set of components, is present in all component models, under various forms. The main functions related to components are lifecycle management, binding, and interaction with other components. The role of a component manager is to perform these functions on behalf of a component, thus acting as a mediator between the component's internals and

[6]The notion of a manager is present in the Reference Model for Open Distributed Processing (RM-ODP) [ODP 1995a], [ODP 1995b], although this model does not explicitly refer to components. RM-ODP defines a *cluster* as a group of objects forming a single unit, together with a cluster manager that controls the cluster's lifecycle and performs various administration functions.

the outside world. The main motivation for this structure is separation of concerns: isolating the functions of the application proper from those related to execution mechanisms, to administration, or to the provision of extra-functional properties.

With this two-level organization, a component's manager screens all interactions of the component with other components and with the environment (e.g., system services). The interaction of a component with its manager (Figure 7.7) thus follows the pattern of inversion of control (2.1.1).

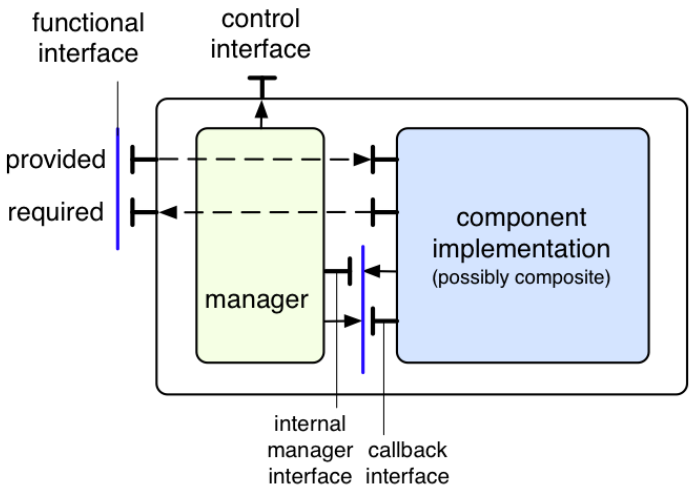


Figure 7.7. Component-manager interaction

At this level of description, we do not make any assumption on the actual implementation of the manager. This aspect is examined in 7.5.

The following functions are provided by a component manager.

- The functions related to the component's lifecycle (7.4.4).
- The support of binding functions for the component, through its provided and required interfaces.
- The mediation of the interactions of the component with other components.
- The functions related to the component's description to be provided to other components. This function may be extended to more general forms of introspection or self-modification, if the model includes a form of reflection.
- If the model includes composite components (i.e., if an assembly of components is a component), the manager controls the composition process and manages the "inside" of a component (i.e., the components that compose it).

Additional framework-related functions (management of system services) are discussed in 7.5.

7.5 Patterns and Techniques for Component Frameworks

The function of a framework, or infrastructure, for components is to provide support for all the post-design phases of the life cycle of a component-based application: development, deployment, execution, administration and maintenance. In this section, we examine the organization of these infrastructures. Specific examples are developed in the following sections.

7.5.1 Functions of a Component Framework

A framework for component support must provide the following functions to the components that it manages.

1. Support for lifecycle: creating, finding, passivating and activating, deleting instances of components. Additional framework-specific functions include pooling (managing virtual instances of components, see 7.5.3).

2. Support for functional use, i.e., binding a component to other components, and using a component through its provided interfaces; this may include local as well as remote invocations.

3. Support for the provision of extra-functional properties, as described in 7.2.2 (persistence, security, availability, QoS), and for the provision of system-managed services, such as transactions (7.2.2). The framework usually acts as a mediator to the providers of such services.

4. Support for self-management of components and configurations. This includes adaptation and introspection (reflective capabilities); for composite components, this also includes managing included/shared components (7.4).

There is no standard terminology for component support frameworks. The most common notion is that of a *container*, defined as an environment for one or more component(s), which screens the components from the outside world by intercepting incoming and possibly outgoing calls, and acts as a mediator for the provision of common services. Due to historical reasons, the term "container" is often connoted as a heavyweight structure, although lightweight containers are common in recent frameworks such as Spring [Spring 2006], PicoContainer [PicoContainer] and Excalibur [Excalibur]. Some systems define their own terminology (e.g., component support in Fractal (7.6) is done by a set of *controllers*).

Two major qualities required from a component framework are adaptability and separation of concerns, which may be obtained using the techniques examined in Chapter 2.

In the rest of this section, we develop different aspects of component framework design: using inversion of control to achieve separation of concerns (7.5.2); efficiently implementing the factory pattern (7.5.3); using interceptors (7.5.4) to control component interaction with the outside world; using reflection and aspects (7.5.5) to achieve framework adaptation.

7.5.2 Inversion of Control and Dependency Injection

The inversion of control pattern (2.1.1) is a basic construct of frameworks for component management. It appears in various forms, some of which are presented in this section. The common feature is that the initiative of a controlled operation resides with the framework, which calls an appropriate interface provided by the component. When recursively applied (i.e., a callback triggers another callback, possibly on a different component), this mechanism is extremely powerful.

The main purpose of inversion of control is to achieve separation of concerns (1.4.2), by isolating the application proper from all decisions that are not directly relevant to its own logic. A typical example is the management of *dependencies*: when a component needs an externally provided resource or service for its correct execution, the identity of the resource or of the service provider is not directly relevant, as long as it satisfies its contract (2.1.3). Therefore this identity should be managed separately from the component's program, and interface-preserving changes should not impact this program.

One solution relies on a separate naming (or trading) service, using the Broker pattern described in 3.3.4. Dependent resources are designated by names (or by properties in the case of trading). This solution makes the application dependent on the (externally provided) naming or trading service. Requests to that service must be explicitly formulated. The alternative solution presented here relies on a configuration description to determine dependencies and to trigger the appropriate actions.

Dependency injection [Fowler 2004] is a pattern that applies inversion of control for the transparent management of dependencies. Although it is usually presented in the context of Java-based components, it has been applied to a variety of situations. Criteria of choice between a naming service (locator) and dependency injection are also discussed in [Fowler 2004].

The problem of dependency management is illustrated by the following simple example. An application consists of two components, *compA* and *compB*, defined by their types (i.e., set of interfaces), *A* and *B* respectively. *compA* uses the interface of *compB* (as defined by its type *B*). *compB* has an attribute *v*, whose initial value is a configuration parameter of the application.

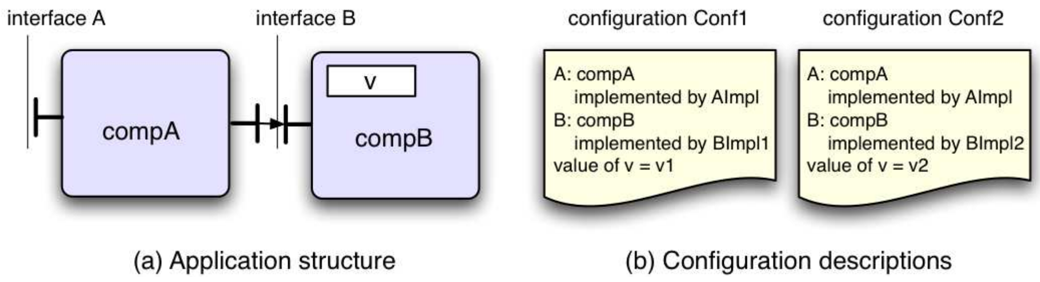


Figure 7.8. Organization of a simple application

This (abstract) organization is shown on Figure 7.8a. Two concrete implementations (or configurations) are described on Figure 7.8b. In both *Conf1* and *Conf2*, *compA* is an instance of an implementation *AImpl* of type *A*. In *Conf1*, *compB* is an instance of an implementation *BImpl1* of type *B*, and the parameter *v* has value *v1*. In *Conf2*, *compB* is an instance *b2* of an implementation *BImpl2* of type *B*, and the parameter *v* has value *v2*. The abstract structure of the application is thus separated from configuration decisions. The goal is to preserve this separation in the actual implementation: changing from *Conf1* to *Conf2* should keep the application code invariant.

In [Fowler 2004], three forms of dependency injection are used to solve the dependency problem. We present them in turn, using the above example throughout.

Constructor Injection

This form of injection uses factories. A factory (2.3.2) is a device that dynamically creates components of the same type. An example is the constructor of a Java class, which is used to create instances of that class, possibly with different initial values for its attributes.

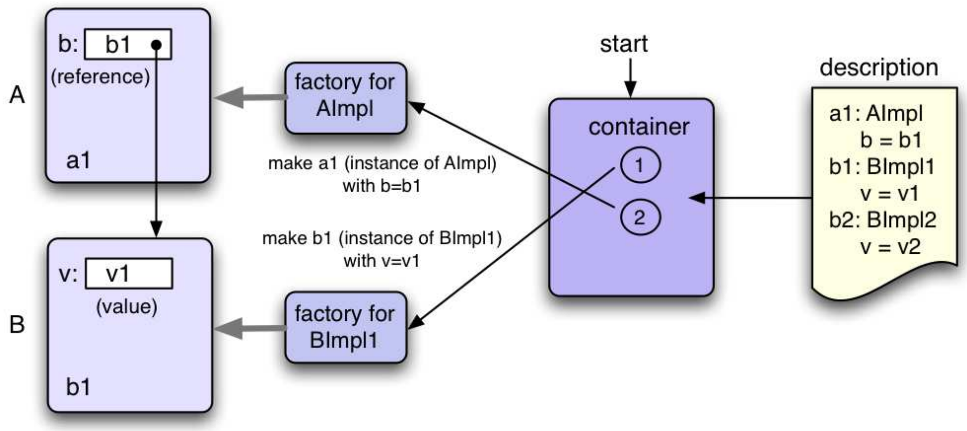


Figure 7.9. Inversion of control: constructor injection

The mechanism used when starting the application is shown on Figure 7.9. The container calls in turn the appropriate factories for *compA* and *compB*, with the appropriate parameters ("appropriate" means as defined by the configuration), creating component instances called *a*1 and *b*1, respectively. The reference to *compB* in *a*1 has value *b*1, and the attribute *v* in *b*1 has value *v*1. In the organization shown, the program of the container uses a configuration description file; in that case, the container must determine the order in which the constructors should be called, by analyzing the configuration file to determine dependencies. Alternatively, the configuration parameters could be directly embedded in the program.

This pattern is used, among others, in Spring [Spring 2006] (a lightweight container framework for Java applications), as well as in the Enterprise Java Beans (EJB) 3.0. container [EJB]. EJB is the component model and framework defined by the JEE standard [JEE].

A form of constructor injection is also used in the configuration framework of Jonathan [Jonathan 2002]. In this solution, a configurator program is first generated from a configuration file, and executed at application startup.

Setter Injection

In setter injection, the process of creating a component is separated from that of setting its attributes. When a component is created by a factory, some or all of its attributes are uninitialized (or set to a default value such as `NULL`). Each attribute may then be initialized by calling the corresponding setter operation. The overall process is shown on Figure 7.10

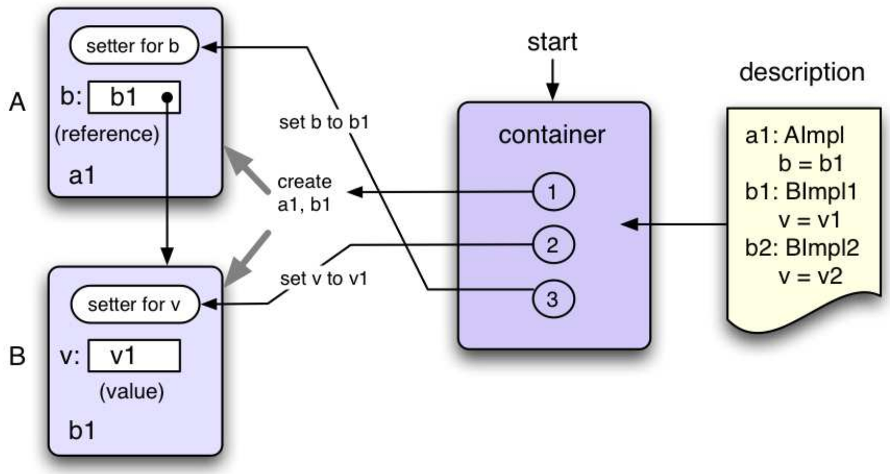


Figure 7.10. Inversion of control: setter injection

This pattern is also used in the above mentioned frameworks (Spring and EJB 3.0). The choice between constructor and setter injection is discussed in detail in [Fowler 2004]. Both mechanisms are useful in different situations, and may be combined. Constructors allow better encapsulation, since an attribute that is set in a constructor can be made immutable by not providing a setter for it. On the other hand, setters provide more flexibility, since some attributes may be set or modified after component construction. Setters may also be preferred if the number of attributes is large, in which case constructors become awkward.

Interface Injection

Interface injection is a more general solution than the two last ones. A new callback injection interface, tailored to each component type, is added to each managed component. The operations defined by this interface perform any needed initialization. Like in constructor injection, the order in which these operations are called may be determined from the configuration description or directly embedded into the program. The overall organization is shown on Figure 7.11

As explained in [Fowler 2004], this process may be made more systematic by defining a generic injector interface, with a single operation *inject(target)*, where *target* is the component being configured. This operation, in turn has a different implementation for each component, using the component-specific injectors. The injectors (code implementing the generic injector interface) may be included in the components (which must then export the generic injector interface), or directly in the container code. At system initialization,

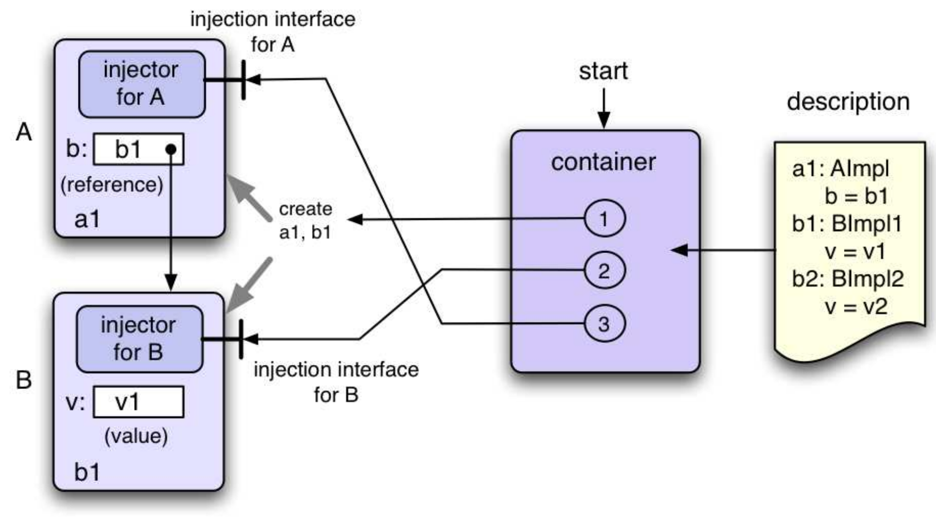


Figure 7.11. Inversion of control: interface injection

the container calls the injectors in an appropriate order, determined by analyzing the configuration description.

7.5.3 Factories

Component creation is done by a component factory (7.4.4). The implementation of a factory for container-managed components is often called a *home*. A container includes a home for each type of components that it manages.

As mentioned in 7.4.4, a component factory uses a form of component template, which may take different forms. The template may be explicitly created using a template factory, and it is used as an input by the component factory. Alternatively, the component factory may create predefined generic components, and use callback methods to initialize these components, as explained in 7.5.2.

In order to reduce the overhead of creating and deleting component instances, containers often use *instance pooling*. Instance pooling is a technique for reusing component instances. The lifecycle diagram of Figure 7.6 is extended as shown on Figure 7.12 (the "stopped" and "passivated" states are not represented). The container maintains a pool of "physical" instances, which are not assigned to a specific component. A new instance of a component is created in a "virtual" state, in which it has no physical counterpart. At first invocation, a physical instance is taken from the pool and assigned to the virtual instance, which now may go to the active state. After some inactivity period (depending on the resource allocation policy), an active component may release the pooled physical instance and return to the virtual state. This technique is similar to paging in virtual memory systems, or to the management of concurrent activities by using a pool of threads.

Note that instance pooling and passivation use similar techniques, but have different objectives. Instance pooling aims at reducing the number of component creations and deletions, while passivation aims at reducing memory occupation. In both cases, callback methods must be invoked in order to keep the component's state consistent (the state

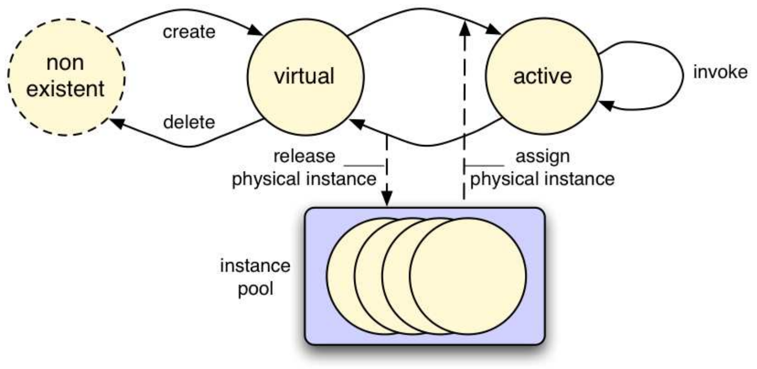


Figure 7.12. Instance pooling

must be loaded when a component is assigned a pooled instance, and saved when the instance is released). This only applies to "stateful" components; nothing has to be done for "stateless" components. Examples are examined in the case studies.

7.5.4 Interceptors

A function of a container is to act as a mediator for its contained components: any (incoming or outgoing) communication of the component with the outside world is mediated by the container.

The technique used for this mediation function relies on interception (2.3.4). The container provides two objects associated with each of its contained components (Figure 7.13).

- An external interface, which acts as an interceptor for all incoming calls to the component. This object is exported by the container, i.e., it is visible from the outside. It provides the same interface(s) as the component.
- An interceptor for outgoing calls, which is internal to the container (invisible from the outside). This interceptor may be optional, depending on the specific technology being used.

In order to reduce the cost of communication between components located in the same container, the container may also provide a local interface.

An interface of a component must be bound prior to invocation. The binding works as follows.

A component's interface (let's say interface I of component C, denoted $C.I$) may be invoked by a thread running within a client or within another component. Prior to invocation, the calling thread must bind to the called interface, using a name to designate it. Two situations may occur.

- The call originates from inside C's container (i.e., from another component in that container). The container is used as a common naming context for its contained components, and the binding operation returns a reference on the local interface of $C.I$.

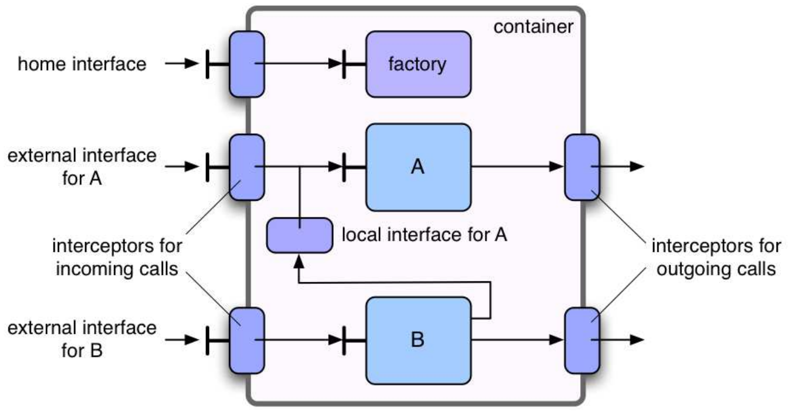


Figure 7.13. Container-mediated communication

- The call originates from outside C's container (e.g., from a component running in in a different container, or from a remote client). In that case, $C.I$ must be designated by a name in some global naming context including C's container, and the binding operation returns a reference on the external interface of $C.I$

To allow these bindings to be performed, a component must register its local and external interfaces in a local and a global registry, respectively.

Another use of interceptors is described in the next section.

7.5.5 Reflective Component Frameworks

As explained in 2.4, reflective aspects have been introduced in middleware construction to deal with highly dynamic environments, for which run time adaptation is required. One approach is to integrate reflection in a component model and framework. Thus any software system, be it an application or a middleware infrastructure, built using reflective components, may benefit from their adaptation capabilities.

Recall that a reflective system is one that provides a representation of itself, in order to inspect or to modify its own state and behavior. This is usually achieved by defining a two-level organization (2.4.1): a base level, which implements the functionality of the system, and a meta level, which allows observing or modifying the behavior of the base level.

This principle may be transposed as follows to a component framework. In addition to its functional interfaces, each component is equipped with one or several meta-level interfaces, which provide operations that allow inspecting or modifying the state or the behavior of the component. The meta-level programs cooperate with the run-time system to fulfill their function. Figure 7.14 gives a summary view of this organization.

In the rest of this section, we first discuss some implementation issues for reflective component frameworks. We next show how a reflective component framework can be used for system management.

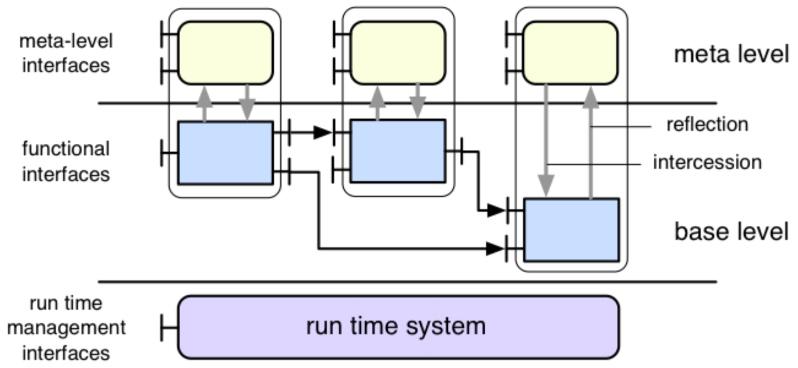


Figure 7.14. A reflective component system

Implementing a Reflective Component Framework

The main motivation for a reflective structure is adaptability, and this quality should apply to the meta-level itself. This might be achieved by adding a new reflection layer (meta-meta level), but this approach is usually considered overly complex and is seldom used.

The solution adopted by several systems is to implement the meta-level as a part of the component's manager (i.e., the software layer that acts as an environment for the components, as defined in 7.5.1), and to make the make the structure of the meta-level apparent, open, configurable, and extensible. This area is the subject of active research, and a number of experimental frameworks have been proposed. The solutions differ on the following points.

- How is the meta-level organized? In most proposals, the meta-level has itself a component-based structure (with no further meta-level). This allows a clear separation between the various functions present at the meta-level. This component structure may be flat or hierarchical.

- How does the meta-level interact with the base level? Two closely related techniques are used: interceptors (2.3.4, and 7.5.4) and aspect-oriented programming (2.4.2). Both techniques allow code sequences to be inserted at specified points in the base level; this insertion may be dynamic.

These techniques have been implemented both in heavyweight containers (such as the JBoss implementation [Fleury and Reverbel 2003] of J2EE), which uses both client-side and server-side interceptors), and lightweight component frameworks such as OpenCom [Clarke et al. 2001, Grace et al. 2006, Coulson et al. 2008], Fractal (7.6), DyMAC [Lagaisse and Joosen 2006]. Recent research [Pessemier et al. 2006a] aims at integrating components and aspects, by using aspects to implement the meta-level of a component framework, while representing aspects as components.

The insertion points at the base level (join points) are usually defined to be the incoming and outgoing calls, at a component's boundary, which respects the encapsulation principle by treating each component as a black box. However, it might be useful, in a limited number of cases, to define join points *inside* a component. This process (treating a component as a "grey box") needs to be carefully controlled since it breaks encapsulation: the internal join points must be explicitly specified in a special interface of the component. This technique has ben proposed in [Pessemier et al. 2006b], extending to components a similar proposal for "open modules" [Aldrich 2005].

Different framework implementations may be developed for a given component model. As an example, two frameworks for the Fractal component model are described in 7.6.2. One of them (AOKell) uses components and aspects to organize the meta-level, while the other one (Julia) uses non-componentized code with interceptors.

Using a Reflective Component Framework

The role of a reflective framework is to facilitate the dynamic adaptation of an application to a changing environment. Here are two examples.

- *Autonomic computing.* Autonomic computing (see more details in 10.2) aims at providing systems and applications with self-management capabilities, allow them to maintain acceptable quality of service in the face of unexpected changes (load peak, failure, attack). This is achieved by means of a feedback control process, in which a manager reacts to observed state changes by sending appropriate commands to the controlled system or application. Designing and implementing such an autonomic manager is a typical application of reflective component framework, since observation and reaction are readily represented by reflection and intercession, respectively. An example is the Jade system [Bouchenak et al. 2006], based on the Fractal component model (7.6).

- *Context-aware computing.* Wireless mobile devices need to adapt to variations of their context of execution, such as network bandwidth, available battery power, changes of location, etc. Middleware for managing such devices therefore needs adaptation capabilities, which may again be provided by reflection. An example may be found in [Chan and Chuang 2003].

Other examples include evolving a system to adapt it to changing user requirements.

7.6 Case Study 1: Fractal, a Model and Framework for Adaptable Components

Fractal [Bruneton et al. 2002, Bruneton et al. 2006] is a software composition framework, based on an original component model, that has been designed to answer the requirements presented in 7.2, with the following goals.

- Soundness: the design relies on an abstract model grounded on a strong mathematical foundation, the Kell calculus [Bidinger and Stefani 2003].

- Generality: the component model is intended to impose as few restrictions as possible (in particular, beyond the basic structural entities, all features are optional); the model is not tied to any programming language.

- Flexibility: the model is designed to allow easy reconfiguration and dynamic evolution, and the framework provides reflective capabilities.

Fractal is not a product, but an abstract framework that can be used as a basis for a family of concrete frameworks complying with the Fractal model. Several implementations of the model are available (7.6.2).

Fractal was developed as a joint effort of France Telecom R&D and INRIA, and is an ongoing project of the ObjectWeb consortium (http://fractal.objectweb.org/).

7.6.1 The Fractal Model

This section presents the main elements of the Fractal model: components, interfaces and types, bindings, control and runtime management.

Components, Interfaces, and Bindings

A Fractal component consists of two parts: a *membrane* and a *content*. The membrane provides control facilities, while the content implements the component's functionality. The Fractal model defines two kinds of components: *primitive* and *composite*. A primitive (non decomposable) component may be used as a unit of application development, or as a means of encapsulating legacy code, making its interfaces visible and allowing it to be integrated into an application together with other components. A composite component encapsulates a set of components, primitive or composite. Thus a complex application (which is a composite component) has the same structure as any of its parts, hence the name "Fractal".

A component, be it primitive or composite, has three kinds of interfaces.

- Provided (or *server*) interfaces, which describe the services supplied by the component; these are represented as T-shaped forms, with the bar on the left ($\vdash$).
- Required (or *client*) interfaces, which describe the services that the component needs for its execution; these are represented as T-shaped forms with the bar on the right ($\dashv$).
- Control interfaces, which describe the services (implemented by the membrane) available for managing the component; these are represented as T-shaped forms with the bar on the top ($\top$).

These are external interfaces, i.e., they are visible from outside the component. In addition, composite components have internal interfaces that are used to make visible some interfaces of their contained components. Such an internal interface is paired with an external interface of a complementary kind (i.e., the pairs are client-server or server-client). These interface pairs may be seen as channels across the composite component's membrane.

These notions are illustrated in the example shown on Figure 7.15, in which the membrane of a component is represented by the border frame that surrounds it, and the interfaces are represented using the notations described above. The composite component

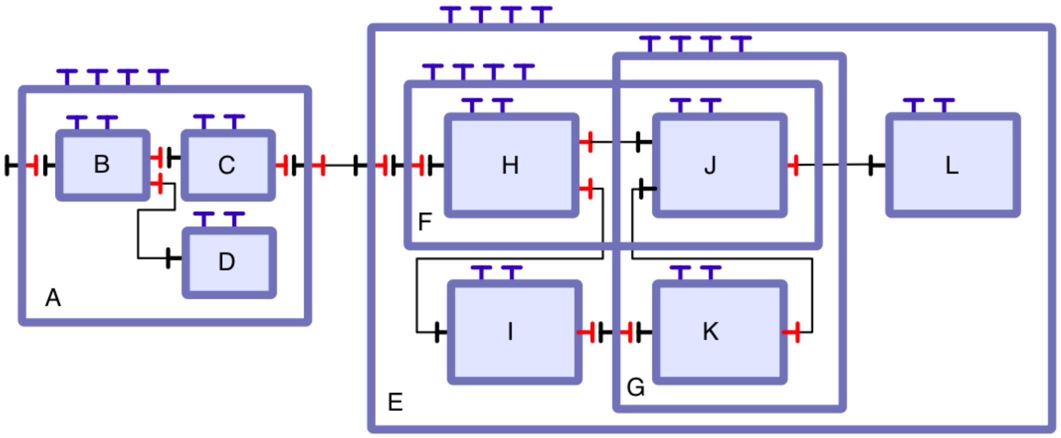


Figure 7.15. Composition and sharing in Fractal

E contains two primitive components (I and L) and two composite components (F and G). F contains H and J, G contains J and K. J is shared between F and G.

The figure shows the two main mechanisms provided by Fractal to define a complex architecture: composition and binding. Client interfaces are bound to server interfaces; also note the interface pairs used to make visible the interfaces of contained components. The bindings involving control interfaces have not been represented; control mechanisms are described later in this section.

Two mechanisms may be used for bindings (see also 3.3). If the bound interfaces are in the same address space (primitive binding), the binding is represented by the equivalent of an address (e.g., a pointer, a Java reference). If the bound interfaces are in different address spaces (composite binding), the binding is embodied in a binding object (or connector) which itself takes the form of a component. As said in 7.3.1, there is no structural difference between a connector and a (possibly composite) component. Distributed applications, involving proxies and communication protocols, can be constructed in this fashion.

An original feature of Fractal is the ability to share components between composite components. This allows the shared component's state to be itself shared between the enclosing components. The alternative to sharing would be either to replicate the component, which entails consistency maintenance, or to keep the component outside the sharing composite components, which complexifies the application's structure and defeats the purpose of encapsulation. Shared components are a convenient way of representing shared stateful resources; they are also useful when the component structure is used to represent management or protection domains (see 10.1.2). The behavior of a shared component is defined by the control part of the smallest component containing all its enclosing composite components (e.g., in the example of Figure 7.15, the behavior of J is controlled by E).

Component Control

The function of controlling a component is devoted to its membrane, which is composed of an arbitrary number of controllers, each in charge of a specific function. The membrane acts as a meta-level in a reflective architecture (2.4.1, 7.5.5), and provides introspection and intercession operations with respect to the component's content. To do so, the membrane maintains a causally connected representation of the component's content.

The Fractal model does not impose the presence of any pre-determined set of controllers, but defines several levels corresponding to an increasing degree of control.

At the lowest level, no control facilities are provided. This is the case of components that encapsulate legacy software for which no control "hooks" are available. The only mandatory interface other than the functional ones is the `Naming` interface (see 3.4.2): a component is a naming context, in which its interfaces have local names. Thus if a component has name `Comp` in some enclosing context (e.g., a directory, or a virtual machine), its interfaces may have such names as `Comp.Int1`, `Comp.Int2`, etc. This ensures name isolation, as different interfaces may have the same name in different components.

The next level of control provides external introspection capabilities, in the form of two interfaces, `Component` and `Interface`, which allow the discovery of all interfaces (both internal and external) owned by the component[7].

The upper levels of control include both introspection and intercession facilities. The most usual controllers defined by the Fractal specification are the following.

- Attribute controller. The `AttributeController` interface provides getter and setter operations to inspect and modify the component's attributes (configurable properties).
- Binding controller. The `BindingController` interface allows binding and unbinding the component's interfaces by means of primitive bindings.
- Content controller. The `ContentController` interface allows a component to inspect and to control its content, by enumerating, adding and removing subcomponents.
- Lifecycle controller. The `LifecycleController` interface allows control on a component's execution, e.g., through the start and stop operations.

In addition, since the Fractal model is open, customized controllers may be added for specific purposes. Thus, for example, components may be enhanced with autonomic management facilities (10.2), as has been done in the Jade system [Bouchenak et al. 2006] based on Fractal.

Components' Lifecycle

Lifecycle covers two aspects: component instantiation and removal, and execution control. Note that the Lifecycle controller only deals with the second aspect.

Components are created using factories. For a language-specific implementation of the Fractal platform, these factories may be directly implemented using the specific creation mechanisms of the supported language (e.g., `new` for Java, etc.). In addition, the available frameworks provide more convenient mechanisms, based on templates (a template is a specialized factory for a given component type; a template is itself created using a generic factory). This process may itself be activated by interpreting ADL constructs, and is thus

[7]This capability is similar to that provided by the `IUnknown` interface of the COM model.

made invisible to developers. Once created, the components must be bound to compose applications. This again may be done directly, using binding controllers, or by interpreting ADL sequences.

A component may be removed from the content of an enclosing component via the `removeFcSubComponent` operation of the `ContentController` interface.

Execution control is done via three operations provided by the `LifecycleController` interface: `getFcState` returns the current state of a component (stopped or started), while `startFc` and `stopFc` trigger state transitions. The semantics of these operations has been voluntarily left as weak as possible; these operations are usually extended or redefined to suit specific applications' needs.

Types and Conformance

In Fractal, interfaces and components are optionally typed. An interface type comprises the following elements.

- an identifier, which is a name valid in the context defined by the component.
- a signature, which consists of a collection of method signatures (the syntax used to describe these signatures is that of Java).
- the role, which may take one of two values: `client` (required) or `server` (provided).
- the contingency, which may take one of two values: `mandatory` or `optional`. The interpretation of the contingency depends on the role. For server interfaces, the contingency applies to the presence of the interface (i.e., a mandatory server interface *must* be provided by the component). For client interfaces, the contingency applies to the binding of the interface (i.e., a mandatory client interface *must* be bound[8]).
- the cardinality, which may take one of two values: `singleton` or `collection`. This indicates whether the interface behaves as a single interface or as a collection (i.e., creating a new instance at each binding, as explained in 7.4.3).

A component type is a collection of interface types, which describe the interfaces that components of this type may or must have at run time (depending on their contingency).

The usual notion of interface conformance, as defined in 2.1.3, applies to Fractal interfaces types, but must be extended to take contingency and cardinality into account. Interface type *IT2* conforms to Interface type *IT1* (noted $IT2 \sqsubseteq_F IT1$)[9] if any instance *it1* of *IT1* can be replaced by an instance *it2* of *IT2*. This requires the following conditions to be met.

- *it1* and *it2* have the same role (server or client)
- if the role is `server`, then (a) $IT2 \sqsubseteq IT1$; and (b) if the contingency of *it1* is `mandatory`, then the contingency of *it2* is `mandatory` too.

[8]this binding may occur at the latest at run time, but before the component is actually used.

[9]we keep the notation $\sqsubseteq$ for the usual subtyping relationship, without contingency and cardinality.

- if the role is `client`, then (a) $IT1 \sqsubseteq IT2$; and (b) if the contingency of *it1* is `optional`, then the contingency of *it2* is `optional` too.
- if the cardinality of *it1* is `collection`, the cardinality of *it2* is `collection`.

The notion of component conformance (or substitutability) follows (see 7.4.2). Let *CT1* and *CT2* be component types (collections of interface types). Then $CT2 \sqsubseteq CT1$ iff:

- each server interface of *CT1* is a super-type of a server interface type defined in *CT2*.
- each client interface of *CT2* is a subtype of a client interface defined in *CT1*.

7.6.2 Fractal Implementation Frameworks

Since most elements of the Fractal specification are optional, an implementation of this specification is best conceived as an extensible framework, whose main function is to allow the programming of component membranes.

Several Fractal implementation frameworks have been developed. They differ by the target language (implementations exist for C, C++, Java, and Smalltalk), by the techniques used, and by the additional capabilities provided. In the rest of this section, we briefly describe two such frameworks, Julia and AOKell. Other frameworks include Think [Fassino et al. 2002], designed for operating systems kernels, using the C language, and ProActive [Baduel et al. 2006], designed for active, multithreaded Java components in a grid environment.

The Julia Framework

Julia [Bruneton et al. 2006] is the reference implementation of the Fractal component model. It is based on Java and provides a set of tools for the construction of component membranes. In addition to the objects that implement the component's contents, two kinds of Java objects are used:

- Objects that implement the control part of the membrane. These include controllers, which implement the control interfaces (an arbitrary number of which may be defined), and interceptors, which may apply to incoming and outgoing method calls.
- Objects that reference the functional and control interfaces of the component. These objects are needed to allow a component to keep references to another component's interfaces.

This is illustrated by Figure 7.16. Controller objects are represented in dark gray, interceptors in light gray, and interface references in white.

Since the different control operations may be related to each other, controllers and interceptors usually contain mutual references. Controllers and interceptors are generated by factories, which take as input a description of the functional and control parts of the components.

Controllers are required to be flexible (to cater for a variety of levels of control) and extensible (to allow user-defined extensions). The construction framework must comply with

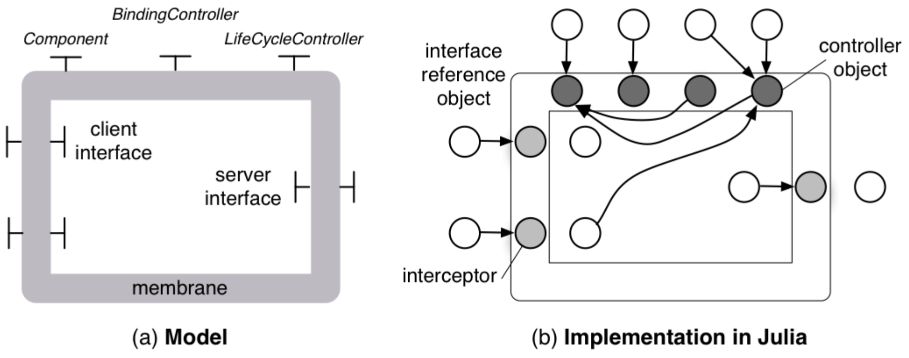


Figure 7.16. Controllers and interceptors in the Julia framework

these requirements. This is done by providing a basic implementation for each controller, which may be extended through the use of mixin classes. This method was preferred to class inheritance, which would lead to code duplication, in the absence of multiple inheritance[10]. Mixin classes allow a method of a base class to be extended by adding code fragments at specified locations, or to be overridden with a new method. Several mixins may be applied to a given base class; the order of application is significant. For efficiency, the mixed classes are dynamically generated in byte-code form.

In the Julia framework, controllers implement the generic (i.e., common to all methods) part of component control. The method-dependent parts, when needed, are implemented by interceptors (2.3.4), which insert code fragments before and/or after an incoming or outgoing method call. In a similar way to controller generation, this additional code may be dynamically inserted in byte-code form. The code insertion mechanism is open and extensible.

Julia provides various optimization mechanisms, whose effect is (a) to merge the various code pieces that make up a component's membrane into a single Java object; and (b) to shorten the chain of indirect calls (through reference objects and interceptors) that makes up an inter-component method call.

Julia has been used to develop various Fractal-based systems, one of which is the Dream communication middleware framework, described in 4.5.2. Experience reported in [Bruneton et al. 2006] shows that the additional flexibility brought by the use of configurable components is paid by a time and space overhead of the order of a few per cent.

The AOKell Framework

Like Julia, the AOKell component framework [Seinturier et al. 2006] is a complete implementation of the Fractal specification. It differs from Julia on two main points: control functions are integrated into components using aspects (2.4.2), and the controllers are themselves implemented as components. The objective of this organization is to improve the flexibility of the control part of the components, by making its structure open, explicit, and easily adaptable.

[10]Another approach would be to use aspect-oriented programming. This has been done in the AOKell framework, described further in this section.

In AOKell, the membrane of a Fractal component is a composite component enclosing a set of interacting components, each of which performs the function of a particular Fractal controller. The membrane may be seen as a meta-level layer (2.4.1), sitting above the base layer that provides the functional behavior of the component. However, this reflective structure does not extend further up: the controllers that make up the membrane are Fractal components equipped with a predefined membrane.

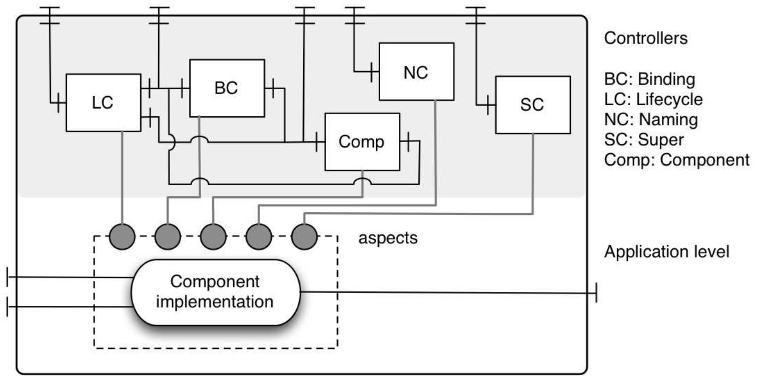


Figure 7.17. A primitive component's membrane in the AOKell framework

The controllers interact with the base level through aspects (2.4.2): each controller is in charge of a specific function, and is associated with an aspect that is used to integrate that function into the component's base layer (the component's functional part). This is done using two mechanisms provided by the AspectJ extension to Java (2.4.2):

- Feature injection. Aspects include method or field declarations, which are injected into the Java classes that implement the base level.
- Behavior extension. This technique uses the point cuts and advice code of AspectJ to insert interceptors at chosen points of the base level classes, e.g., before or after method calls.

The membrane has a different structure for primitive and for composite components (in particular, the former do not have a content controller). The overall organization of a primitive component is described on Figure 7.17.

A component's membrane may be easily extended with additional controllers. For instance, a logging controller may be added, e.g., to log method calls. This may be done for a specific component, or for a whole set of components by defining a new membrane template including the logging controller.

The overall performance of AOKell is comparable to that of Julia within a 10% margin (AOKell is more efficient with injection, less efficient with interception).

7.6.3 Fractal ADL

The Fractal Architecture Description Language (Fractal ADL) is an extensible formalism to describe the global architecture of a system built out of Fractal components. It is associated with an extensible set of tools that may perform various tasks such as code generation, deployment, analysis, etc.

The Fractal ADL and its associated tool-set are open, in the sense that they are not tied to a particular implementation language, nor to a specific step in the software's lifecycle.

The description language

A description in Fractal ADL is organized as a set of XML modules, each associated with a DTD (Document Type Definition) that defines its structure. This modular structure is designed for extensibility: to introduce a new aspect in the description, one needs to provide the corresponding DTD. Thus, in addition to the basic aspects (interfaces, bindings, attributes, containment relationships), a number of additional aspects can be introduced, such as typing, implementation properties, deployment, QoS contracts, logging, etc.

A simple example, taken from the Fractal tutorial [Fractal] gives a flavor of the form of the core Fractal ADL. The `HelloWorld` application is organized as a composite component, which contains two primitive components, `Runnable` and `Service`. `Service` exports an interface `s` used by `Runnable`, which in turn exports an interface `r` re-exported by the `HelloWorld` component. This organization is summarized on Figure 7.18.

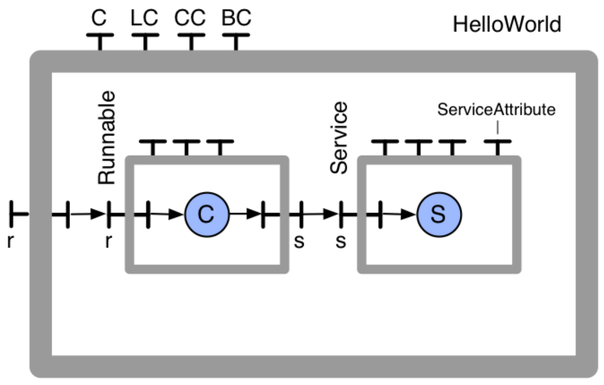


Figure 7.18. A simple Fractal application (from [Fractal])

This application is described by the following Fractal ADL fragment

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC
    "-//objectweb.org//DTD Fractal ADL 2.0//EN"
    "classpath://org/objectweb/fractal/adl/xml/basic.dtd">

<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="client" definition="ClientImpl"/>
  <component name="server" definition="ServerImpl"/>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```

This fragment specifies the `HelloWorld` component in terms of its contained components, whose definitions follow (XML headers are omitted from here on):

```
<definition name="ClientImpl">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <interface name="s" role="client" signature="Service"/>
  <content class="ClientImpl"/>
</definition>

<definition name="ServerImpl">
  <interface name="s" role="server" signature="Service"/>
  <content class="ServerImpl"/>
</definition>
```

The names of the implementation classes are the same as the names of the definitions. This is only a convention, not a necessity.

These definitions could in fact be embedded into the description of the `HelloWorld` component, which would make the overall description more concise; however, in that case, the descriptions of the contained components could not be reused.

The controller of a primitive or composite component may be specified by defining a controller descriptor, as well as an attribute controller interface, and attribute values:

```
<definition name="ServerImpl">
  <interface name="s" role="server" signature="Service"/>
  <content class="ServerImpl"/>
  <attributes signature="ServiceAttributes">
    <attribute name="header" value="->"/>
    <attribute name="count" value="1"/>
  </attributes>
  <controller desc="primitive"/>
</definition>
```

Component types may be defined, which allows using inheritance, by adding new clauses to an existing type:

```
<definition name="ClientType">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <interface name="s" role="client" signature="Service"/>
</definition>

<definition name="ClientImpl" extends="ClientType">
  <content class="ClientImpl"/>
</definition>
```

The above definition of `ClientImpl` is equivalent to that previously given. New clauses can also override existing ones.

Other aspects of the Fractal ADL may be found in the tutorial [Fractal].

The Fractal ADL tool-set

The first set of tools using the Fractal ADL are factories used to instantiate a configuration from its ADL description. Thus, using the `HelloWorld` example, the following sequence instantiates a `ClientImpl` component:

```
Factory f = FactoryFactory.getFactory();
Object c = f.newComponent("ClientImpl", null);
```

Here `Factory` and `FactoryFactory` are provided in the Fractal distribution. The second argument of the newComponent method may be used to specify additional properties. The component creation proper is done by a back-end component, which comes in several versions (using either the Java reflection package or the API provided by the Fractal framework for component creation).

Another use of the Fractal ADL is as a pivot language for various tools. In particular, the ADL serves as a support for a graphical interface that provides a graphical view of a Fractal configuration, which can be directly manipulated by the users.

As a final illustration of the association between tool-sets and architectural system descriptions, we briefly describe the design of an extensible tool-set [Leclercq et al. 2007] initially developed with Fractal ADL as a support language, but potentially usable by other ADLs. The tool-set is operated as a workflow, made up of three main components.

- The loader reads a set of input files (typically ADL descriptions) and generates an abstract syntactic tree (AST), using grammars and parsers for each input language. The AST represents the structure of a system in an extensible, language independent form. Each node represents an element of the system under description (e.g., components, interfaces, methods, etc.), and has an extensible set of interfaces (e.g., to retrieve its properties or to give access to its children).

- The organizer processes the AST to generate a graph of tasks to be executed (e.g., creating component instances, creating bindings between components, etc.).

- The scheduler determines the dependencies between the tasks in the graph, and schedules the execution of the tasks in an order that respects these dependencies.

The tool-set support the following services: verification of the architecture's correctness (e.g., correct binding of the interfaces, including type-checking); generation of "glue code" for components; generation of stubs and skeletons for distributed components; compilation of source code and instantiation of components. This range of services is easily extensible, thanks to the modular structure of the tool-set and to the extensibility of the AST format. More details may be found in [Leclercq et al. 2007].

In conclusion, the main features of the Fractal ADL are its modularity, its extensibility, and its use as a pivot format for a variety of tools. Dynamic extensions are the subject of current research.

7.7 Case Study 2: OSGi, a Dynamic, Component-Based, Service Platform

The OSGi Alliance [OSGI Alliance 2005], founded in 1999, is an independent non-profit corporation that groups vendors and users of networked services. Its goal is to develop specifications and reference implementations for a platform for interoperable, component-based, applications and services.

The OSGi specification defines both a component model and a run time framework, targeted at Java applications ranging from high-end servers to mobile and embedded devices. OSGi components are called bundles. A *bundle* is a modular unit composed of Java classes, which exports services, and may import services from other bundles running on the same Java Virtual machine (JVM). A *service* is typically implemented by a Java class provided by a bundle, and is accessible through one or several interfaces. In addition, a service may be associated with a set of *properties*, which allow services to be dynamically published and searched, using a service discovery service (3.2.3) provided by the framework.

In the rest of this section, we briefly summarize the main aspects of the OSGi specification, and we show how it is used to develop service-based applications. We conclude with a review of some implementations and extensions.

7.7.1 The OSGi Component Model

An OSGi bundle is a unit of packaging and deployment, composed of Java classes and other resources such as configuration files, images, or native (i.e., processor-dependent) libraries. A bundle is organized as a Java Archive (JAR) file[11], containing Java classes and other resources, together with a manifest file, which describes the contents of the JAR file and provides information about the bundle's dependencies (i.e., the resources needed for running the bundle).

The unit of code sharing between bundles is a Java package. A bundle can export and import packages[12]. Exported packages are made available to other bundles, while imported packages are taken from those exported by other bundles. If the same package is exported by several bundles, a single instance is selected (the one exported by the bundle with the highest version number, and the oldest installation date). Packages that are neither imported nor exported are local to the bundle.

Package sharing between bundles takes place within a JVM and relies on the class loading mechanism of Java. Each bundle has a single class loader. The class space of a bundle is the set of classes visible from its class loader, through class loading delegation links; it includes, among others, the imported packages and the required bundles specified by the bundle. Thus the OSGi framework supports multiple class spaces, which allows multiple versions of the same class to be in use at the same time.

[11]Since release 4, a number of *fragment bundles* may be attached to a *host bundle*, thus allowing a bundle to be conditioned in the form of several JARs; fragment bundles are mainly used for packaging processor- or system-dependent resources.

[12]In addition, a bundle may *require* another bundle. This mechanism provides a stronger degree of coupling than package sharing. It may seem convenient, but it leads to confusing situations in the case of multiply exported packages.

In terms of the Fractal component model, OSGI bundles may be seen as having the equivalent of a binding controller and a lifecycle controller. A bundle's lifecycle is represented on Figure 7.19. The states and transitions are summarized as follows.

Installation Installation is the process of loading the JAR file that represents the bundle into the framework. The bundle must be valid, i.e., its contents must be consistent and error-free. A unique identifier is returned for the installed bundle. The installation process is both persistent (the bundle remains in the INSTALLED state until explicitly un-installed) and atomic (if the installation fails, the framework remains in the state in which it was prior to this operation).

Resolution An installed bundle may enter the RESOLVED state when all its dependencies, as specified in its manifest, are satisfied. Resolution is a binding process, in which each package declared as imported is bound (or "wired", in the OSGi terminology) to an exported package of a RESOLVED bundle, while respecting specified constraints. These constraints take the form of matching attributes, e.g., version number range, symbolic name, etc. Resolution of a bundle is delayed until the last possible moment, i.e., until another bundle requires a package exported by that bundle.

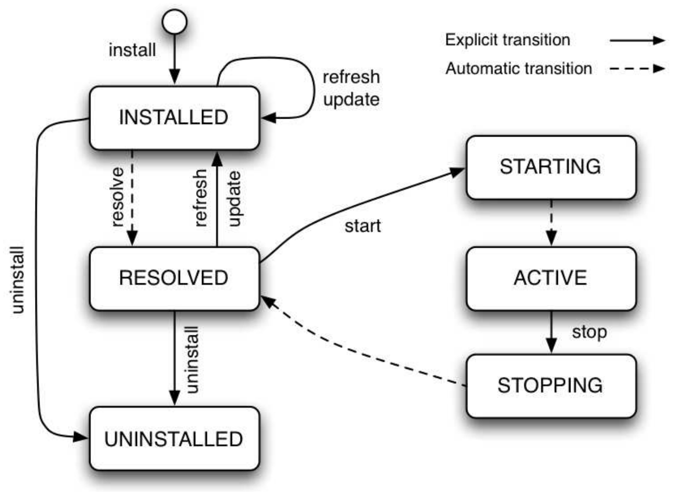


Figure 7.19. The lifecycle of an OSGI bundle (from [OSGI Alliance 2005])

Imported packages may be specified as mandatory (the default case), optional, or dynamic. Mandatory imports must be satisfied during the resolution process. Optional imports do not prevent resolution if no matching export is found. Dynamic imports may wait until execution time to be resolved.

After a bundle is resolved, the framework creates a class loader for the bundle.

Activation Once in the RESOLVED state, a bundle may be activated. The bundle's manifest must specify a `Bundle-Activator` class, which implements the predefined `BundleActivator` interface. This interface includes two methods, `start` and `stop`. When a bundle is activated, the framework creates an instance of `Bundle-Activator` and invokes the `start(BundleContext)` method on this instance, where `BundleContext` represents the current context of the bundle. The context is an object that contains the API of the OSGi framework, as well as specific parameters. The `start` method may look for required services (see 7.7.2), register services, and start new threads if needed. If the method succeeds, the bundle enters the ACTIVE state. If it fails, the bundle returns to the RESOLVED state. The STARTING state is the transient state of the bundle during the execution of the `start(BundleContext)` method.

Stopping An active bundle may be stopped by invoking the `stop(BundleContext)` method of its `Bundle-Activator` class instance. This method must unregister the provided services (see 7.7.2), release any required services, and stop all active threads. The bundle returns to the RESOLVED state. The bundle is in the transient STOPPING state while the `stop` method is executing.

Update A bundle may be updated (i.e., modified to produce a new version). If the bundle is active, it is first stopped. The framework then attempts to reinstall and to resolve the bundle. Since the update may change the list of imported and exported packages, resolution may fail. In that case, the bundle returns to the INSTALLED state and its class loader is removed.

Un-installation Un-installing a bundle moves it to the UNINSTALLED state (a terminal state, from which no further transition is possible). The JAR files of the bundle are removed from the local file system. If the bundle was in the RESOLVED state, those of its classes that were imported remain present as long as the importing bundles are in the ACTIVE state. The `refresh` operation restores a new consistent system after some bundles have been un-installed: it stops, resolves again, and restarts the affected (dependent) bundles.

7.7.2 Dynamic Service Management in OSGi

OSGi implements a service-oriented architecture [Bieber and Carpenter 2002], based on the modular infrastructure provided by bundles. A *service* (2.1) is accessible through one or more Java interface(s), and implemented by a Java class exported by a bundle. The OSGi service architecture is dynamic, i.e., services may appear or disappear at any time. An event-based cooperation model allows clients of services to be aware of this dynamic behavior in order to keep applications running.

Service management is based on a (centralized) service registry provided by the OSGi framework, together with an API allowing users to register and to discover services. To facilitate discovery, a service is registered with one or more interface name(s) and a set of properties. Properties have the form of key-value pairs, allowing the use of LDAP-style [LDAP 2006] filter predicates. Following the service usage pattern described in 3.3.4, a client bundle can discover a service, get a reference for it, bind to the service, and invoke the methods specified by the service interface. When the client stops using the service, it should release it, which decrements the service reference count. When the count reaches zero, the providing bundle may be be stopped, and unreferenced objects may be garbage-collected.

Events are triggered when a service is registered or unregistered, and when properties of a registered service are modified. The OSGi framework provides interfaces to observe these events and to react to them, both in a synchronous and an asynchronous mode. In addition, the framework provides a `ServiceTracker` utility class, which allows a client to subscribe to events related to a specific service or set of services, and to activate appropriate handlers for these events.

The Service Component Runtime (SCR) defines a component model that facilitates the management of an application as a set of interdependent services. In this model, a declarative description, in the XML format, is associated with each bundle that provides or uses a service, and lists the dependencies of that service, i.e., the mandatory or optional services needed for its operation, and the callback handlers associated with various events. This description is used by the runtime to automatically manage the execution of the application, by activating the appropriate service registering and unregistering operations, which in turn trigger the corresponding callbacks.

7.7.3 OSGi Standard Services, Implementations and Extensions

A number of standard services have been specified by the OSGi Alliance. Some of them are generic, while others address a specific application domain. Examples of generic services are the HTTP service, which allows OSGi bundles to publish an application including both static and dynamic web pages; the Wire Admin service, which allows building sensor-based services, linking producers of measurement data to consumers; and the Universal Plug and Play (UPnP) Device Driver, which is used to develop UPnP-based applications [UPnP] over an OSGI platform.

The OSGi specifications have been implemented by several commercial vendors. Several open-source implementations are also available: Knopflerfish [Knopflerfish 2007], Oscar [Oscar 2005], Felix [Felix 2007]. On the server side, the OSGi standard has also been adopted by the Eclipse project [Gruber et al. 2005], and by several projects of the Apache Foundation.

The current OSGI specification defines a centralized framework, in which the bundles that compose an application are loaded in a single JVM. We briefly describe R-OSGi [Rellermeyer et al. 2007], an experimental extension of OSGi to a distributed environment.

R-OSGI is an infrastructure for building distributed OSGi applications. Services are units of distribution, and interact through proxies (2.3.1). Proxies allow remote interactions between OSGi frameworks located at different sites; they have two main uses: remote service invocation and event forwarding.

Service distribution in R-OSGi is transparent, i.e., a remote service is functionally equivalent to a local service (except of course for distribution-aware functions such as system management). Transparency is achieved using the following techniques:

- Dynamic proxy generation at bind time (i.e. when a client needs to bind to a remote service). The proxy is generated, as usual, from the service's interface description.

- Providing a distributed Service Registry. When a service allowing remote access is registered in a local framework, R-OSGi registers it with a remote service discovery layer, which is based on SLP (3.2.3).

- Mapping network and remote failures to local events (such as service removal), which applications running on a centralized OSGi framework are already prepared to handle.

- Using type injection to resolve distributed type systems dependencies. The problem arises when the interface of a remote service contain parameter types that are not available at the client site (because they do note belong to the standard Java classes). Types present in interfaces and present in exported bundles are collected when a remote service is registered, and transmitted to the client at proxy generation time.

The performance of R-OSGi for remote service invocations has been found slightly better than that of Java RMI (5.4).

7.8 Historical Note

As recalled in the Introduction, the vision of software being produced by assembling "off the shelf" software components [McIlroy 1968] dates from the early years of software engineering. This vision was first embodied in the notion of a *module* as a unit of composition; a module is a piece of program that can be *independently* designed and implemented, in the sense that it only relies on the specification of the modules it uses, not on their implementation. Seminal papers in this area are [Parnas 1972], which identifies decomposition criteria based on the "information hiding" principle (a form of encapsulation), and [DeRemer and Kron 1976], which introduces a first approach to software architecture description in the form of a Module Interconnection Language (MIL). A few programming languages include a notion of module (e.g., Mesa [Mitchell et al. 1978], Modula 2 [Wirth 1985], Ada [Ichbiah et al. 1986]), but none provides any construct for expressing global composition. Several experimental MILs were proposed (e.g., Conic [Magee et al. 1989], Jasmine [Marzullo and Wiebe 1987]), but none has achieved widespread use.

A further step in the definition of composition units was the advent of *objects*, which introduced the notions of classes and instances, inheritance and polymorphism. Early object-oriented languages are Simula 67 [Dahl et al. 1968] and Smalltalk 80 [Goldberg and Robson 1983]. However, objects did not bring new concepts in global architectural description. Distributed objects (see a brief historical review in Section 1.5) provided building blocks for the first middleware products, which appeared in the early 1990s.

Architectural concerns were revived in the early and mid-1990s (see [Shaw and Garlan 1996]), with further progress in the understanding of the notions of *component*, *connector*, and *configuration*. Based on these notions, the concept of an Architecture Description Language (ADL) was proposed as a way of expressing global composition. A number of proposals were put forward, both for component models (e.g., Fractal [Bruneton et al. 2006], Koala [van Ommering et al. 2000]) and for ADLs (e.g., Darwin [Magee et al. 1995], Wright [Allen 1997], Rapide [Luckham and Vera 1995], Acme [Garlan et al. 2000]). An attempt towards unifying global architectural description with component implementation is ArchJava [Aldrich et al. 2002], an extension of the Java programming language. An attempt towards a taxonomy of component models may be found in [Lau and Wang 2007].

Research on the theoretical foundations of component models has been going on since the 1990s, but no universally accepted formal model has yet been proposed. A collection of articles describing early work may be found in [Leavens and Sitaraman 2000]. More recent work is presented in [Lau et al. 2006], [Bidinger and Stefani 2003] (the latter gives a formal base for the Fractal component model (7.6)).

Component-based commercial platforms (JEE [J2EE], .NET [.NET]) appeared in the late 1990s and early 2000s, introducing components in the world of applications. Several container-based frameworks relying on inversion of control (7.5.2) have also been developed (e.g., Spring [Spring 2006], Excalibur [Excalibur], PicoContainer[PicoContainer]). Contrary to JEE and .NET, these frameworks do not assume a specific component format, and directly support Java objects[13].

In the same period, several groups were investigating the use of “component toolkits” to build configurable operating systems kernels. Projects in this area include OSKit/Knit [Ford et al. 1997, Reid et al. 2000], Pebble [Gabber et al. 1999], Think [Fassino et al. 2002], CAmkES [Kuz et al. 2007]. The growing use of embedded systems should foster further proposals in this area.

The development of Service Oriented Architectures (SOA, see 2.1) puts emphasis on applications built out of dynamically evolving, loosely coupled elements. The OSGi service platform (7.7 and [OSGI Alliance 2005]), initially targeted at small devices, allows dynamic service composition and deployment on a wide range of infrastructures. Recent research on ADLs concentrates on extensible ADLs (e.g., xADL [Dashofy et al. 2005], the Fractal ADL (7.6.3)), and on dynamic aspects of ADLs.

In the late 1990s, it was recognized that the areas of software architecture, configuration management, and configurable distributed systems were based on a common ground [van der Hoek et al. 1998]. Configuration management (see 10.4) is emerging as a key issue, and a respectable subject for research. The advent of mobile and ubiquitous systems emphasizes the need for dynamic reconfiguration capabilities (see 10.5). A better understanding of the fundamental concepts of composition is needed to make deployment and (re)configuration reliable and safe. Active research is ongoing is this area, but its impact on current industrial practice is still limited.

[13]often referred to as POJOs, Plain Old Java Objects.

Chapter 8

Persistence

This chapter is still unavailable.

Chapter 9

Transactions

Transactions are a mechanism that allows endowing a sequence of actions with the properties of a single atomic (indivisible) action. Originally introduced in the context of database management systems, transactions are a powerful tool to help building complex applications, by ensuring that the system remains in a consistent state in all circumstances.

In this chapter, we examine how transactional mechanisms are being implemented and used in middleware systems. After a general introduction, the chapter briefly reviews the main concepts and techniques developed for implementing transactions, with special emphasis on distributed transactions. It goes on with a discussion of transactional frameworks for middleware systems, and concludes with a case study.

9.1 Motivations and Main Concepts

The notion of an *atomic action* is a powerful tool for overcoming the difficulties of concurrent programming. To illustrate this point, consider the case of multithreaded programs in a shared memory multiprocessor. Elementary hardware-implemented operations such as atomic exchange (exchanging the contents of a processor register and a memory location in a single, indivisible action) are the basic building blocks on which more elaborate synchronization mechanisms can be constructed and made available to application developers.

Elementary atomic operations have simple properties, which facilitate reasoning and allow ensuring that the mechanisms built upon these operations meet their specifications. For instance, the concurrent execution of two atomic operations A and B may only have two outcomes (in the absence of failures): it has the same effect as either A followed by B or B followed by A.

A composite action is defined as a sequence of actions; ultimately, any action consists of a sequence of elementary atomic operations.The main idea of transactions is to extend the properties of these elementary operations to composite actions.

A few notions are needed to specify the effect of actions. The universe (also called "the system") is composed of objects (in the sense of 2.2.2), each of which has a well-defined state (typically, the values of a set of variables). The global state of the universe is the union of the states of the individual objects. A process is a sequence of actions, each of which is defined as follows: given an initial state of the system and a (possibly void) input value, the effect of the action, when executed alone, is to bring the system in a final state in finite time, and to deliver a (possibly void) output value[1]. For a given action, the final state and the output value are specified functions of the initial state and the input value. This is essentially a state machine model, which we also use in 11.3.1.

Process execution may be subject to failures. A *failure* occurs when the behavior of the system (or some component of it) does not conform to its specification. Different classes of failures, according to their degree of severity, have been identified (see 11.1.3 for a detailed discussion). Here we only consider *fail-stop* failures: either the component works according to its specification, or it does nothing. We assume the existence of *stable storage*, i.e., storage that guarantees that the data committed to it are immune from failures[2]. Techniques for the implementation of stable storage are described in [Lampson and Sturgis 1979].

We now come to the notion of *atomicity*, a property of elementary actions that we wish to extend to composite actions. Atomicity has two facets.

- *Concurrency atomicity.* Consider concurrent actions, i.e., actions executed by concurrent processes. Concurrency atomicity is then expressed as follows: let A and B be two actions that are executed concurrently (which we denote by $A||B$). Then, in the absence of failures, the effect of this execution is either that of $A;B$ or that of $B;A$ (the effect is defined by the output values and the change of state of the system).

  A different formulation of this property is as follows. A composite action goes through intermediate states of the system (the final states of its composing actions). No such intermediate state may be visible to any concurrently executing action. Stated in this form, concurrency atomicity is called *isolation*.

- *Failure atomicity.* Consider an action A, which (when executed alone) makes the system go from state $S1$ to state $S2$ (which we denote by $S1\ \{A\}\ S2$). Then, if a failure may occur during the execution of A, the final state of the system is either $S1$ or $S2$. In other terms, the action is performed either entirely or not at all[3]. Again, no intermediate state may be visible.

  In addition to failures provoked by external causes, an action may decide to interrupt its execution, and to cancel all the changes made so far to the system state. This operation is called *abort*. If $S1\ \{A\}\ S2$, the effect of an abort in the execution of A is to restore the state of the system to $S1$, an operation called *rollback*.

Thus the atomicity property allows us to limit the uncertainty about the outcome of a possibly concurrent execution of a set of actions in the presence of failures. A simple form of transaction is defined as a sequence of actions that has the atomicity property (both for concurrency and failures).

Transactions have been historically introduced in the context of database management systems, which motivated the introduction of two additional properties.

[1]Two remarks are in order: (a) the effect of an action may be to create new objects; and (b) output delivery may take the form of an action in the "real world" (e.g., printing a form, or delivering cash).

[2]We exclude *catastrophes*, exceptional events that would cause an extensive loss of storage media.

[3]"Real actions", i.e., output actions operating in the real world (see note 1), pose a problem here, since their effect can not always be canceled. More on this issue in 9.3.

- *Consistency.* Consistency in a database is usually expressed as a set of *integrity constraints*, i.e., predicates that apply to its state. These constraints are meant to express invariant properties of the real world entities that are represented in the database (e.g., banking accounts, inventory, contents of a library, etc.). Since consistency needs to be preserved through the life of the database, a transaction is defined as an atomic (composite) action that brings the system from a consistent state to another consistent state. During the execution of a transaction, some intermediate states may be inconsistent; however, by virtue of atomicity, they are never visible outside the transaction. By definition, consistency is an application dependent notion.

- *Durability.* The durability (or permanence) property states that, if the transaction succeeds (i.e., if it concludes without failures and does not abort), the changes it made to the database are permanent. Of course, the state may be modified by subsequent transactions. The operation that marks the successful end of a transaction is called *commit*. Durability means that, after a transaction has committed, the changes that it made are immune to failures (catastrophes excluded).

The acronym ACID (Atomicity, Consistency, Isolation, Durability) is commonly used to qualify transactions in a database environment (note that "atomicity" in ACID has the restricted meaning of "failure atomicity", as "isolation" stands for "concurrency atomicity"). Thus if a database is initially in a consistent state, and all operations on it are included in (consistency preserving) transactions, it is guaranteed to remain in a consistent state, and all outputs are likewise consistent.

Transactions favor separation of concerns (1.4.2), since they allow the application developers to concentrate on the logic on the application (i.e., writing transactions that preserve consistency), while a *transaction management system* is responsible for ensuring the other transactional properties (atomicity, isolation and durability). These properties are guaranteed through the following methods.

- Concurrency atomicity (or isolation) is ensured by *concurrency control*, which uses locking (the main technique) or time-stamping.

- Failure atomicity is ensured by *recovery*, which in turn is based on logging techniques.

- Durability depends on failure atomicity, and relies on *stable storage*.

The transaction management system provides a simple interface to its users. While the syntax of the operations may vary according to the system, the basic interface is usually composed of the following operations.

- *begin_transaction*: start of a new transaction.

- *commit_transaction*: normal end of the transaction (changes are durably registered).

- *abort_transaction* (or *rollback_transaction*): discontinue the execution of the transaction and cancel the changes made by the transaction since its beginning.

A transaction is uniquely identified, usually through an identifier delivered by the transaction management system at the start of the transaction. In some systems, a transaction is associated with an object, which provides the above operations in the form of methods. Additional operations may be available, e.g., consulting the current state of the transaction, etc.

The above operations achieve *transaction demarcation*, i.e., delimiting the operations that are part of a transaction within the program being executed. Transaction demarcation may be explicitly done by the application programmer, or may be automatically generated from a declarative statement by the programming environment (more details in 9.6).

The topic of transactions is an important area in its own right, and is the subject of a vast amount of literature, including several reference books [Bernstein et al. 1987, Gray and Reuter 1993, Weikum and Vossen 2002, Besancenot et al. 1997]. In the context of this book, we concentrate on transactional aspects of middleware; our presentation of the fundamental aspects of transaction management is only intended to provide the necessary context, not to cover the subject in any depth.

The rest of this chapter is organized as follows. Section 9.2 examines concurrency control. Section 9.3 is devoted to recovery techniques. Distributed transactions are the subject of Section 9.4. Section 9.5 presents nested transactions and relaxed transactional models. Transactional middleware is introduced in Section 9.6, and illustrated by a case study in Section 9.7. A brief historical note (9.8) concludes the chapter.

9.2 Concurrency Control

If transactions were executed one at a time, consistency would be maintained, since each individual transaction, by definition, preserves consistency. However, one wishes to allow multiple transactions to execute concurrently, in order to improve performance, through the following effects.

- Concurrent execution allows taking advantage of multiprocessing facilities, both for process execution and for input-output.
- Concurrent execution potentially allows a better usage of resources, both physical and logical, if each transaction only needs a subset of the available resources.

The goal of *concurrency control* is to ensure isolation (or concurrency atomicity, as defined in 9.1) if several transactions are allowed to execute concurrently on the same set of data. The main concept of concurrency control is *serializability*, a property of a set of transactions that we examine in 9.2.1. The main mechanism used to enforce serializability is *locking*, which is the subject of 9.2.2. In this section, we assume that all transactions take place on a single site, and that they actually commit. Aborts and failures are examined in section 9.3, and distributed transactions are the subject of section 9.4.

9.2.1 Serializability

The theory of concurrency control has developed into a vast body of knowledge (see e.g., [Bernstein et al. 1987, Papadimitriou 1986, Weikum and Vossen 2002]). Here we only present a few results that are widely used in current practice.

An acceptable (or "correct") concurrent execution of a set of transactions is one that ensures isolation, and thus preserves consistency. The first issue in concurrency control is to find a simple characterization of correct executions. A widely accepted notion of correctness is that the concurrent execution be equivalent to *some* serial execution of this same set of transactions[4]. Here, "equivalent to" means "having the same effect as"; a more precise definition is introduced later in this section. Note that we do not specify a particular order for the serial execution: any order is acceptable. If each transaction preserves consistency, so does any serial execution of the set of transactions.

Consider a set of transactions $\{T_i\}$, operating on a set of objects $\{a, b, \ldots\}$. When executed alone, each transaction is a sequence of read and write operations on the objects (each of these individual operations is atomic). We do not consider object creation and deletion; these are discussed in 9.2.3. When the transactions $\{T_i\}$ are executed concurrently, the operations of the transactions are interleaved in a single sequence, in which the order of the operations of each individual transaction is preserved. Such a sequence is called a *schedule*. By our definition of correctness, an acceptable schedule (one generated by a correct concurrent execution of the set $\{T_i\}$) is one equivalent to a *serial schedule*, i.e., one corresponding to a serial execution of the transactions of $\{T_i\}$, in some (unspecified) order. A schedule that is equivalent to a serial schedule is said to be *serializable*.

In order to characterize serializable schedules, let us identify the situations in which consistency may be violated. A read (resp. write) operation performed by transaction T_i on object x is denoted $R_i(x)$ (resp. $W_i(x)$). Consistency may be violated when two transactions T_i and T_j ($i \neq j$) execute operations on a shared object x, and at least one of these operations is a write. Such a situation is called a *conflict*. Three types of conflicts may occur, characterized by the following patterns in the execution schedule.

- Unrepeatable read: $R_i(x) \ldots W_j(x) \ldots$. Then a subsequent read of x by T_i may deliver a different result from that delivered in a serial execution.
- Dirty read: $W_i(x) \ldots R_j(x) \ldots$. Then the result of $R_j(x)$ may be different from that delivered in a serial execution.
- Lost write: $W_i(x) \ldots W_j(x) \ldots$. Then the effect of the first write may be canceled by that of the second write.

Note that a conflict does not necessarily cause an inconsistency (this depends on whether T_i commits before or after the operation by T_j). [Berenson et al. 1995] thoroughly analyze various cases of anomalous behavior, including the three above situations.

By definition, for a given execution, transaction T_j *depends on* transaction T_i if, for some object x, there is a conflict between T_i and T_j, and the operation executed by T_i on x appears first in the schedule. This relation is captured by a *dependency graph*, in which vertices represent transactions, and edges (labeled by object names) represent conflicts. Edges are oriented by the order of conflicting operations (e.g., for any of the conflicts illustrated above, the edge would be oriented from T_i to T_j).

[4]Two remarks are in order: (1) This definition only provides a sufficient condition for correctness. In other words, some consistency-preserving executions may not be equivalent to a serial execution. (2) For efficiency reasons, weaker correctness conditions have been proposed. See an example (snapshot isolation) in 9.2.3.

We may now define a form of equivalence for schedules: two schedules are *conflict-equivalent* if they contain the same operations and if they have the same dependency graph (i.e., if each pair of conflicting operations appears in the same order in both schedules). A schedule is conflict-serializable if it is conflict-equivalent to a serial schedule[5].

The following result has been proven [Papadimitriou 1979]: *a schedule is conflict-serializable if and only if its dependency graph is acyclic.* This property gives a convenient characterization of a class of consistency-preserving schedules, and is the base of the most common concurrency control algorithms used in practice. A simple example follows.

Example 1. Consider two transactions T_1 and T_2, defined as follows:

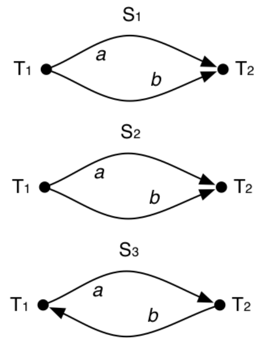
Transaction T_1	Transaction T_2
begin	
if $(A \geq X)$	*begin*
$\{A = A - X\}$	$A = (1+R)A$
else *abort*	$B = (1+R)B$
$B = B + X$	*commit*
commit	

A and B denote banking accounts (and the corresponding balances). Transaction T_1 transfers an amount X from A to B. Transaction T_2 applies an interest rate R to both accounts A and B. In fact, T_1 and T_2 are patterns for transaction types, in which A, B, X and R are parameters.

The integrity constraints are the following: (a) the balance of each account is non-negative; (b) the sum of all accounts is invariant after the execution of a transaction of type T_1; and (c) the sum of all accounts involved in a transaction of type T_2 is multiplied by $(1+R)$ after the execution of the transaction.

Now consider the concurrent execution of two transactions T_1 and T_2, with parameters a, b, x, r, and assume that, initially, all parameters are positive and $a \geq x$ (so that T_1 does not abort). Three possible schedules of this execution are shown below, together with their dependency graphs.

Schedule S_1	Schedule S_2	Schedule S_3
$begin_1$	$begin_1$	$begin_1$
$a = a - x$	$a = a - x$	$a = a - x$
$b = b + x$	$begin_2$	$begin_2$
$commit_1$	$a = (1+r)a$	$a = (1+r)a$
$begin_2$	$b = b + x$	$b = (1+r)b$
$a = (1+r)a$	$commit_1$	$commit_2$
$b = (1+r)b$	$b = (1+r)b$	$b = b + x$
$commit_2$	$commit_2$	$commit_1$



Schedule S_1 is serial $(T_1; T_2)$. S_2 derives from S_1 by exchanging two write operations on different objects; since this does not cause a new conflict, S_2 is equivalent to S_1

[5]Note that conflict-equivalence is a restricted notion of equivalence, introduced because of its easy characterization. Thus a schedule may be serializable, but not conflict-serializable. However, the two notions coincide if a transaction always reads an object before writing it, a frequent situation in practice [Stearns et al. 1976].

and thus is serializable. S_3 has a write-write conflict causing the first write on b to be lost, violating consistency. The dependency graph of S_3 contains a cycle.

We now have a simple characterization of serializable transactions. A mechanism that allows enforcing serializability is introduced in the next section.

9.2.2 Locking

In order to enforce serializability of concurrent transactions, one needs a mechanism to delay the progress of a transaction. Locking is the most frequently used technique. In order to allow concurrent access to a shared object in read-only mode, two types of locks are defined: exclusive and shared (this distinction is inspired by the canonical synchronization scheme of readers and writers [Courtois et al. 1971]). To acquire a lock on object o, the primitives are $lock\_exclusive(o)$ and $lock\_shared(o)$. An acquired lock is released by $unlock(o)$. The following rules apply for access to shared objects within a transaction.

- A transaction needs to acquire a lock on a shared object before using it (shared lock for read-only access, exclusive lock for read-write access).
- A transaction may not lock an object on which it already holds a lock, except to upgrade a lock from shared to exclusive.
- No object should remain locked after the transaction concludes, through either commit or abort.

The first two rules directly derive from the readers-writers scheme. The last rule avoids spurious locks, which may prevent further access to the locked objects. A *well-formed* transaction is one that respects these rules.

Locks are used to constrain the allowable schedules of concurrent transactions. Two rules are defined to this effect.

- An object locked in shared mode by a transaction may not be locked in exclusive mode by another transaction.
- An object locked in exclusive mode by a transaction may not be locked in any mode by another transaction.

A *legal* schedule is one that respects these rules. To ensure legality, locking operation that would break these rules is delayed until all conflicting locks are released.

Two-Phase Locking

While legal schedules of well-formed transactions ensure the valid use and correct operation of locks, all such schedules are not guaranteed to be serializable. Additional constraints are needed. The most commonly used constraint, called *two-phase locking* (2PL), is expressed as follows.

After a transaction has released a lock, it may not obtain any additional locks.

A transaction that follows this rule therefore has two successive phases: in the first phase (growing), it acquires locks; in the second phase (shrinking), it releases locks. The point preceding the release of the first lock is called the maximum locking point.

The usefulness of 2PL derives from the following result [Eswaran et al. 1976].

> Any legal schedule S of the execution of a set $\{T_i\}$ of well-formed 2PL transactions is conflict-equivalent to a serial schedule. The order of the transactions in this serial schedule is that of their maximum locking points in S.

Conversely, if some transactions of $\{T_i\}$ are not well formed or 2PL, the concurrent execution of $\{T_i\}$ may produce schedules that are not conflict-equivalent to any serial schedule (and may thus violate consistency).

Example 2. Consider again the two transactions T_1 and T_2 of Example 1. Here are two possible implementations of T_1 (T_{1a} and T_{1b}), and one implementation of T_2, using locks (for simplicity, we have omitted the test on the condition $x \leq a$, assuming the condition to be always true).

Transaction T_{1a}	Transaction T_{1b}	Transaction T_2
$begin_{1a}$	$begin_{1b}$	$begin_2$
$lock\_exclusive_{1a}(a)$	$lock\_exclusive_{1b}(a)$	$lock\_exclusive_2(a)$
$a = a - x$	$a = a - x$	$a = (1 + r)a$
$unlock_{1a}(a)$	$lock\_exclusive_{1b}(b)$	$lock\_exclusive_2(b)$
$lock\_exclusive_{1a}(b)$	$unlock_{1b}(a)$	$b = (1 + r)b$
$b = b + x$	$b = b + x$	$unlock_2(b)$
$unlock_{1a}(b)$	$unlock_{1b}(b)$	$unlock_2(a)$
$commit_{1a}$	$commit_{1b}$	$commit_2$

T_{1b} and T_2 are 2PL, while T_{1a} is not.

Here are two possible (legal) schedules S_a and S_b, respectively generated by the concurrent execution of T_{1a}, T_2 and T_{1b}, T_2.

$S_a(T_{1a}, T_2)$	$S_b(T_{1b}, T_2)$
$begin_{1a}$	$begin_{1b}$
$lock\_exclusive_{1a}(a)$	$lock\_exclusive_{1b}(a)$
$a = a - x$	$a = a - x$
$unlock_{1a}(a)$	$lock\_exclusive_{1b}(b)$
$begin_2$	$unlock_{1b}(a)$
$lock\_exclusive_2(a)$	$begin_2$
$a = (1 + r)a$	$lock\_exclusive_2(a)$
$lock\_exclusive_2(b)$	$a = (1 + r)a$
$b = (1 + r)b$	$b = b + x$
$unlock_2(b)$	$unlock_{1b}(b)$
$lock\_exclusive_{1a}(b)$	$lock\_exclusive_2(b)$
$b = b + x$	$b = (1 + r)b$
$unlock_2(a)$	$commit_{1b}$
$commit_2$	$unlock_2(b)$
$unlock_{1a}(b)$	$unlock_2(a)$
$commit_{1a}$	$commit_2$

Schedule S_a violates consistency, since T_2 operates on an inconsistent value of b. Schedule S_b is equivalent to the serial schedule $T_{1b}; T_2$

There are two variants of two-phase locking. In *non-strict 2PL*, a lock on an object is released as soon as possible, i.e., when it is no longer needed. In *strict 2PL*, all locks are released at the end of the transaction, i.e., after commit. The motivation for non-strict 2PL is to make the objects available to other transactions as soon as possible, and thus to increase parallelism. However, a transaction may abort or fail before having released all its locks. In that case, an uncommitted value may have been read by another transaction, thus compromising isolation, unless the second transaction aborts. Strict 2PL eliminates such situations, and makes recovery easier after a failure (see 9.3.2). In practice, the vast majority of systems use strict 2PL.

Deadlock Avoidance

Using locks raises the risk of *deadlock*, a situation of circular wait for shared resources. For example, transaction T_1 holds an exclusive lock on object a, and attempts to lock object b; however, b is already locked in exclusive mode by transaction T_2, which itself tries to lock object a. More generally, one defines a (directed) *wait-for graph* as follows: vertices are labeled by transactions, and there exists an edge from T_1 to T_2 if T_1 attempts to lock an object already locked by T_2 in a conflicting mode. A cycle in the wait-for graph is the symptom of a deadlock.

There are two approaches to deadlock avoidance, *prevention* and *detection*. A common prevention method is that all transactions should lock the objects in the same order. However, this is not always feasible, since in many applications transactions are independent and do not have an advance knowledge of the objects they need to lock. In addition, imposing a locking order (independent from the transaction's semantics) may be overly restrictive, and may thus reduce concurrency.

A more flexible approach, dynamic prevention, has been proposed by [Rosenkrantz et al. 1978]. At creation time, each transaction receives a time-stamp, which defines a total order among transactions. Conflicts are resolved using a uniform precedence rule based on this order. Consider two transactions T_1 and T_2, which both request a shared object x in conflicting modes, and suppose T_1 has acquired a lock on x. Two techniques may be used to resolve the conflict:

- Wait-Die: if T_2 is "older" than T_1, it waits until the lock is released; otherwise, T_2 is forced to abort, and restarts later on.
- Wound-Wait: if T_2 is "older" than T_1, it preempts object x (breaking the lock); otherwise, it waits until the lock is released. After preemption, T_1 must wait for T_2 to commit or to abort before trying to re-acquire a lock on object x.

Thus, a wait-for edge is only allowed from an older to a younger transaction[6] (in wait-die) or in the opposite direction (in wound-wait). This avoids starvation (waiting indefinitely to acquire a lock). Wound-wait is usually preferable since it avoids aborting transactions.

Contrary to prevention, deadlock detection imposes no overhead when deadlocks do not occur, and is often preferred for performance reasons. Deadlock detection involves

[6]To guarantee this property, a transaction must keep its initial time-stamp when restarted after abort.

two design decisions: defining a deadlock detection algorithm; defining a deadlock resolution method. Deadlock detection amounts to finding a cycle in the wait-for graph; in a distributed system, this involves information interchange between the nodes, so that the deadlock may be detected locally at any site. A typical algorithm (introduced in the R* system [Mohan et al. 1986]) is described in [Obermarck 1982]. Once detected, a deadlock is resolved by aborting one or more "victim" transactions. A victim is chosen so as to minimize the cost of its rollback according to some criterion (e.g., choosing the "youngest" transaction, or choosing a transaction local to the detection site, if such a transaction exists).

Two-phase locking, in its strict version, is the most commonly used technique to guarantee a consistent execution of a set of transactions.

9.2.3 Additional Issues

In this subsection, we present a few complements regarding concurrency control.

Object Creation and Deletion

In the above presentation, we assumed that the database contained a fixed set of objects, accessible through read or write operations. Object creation and deletion raises additional problems. Consider the following transactions:

T_1 : ... create(o_1) in set S; ... create(o_2) in set S; ...
T_2 : ... for all x in S do {read(x)... } ...

Suppose the concurrent execution of T_1 and T_2 generates the following schedule:

... create(o_1) in set S; for all x in S do {read(x)... }; create(o_2) in set S; ...

Although the schedule appears to be conflict-serializable (in the order $T_1; T_2$), this is not the case, because object o_2 "escaped" the "for all" operation by T_2. Such an object, which is potentially used before being created, is called a *phantom object*. A symmetrical situation may occur with object deletion (the object is potentially used after having been deleted).

The problem of phantoms is solved by a technique called *multilevel locking*, in which objects are grouped at several levels of granularity (e.g., in files, directories, etc.), and locks may be applied at these levels. Besides read and write locks, intentional locks (*intent_read*, *intent_write*, ...) may be applied at the intermediate levels. The following rules apply: (1) Before locking an object, a transaction must have obtained an intentional lock, in a compatible mode, on all the groups that contain the object; and (2) An object may only be created or deleted by a transaction T if T has obtained an *intent_write* lock on a higher level group including the object. This ensures that object creation and deletion will be done in mutual exclusion with potential accesses to the object.

Time-stamp Based Concurrency Control

The principle of time-stamp based concurrency control is to label each transaction with a unique time-stamp, delivered at creation, and to process conflicting operations in time-stamp order. Thus the dependency graph cannot have cycles, and the serialization order is determined a priori by the order of transactions' time-stamps.

Processing conflicting operations in time-stamp order may involve aborting a transaction if it attempts to perform an "out-of-order" operation, i.e., if the order of the conflicting operations does not agree with that of the corresponding time-stamps.

In the basic variant of the method, each object x holds two time-stamps: R_x, the time-stamp of the most recent transaction (i.e., the one with the largest time-stamp) that has read x, and W_x, the time-stamp of the most recent transaction that has written x. Suppose that a transaction T, time-stamped by t, attempts to execute an operation $o(x)$ on an object x, that conflicts with an earlier operation on x. The algorithm runs as follows:

- If $o(x)$ is a read, then it is allowed if $t > W_x$; otherwise, T must abort.
- If $o(x)$ is a write, then it is allowed if $t > max(W_x, R_x)$; otherwise, T must abort.

In both cases, if the operation succeeds, R_x or W_x are updated.

The case of the write-write conflict may be optimized as follows: if $R_x < t < W_x$, then the operation succeeds, but the attempted write by T is ignored since a most "recent" version exists. In other words, "the last writer wins". This is known as Thomas' write rule [Thomas 1979].

Multiversion concurrency control [Bernstein and Goodman 1981, Bernstein et al. 1987] is a variant of time-stamp based concurrency control that aims at increasing concurrency by reducing the number of aborts. The above rules for conflict resolution are modified so that a read operation is always allowed. If a write operation succeeds (according to the rule presented below), a new version of the object is created, and is labeled with the transaction's time-stamp. When a transaction time-stamped by t reads an object x, it reads the version of x labeled with the highest time-stamp less than t, and t is added to a read set associated with x. When a transaction T, time-stamped by t, attempts to write an object x, it must abort if it invalidates a previous read of x by another transaction. Invalidation is defined as follows: let W be the interval between t and the smallest write time-stamp[7] of some version of x. If there is a read time-stamp (a member of the read set of x) in the interval W, then the attempted write invalidates that read (because the version read should have been the one resulting from the attempted write).

Snapshot isolation [Berenson et al. 1995] is a further step towards improving performance by increasing concurrency, building on the notion of multiversion concurrency control. When a transaction T starts, it gets a start time-stamp $ts(T)$ and takes a "snapshot" of the committed data at that time. It then uses the snapshot for reads and writes, ignoring the effect of concurrent transactions. When T is ready to commit, it gets a commit time-stamp $tc(T)$, which must be larger than any other start or commit time-stamp. T is only allowed to commit if no other transaction with a commit time-stamp in the interval $[ts(T), tc(T)]$ had a write-write conflict with T. Otherwise, T must abort.

While this method improves performance, since there is no overhead in the absence of write-write conflicts, it is subject to undesirable phenomena called skew anomalies. Such anomalies occur when there exists constraints involving different objects, such as

[7]If no such time-stamp exists, $W = [t, \infty]$.

$val(o_1) + val(o_2) > 0$, where $val(o)$ denotes the value of object o. If two transactions T_1 and T_2 run in parallel under snapshot isolation, T_1 updating o_1 and T_2 updating o_2, then the constraint may be violated although no conflict is detected. The inconsistency would have been detected, however, under a serial execution.

[Fekete et al. 2005] propose techniques to remedy this situation, thus making snapshot isolation serializable. Snapshot isolation is used in several commercial database systems. It is also used as a consistency criterion for data replication (see 11.7.1)

9.3 Recovery

The function of recovery in a transaction management system is to guarantee atomicity[8] and durability in the face of failures. Recall that storage is organized in two levels: non-permanent (or volatile), such as main memory, and permanent, such as disk storage. Response to failures thus raises two issues: (1) how to recover from failures that only affect volatile storage, such as system crash or power supply failure; and (2) how to make the system resistant to media failures (excluding catastrophes). The latter problem is treated by data replication, which is the subject of 11.7. In this section, we only consider recovery from volatile storage loss. Note that the problem of processing an abort (rolling back a transaction) is a special case of failure recovery.

A failure may leave the system in an inconsistent state. Restoring the system to a consistent state is not always feasible, as shown by the following example. Consider a transaction T_1 which reads an object written by a transaction T_2. Suppose that T_1 commits, and that T_2 aborts (or is affected by a failure) at a later time. Then rolling back T_2 makes the value read by T_1 non-existent. But since T_1 has committed, its effect (which may depend on the read value) cannot be undone. A similar situation occurs if T_1 writes an object that was previously written by T_2.

Therefore, a desirable property is that the system be *recoverable*, i.e., that the failure of a transaction cannot invalidate the result of a committed transaction. This property may be translated in terms of schedules: a schedule is *strict* if, for any pair of transactions T_1 and T_2 such that an operation op_2 of T_2 precedes a conflicting operation op_1 of T_1 on the same object, the commit event of T_2 also precedes[9] op_1. Strictness is a sufficient condition for recoverability. Note that strict two-phase locking (9.2.2) only allows strict schedules, since locks (which prevent conflicting operations) are only released at commit time. In practice, most transaction systems use strict two-phase locking and therefore are recoverable.

The problem of transaction recovery is a complex one, and we only outline the main approaches used to solve it. For an in-depth analysis, refer to [Haerder and Reuter 1983] and to the specialized chapters of [Gray and Reuter 1993] and [Weikum and Vossen 2002]. In the rest of this section, we examine the issues of *buffer management* (how to organize information exchange between the two levels of storage in order to facilitate recovery), and *logging* (how to record actions so as to be able to recover to a consistent state after a failure).

[8]in the sense of "failure atomicity" (9.1).

[9]As a consequence, if T_2 aborts or fails, it does so before T_1 executes op_1.

9.3.1 Buffer Management

In order to exploit access locality, access to permanent storage uses a buffer pool in volatile storage, which acts as a cache, usually organized in pages[10] (unit of physical transfer). A read operation first looks up the buffer pool. If the page is present, the read does not entail disk access; if not, the page is read from disk unto the buffer pool, which may cause flushing the contents of a page, if there is no free frame in the pool. A write operation modifies the page in the pool (after reading it if it was not present). The contents of the modified page may be written on disk either immediately or at a later time, depending on the policy (more details further on).

Two main approaches are followed regarding updates in a database. In the direct page allocation, each page has only one version on disk, and all updates (either immediate or delayed) are directed to this page. In the indirect page allocation, updates are done on a different version of the page, called a *shadow*. Thus the original version of the page still exists, together with the most recent version (and possibly all intermediate versions, depending on the strategy adopted). To actually perform the update, one makes the (most recent) shadow to become the current version (an operation called *propagation*), and this may be done some time after the update. The advantage of indirect allocation is that it is quite easy to undo the effect of a sequence of updates, as long as the changes have not been propagated. The drawback is a performance penalty in regular operation (in the absence of failures), due to the overhead of maintaining the shadows and the associated data structures.

One may now identify three views of the database: (1) the current view, i.e., the most recent state, which comprises modified, unsaved pages in the buffer pool and up to date pages on disk; (2) the materialized view, i.e., the view after a system crash causing the loss of the buffer pool; this view consists of the propagated pages on disk; and (3) the physical view, again after a system crash, consisting of both propagated pages and unpropagated shadow pages on disk. If direct page allocation is used, the physical and materialized views coincide.

Two strategies may now be defined regarding the management of page updates in a transaction.

- ATOMIC: all the updates made by a transaction (from begin to commit) are performed in a single (indivisible) operation. Technically, this requires indirect page allocation (using indirection allows global propagation to be done by updating a single pointer, an atomic operation).

- NOT ATOMIC: some, but not all, of the pages updated by a transaction may be present in the materialized view. This happens with direct page allocation, since there is no simple way to write the contents of a set of updated pages as a single indivisible operation.

While ATOMIC achieves indivisible global write, its drawbacks are those of indirect page allocation. On the other hand, NOT ATOMIC is more efficient in normal operation, but necessitates to perform an undo algorithm after a system crash (see 9.3.2).

[10]Following the common usage, we distinguish between *pageframes* (containers) and *pages* (contents).

Regarding buffer pool management, the main issue is: "when is the contents of a modified page actually written to disk?". Writing a modified page to disk may be caused by two events: either its frame in the buffer pool is allocated to another page, or an explicit write decision is made by the transaction management system. Thus the options are the following:

- Steal: A page that has been modified by a transaction may be "stolen" (its frame is allocated to another transaction) before the first transaction has committed (the page must then be written to disk). This reduces the space requirements of the buffer pool.
- No Steal: The pages modified by a transaction remain in the buffer pool at least until the transaction commits.
- Force: Before a transaction commits, all the pages it has modified must have been written to disk.
- No Force: Some pages modified by a transaction may remain (non reflected on disk) in the buffer pool after the transaction has committed. This reduces I/O costs if such pages are needed by another transaction. The drawback is a more complex recovery procedure in the case of system failure.

Combining the three above binary choices leads to eight different policies. The (Steal, No Force) combination is frequently used, since it favors performance and allows flexible buffer management. System-R [Gray et al. 1981], the first full scale implementation of a relational database, used (Atomic, Steal, No Force), with shadow pages. Many current transaction managers are based on Aries (Algorithms for Recovery and Isolation Exploiting Semantics) [Mohan et al. 1992], which uses (Not Atomic, Steal, No Force). Aries emphasizes performance in normal operation, using direct page allocation rather than shadow paging, at the expense of a more complex recovery algorithm (see next section). This choice is justified, since failures are infrequent in current systems.

9.3.2 Logging

Logging is a general technique used to restore the state of a system after a failure which caused the contents of the volatile storage to be lost. A *log* is a sequence of records written on stable storage in the append-only mode. Each record registers a significant event that changed the state of the system. Typically, a record contains the date of the event, the state of the modified element before and after the change (possibly using a form of `diff` for economy of space), and the nature of the operation that performed the change.

In a transaction management system, the log may register the change of a logical element (a database record) or a physical element (a page); some systems register both classes of events, depending on their nature, thus constructing a "physiological" (physical + logical) log.

After a system failure, the log is used for two main purposes:

- To Redo the changes performed by a transaction that has committed at the time of the failure, but whose changes have not been written to the disk in entirety (this may occur with the No Force option). This guarantees durability.

- To UNDO the changes performed by a transaction that has not committed at the time of the failure. This guarantees atomicity. UNDO also applies to aborted transactions whose initial state has not yet been restored.

In addition to logging, the recovery system may use *checkpointing* to register periodically a consistent state. This limits the number of log records to be processed after a failure, since one only needs to restart from the last checkpoint. The difficulty lies in capturing a *consistent* state of the system, which may be a complex and costly process in itself, specially in a distributed system.

The log, which is on stable storage, must contain enough information to execute the necessary UNDO and REDO operations after a failure that caused the loss of the contents of volatile storage. This is ensured by the technique of *Write Ahead Logging* (WAL), which enforces two conditions:

- A transaction is only considered committed *after* all its log records have been written to stable storage (condition for REDO).
- All log records corresponding to a modified page must be written to stable storage *before* the page is itself written to permanent storage (condition for UNDO).

Recovery proceeds in three phases. An analysis of the situation first determines the earliest log records to be used for the next two phases. A REDO (forward) phase, applied to all transactions, including those committed, ensures that all updates done before the failure are reflected on permanent storage. Then an UNDO (backward) phase ensures that the effect of all uncommitted transactions is undone. Both REDO and UNDO are idempotent; thus, if a failure occurs during one of these phases, the operation may simply be restarted without additional processing.

While the principle of log-based recovery is fairly simple, implementing a WAL-based recovery system is a challenging proposition. Note that recovery interacts with concurrency control; in that respect, *strict* two-phase locking facilitates recovery, because it prevents dirty reads (9.2.1). A detailed description of the WAL-based recovery system of ARIES may be found in [Mohan et al. 1992]. A summary description is in [Franklin 2004].

9.4 Distributed Transactions

We have assumed, up to now, that a transaction takes place on a single site. We now consider *distributed transactions*, in which the objects may reside on a set of nodes connected by a network.

In a distributed system, the transaction manager is organized as a set of cooperating local managers, each of which manages the data present on its node. These managers need a mutual agreement protocol, in order to preserve global consistency. The main subject of agreement is the decision on whether to commit or to abort, since this decision must be consistent among all nodes involved in the transaction. In other words, distributed commitment should be *atomic* (all nodes commit, or none). In addition to commitment proper, an atomic commitment protocol can be used for concurrency control, through a method called commitment ordering [Raz 1992]. Atomic commitment is therefore a central issue for distributed transactions. In the following subsections, we specify the problem and examine its main solutions.

9.4.1 Atomic Commitment

On each node of the distributed system, a process executes the local transaction manager, and communicates with similar processes on the other nodes. We assume the following properties.

- Communication is *reliable* (messages are delivered unaltered) and *synchronous* (upper bounds are known for message transmission time and ratio of processor speeds at the different nodes, see 4.1.2). In addition, we assume that communication or process failures cannot partition the network.
- Processes may fail (in the fail-stop mode, see 11.1.3). A failed process may be repaired and be reintegrated in the transaction. A correct process is one that never failed.
- Each node is equipped with stable storage, which survives failures.

The assumption of reliable, synchronous communication is needed to ensure that an agreement protocol can run in finite time (see details in 11.1.3).

The atomic commitment protocol may be started by any process involved in the distributed transaction. Each process then casts a vote: YES if it is ready to commit locally (make permanent its updates to the local data), NO otherwise. The protocol must satisfy the following requirements.

- Validity: The decision must be either *commit* or *abort*.
- Integrity: Each process decides at most once (once made, a decision cannot be revoked).
- Uniform agreement: all processes that decide (be they correct or not) make the same decision.
- Justification: If the decision is *commit*, then all processes voted YES.
- Obligation: If all processes have voted YES, and if all processes are correct, the decision must be *commit*.
- Termination: each correct process decides in finite time.

A few remarks are in order about these specifications. The "atomic" property is ensured by *uniform* agreement, in which the decision of faulty processes is taken into account. This is because a process may perform an irreversible action, based on its decision, before failing. Agreement must therefore include all processes. Note that the two possible outcomes are not symmetric: while unanimity is required for *commit*, it is not required that all processes have voted NO to decide *abort* (in fact, a single NO vote is enough). The implication in the Justification requirement is not symmetric either: the outcome may be *abort*, even if all processes have voted YES (this may happen if a process fails between the vote and the actual decision). The Obligation requirement is formally needed to exclude trivial solutions, such as all processes always deciding *abort*.

We now present some protocols for atomic commitment.

9.4.2 Two-phase Commit

The two-phase commit protocol, or 2PC, has been initially proposed by [Gray 1978]. One of the processes is chosen as *coordinator*, and it starts[11] by requesting a vote from the other processes (participants). Each participant (and the coordinator itself) sends its vote (YES or NO) to the coordinator. When the coordinator has collected all votes, it decides (*commit* if all votes are YES, *abort* otherwise), and sends the decision to all participants. Each participant follows the decision and sends an acknowledge message to the coordinator. In addition, all events (message sending and receiving) are logged. This is summarized on Figure 9.1.

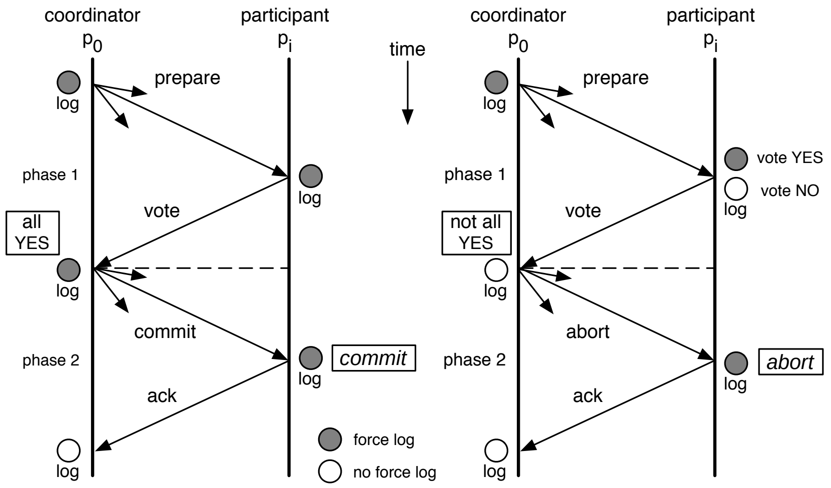


Figure 9.1. Two-phase commit: basic protocol

Logging may be forced (i.e., it takes place immediately and therefore blocks execution until finished) or non forced (it may be delayed, for example piggybacked on the next forced log).

Note that the coordinator and each participant start a round-trip message exchange, and can therefore use timeouts to detect failure. Recall that communication is assumed to be synchronous; call δ the upper bound for message propagation, and ϵ and T estimated upper bounds for processing a vote request and a vote reply, respectively. Then timeouts may be set at the coordinator and at each participant site, as shown on Figure 9.2.

If the coordinator has detected a participant's failure, it must decide *abort* (Figure 9.2a). If a participant has detected the coordinator's failure (Figure 9.2b), it first tries to determine whether a decision has been made, by consulting the other participants. If it fails to get an answer, it starts a protocol to elect a new coordinator.

[11] As noted above, the protocol may be initiated by any process, which then needs to send a "start" message to the coordinator.

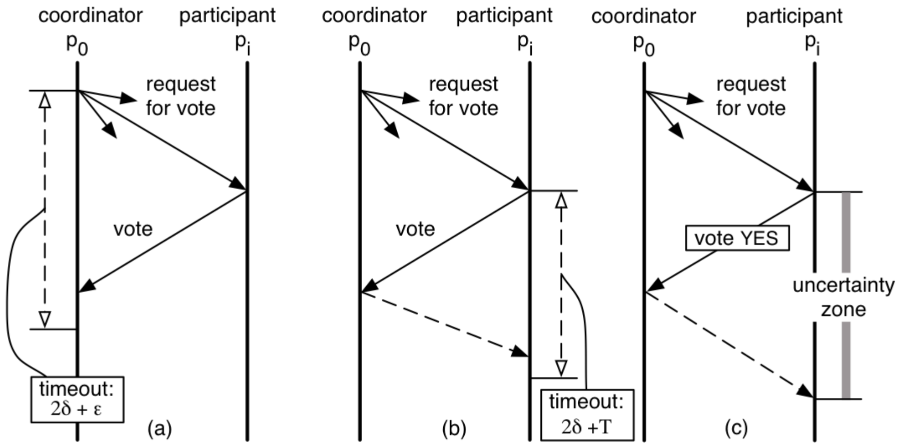


Figure 9.2. Two-phase commit: failure detection

When the coordinator or a participant restarts after a failure, it uses the log records to determine the state of the transaction, and to act accordingly. For instance, if it finds itself in phase 1, it restarts the transaction; if it finds itself in phase 2, it sends the logged decision to the participants; if the last log record is the end of the transaction, there is nothing to do.

Note that a participant that voted YES enters an "uncertainty zone" until it has received a decision from the coordinator, or until timeout (Figure 9.2c). This is because another participant may have already made either a *commit* or an *abort* decision (none is excluded for the time being). This may lead to a *blocking* scenario: suppose p_i has voted YES and is in the uncertainty zone; suppose that the coordinator made a decision, sent it to some participants, and then failed; suppose, in addition, that these latter participants also failed. Then a decision has actually been made, but p_i, although being correct, has no means to know it, at least until after the coordinator has been repaired (if the coordinator has logged the decision on stable storage before failing, the decision can be retrieved and resent to the participants).

Even if it is eventually resolved (either by retrieving the decision or by calling a new vote), a blocking situation is undesirable, because it delays progress and wastes resources that are held by blocked processes. Therefore various solutions have been devised to avoid blocking. Two of them are presented in the following subsection.

9.4.3 Non-Blocking Commitment Protocols

The first solution [Skeen 1981] proposed for avoiding blocking is the three-phase commit (3PC) protocol. If all participants have voted YES, the coordinator sends a PREPARE message to all participants; if not, it decides *abort*, like in 2PC.

When a participant receives a PREPARE message, it enters a "prepared to commit" phase, and replies with an ACK message to the coordinator. If the coordinator has received ACK from all processes, it decides *commit* and sends this decision to all participants. This protocol is summarized on Figure 9.3. Note that, if the communication system does not lose messages, acknowledgments are redundant.

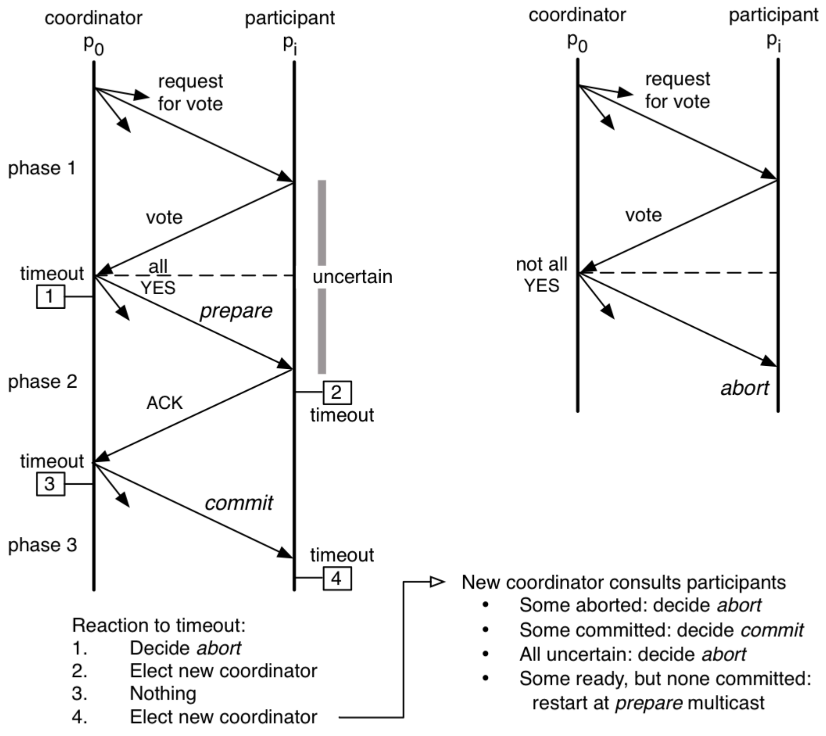


Figure 9.3. Three-phase commit

Why does this protocol eliminate blocking? The role of the "prepared to commit" phase is to reduce a participant's uncertainty as to the outcome of the transaction. After a participant p_i has voted YES, there is no situation in which p_i is uncertain and some process (say p_j) has already decided *commit*. Otherwise said: in 2PC, a process that decides *commit* "knows" that all have voted YES; in 3PC, a process that decides *commit* "knows", in addition, that all know that all have voted YES (because they all received the PREPARE message).

Dealing with failures is more complex than in 2PC, since there are more different possible states for the system. The situation is summarized on Figure 9.3.

A second, more elegant solution [Babaoğlu and Toueg 1993], consists in using a multicast protocol with strong properties, again with objective of reducing uncertainty among the participants. The algorithm is directly derived from 2PC: in the program of the coordinator, replace "send the decision (*commit* or *abort*) to the participants" by "broadcast the decision to the participants using UTRB (Uniform Timed Reliable Broadcast)". UTRB is a reliable multicast protocol (11.3.2) that has two additional properties:

- Timeliness: there exists a known constant δ such that, if the multicast of message m was initiated at real time t, no recipient process delivers m after real time $t + \delta$.
- Uniform Agreement: if any of the recipient processes (correct or not) delivers a message m, all correct recipient processes eventually deliver m.

The first property derives form the synchrony assumption for communication. The second property is the one that eliminates uncertainty, and thus prevents blocking: as soon as a participant has been notified of a decision by the coordinator, it "knows" that all correct participants will also receive the same information. If a participant times out without having received a decision, it can safely decide *abort* by virtue of the timeliness property.

9.4.4 Optimizing Two-phase Commit

As is apparent from the above description of 2PC (9.4.2), there is a significant difference, as regards the number of logging and communication operations, between the failure and no-failure cases. This observation suggest a potential way of improving the performance of 2PC in a situation in which the abort rate of transactions can be estimated, by adapting the protocol to the most likely behavior. The definition of the 2PC Presumed Commit (2PC-PC) and 2PC Presumed Abort (2PC-PA) protocols [Mohan et al. 1986] is an attempt towards this goal. The main idea is to reduce the number of FORCE log writes and to eliminate superfluous acknowledge messages.

2PC-PA is optimized for the case of abort, and is identical to standard 2PC in the case of commit. An analysis of the abort case shows that it is safe for the coordinator to log the *abort* record in the NO FORCE mode, and to "forget" about the transaction immediately after the decision to abort. Consequently, a participant which receives the *abort* message does not have to acknowledge it, and also logs it in the NO FORCE mode.

2PC-PC is optimized for the case of commit, and is identical to standard 2PC in the case of abort. In this protocol, the absence of information is interpreted as a *commit* decision. However, the 2PC-PC protocol is not exactly symmetrical to 2PC-PA, in order to avoid inconsistency in the following situation: the coordinator crashes after having broadcast the *prepare* message, but before having made a decision; upon recovery, it aborts the transaction and forgets about it, without informing anyone, since it does not know the participants. However, if a prepared participant times out and inquires the coordinator, it will get (by default) the *commit* answer, which is inconsistent.

To avoid this problem, the coordinator records (by a FORCE log) the names of the participants, before sending the prepare message. It will then know whom to inform of the abort decision after recovery from a crash. If the coordinator decides to commit, it FORCE logs this decision and sends it to the participants. Upon receipt, a participant FORCE logs the decision, without sending an acknowledge message. If the decision is to abort, the coordinator informs the participants that voted YES and waits for the acknowledgments. It then logs the decision in the NO FORCE mode.

Commit protocol	Messages		Forced log writes	
	Commit	Abort	Commit	Abort
2PC	$4p$		$1+2p$	
2PC-PA	$4p$	$3p$	$1+2p$	p
2PC-PC	$3p$	$4p$	$2+p$	$1+2p$

Table 9.1. Compared costs of 2PC protocols (from [Serrano-Alvarado et al. 2005])

The compared costs of the protocols are summarized on Table 9.1, which shows the gains of 2PC-PA and 2PC-PC with respect to standard 2PC, in the case of abort and commit, respectively. In this table, p denotes the number of participants.

To exploit this optimization, [Serrano-Alvarado et al. 2005] propose to dynamically adapt the commit protocol to the behavior of the application, by selecting 2PC-PA or 2PC-PC according to the observed abort rate. As a result, the average completion time of a transaction is reduced with respect to a non-adaptive commit scheme. The commit protocols are implemented in terms of Fractal components within the GoTM framework (see 9.7): each protocol contains the same basic components, with a different configuration.

9.5 Advanced Transaction Models

The transaction model described in sections 9.2 and 9.3 was developed for centralized databases used by short transactions with a moderate amount of concurrency. This model (called "traditional", or "flat") was found overly constraining for the complex, long-lived applications that came to existence with the rise of large scale distributed systems. Such applications are typically developed by composing several local applications, each of which may need to preserve some degree of autonomy. Relevant application domains include computer aided design and manufacturing (CAD/CAM), software development, medical information systems, and workflow management.

The coupling between the local applications may be strong (e.g., synchronous client-server) or loose (e.g., message-based coordination). The running time of the composite application may be long (hours or days, instead of seconds for traditional applications). The probability of failures increases accordingly. Blocking resources for the entire duration of the application would be overly expensive.

As a consequence, the application may be subject to one or more of the following requirements.

- A local application may be allowed to fail without causing the failure of the global application.
- After a failure, the global application may be allowed to roll back to an intermediate checkpoint, instead of undoing the whole work performed since its beginning.
- Some partial results may need to be made visible before the global application concludes.

Thus the atomicity and isolation requirements of the traditional transaction model may need to be relaxed for a global application, even if they remain valid for the composing sub-applications.

Presentations of advanced transaction models often use the canonical example of a travel agency, which prepares a travel plan by reserving flight tickets, hotel rooms, car rentals, show tickets, etc. For efficiency, reservations for the various kinds of resources can be done in parallel, each in a separate transaction. On the other hand, these transactions are not independent: if a hotel room cannot be found at a certain place, one may try a different location, which in turn would need changing the flight or car rental reservation, even if the corresponding transaction has already committed.

A number of models that relax the ACID properties in some way have been proposed under the general heading of “advanced transaction models”. [Elmagarmid 1992] and [Jajodia and Kerschberg 1997] are collections of such proposals, few of which have been tested on actual applications. In the following subsections, we present two models that have been influential for further developments. Both of them rely on a decomposition of a global transaction into sub-transactions. Nested transactions (9.5.1) use a hierarchical decomposition, while sagas (9.5.2) use a sequential one.

9.5.1 Nested Transactions

The nested transactions model has been introduced in [Moss 1985]. Its motivation is to allow internal parallelism within a transaction, while defining atomicity at a finer grain than that of the whole transaction.

The model has the following properties.

- A transaction (the parent) may spawn sub-transactions (the children), thus defining a tree of transactions.
- The children of a transaction may execute in parallel.
- A transaction T_j that is a sub-transaction of T_i starts after T_i and terminates before T_i.
- If a transaction aborts, all of its sub-transactions (and, by recursion, all its descendants) must abort; if some of these transactions have committed, the changes they made must be undone.
- A nested transaction (one that is not the root of the tree) may only (durably) commit if its parent transaction commits. By virtue of recursion, a nested transaction may only commit if all of its ancestors commit.

Thus the model relaxes the ACID properties, since durability is not unconditionally guaranteed for a committed nested transaction. The top-level (the root) transaction has the ACID properties, while the nested transactions (the descendants) only have the AI properties.

Several variants of this model have been proposed. In the simplest form, only leaf transactions (those at the lowest level) can actually access data; transactions at the higher level only serve as controlling entities. Other forms allow all transactions to access data, which raises the problem of concurrency control between parent and child transactions.

Concurrency control may be ensured by a locking protocol that extends 2PL with lock transmission rules between a transaction and its parent and children. A sub-transaction “inherits” the locks of its ancestors, which allows it to access data as if these locks were its own. When a sub-transaction commits, its locks are not released, but “inherited” (upward transmission) by its parent. When a sub-transaction aborts, only the locks it directly acquired are released; inherited locks are preserved. As a consequence, if two sub-transactions are in conflict for access to a shared object, one of them is blocked until their lowest common ancestor has committed.

The benefits of nested transactions are the possibility of (controlled) concurrent execution of sub-transactions, the ability to manage fine-grain recovery within a global transaction, and the ability to compose a complex transaction out of elementary ones (modularity).

9.5.2 Sagas

The traditional transaction model is not well suited for long-lived transactions, for two main reasons: (1) to maintain atomicity, some objects need to be locked for a long time, preventing other transactions from using them; and (2) the probability of a deadlock increases with the length of the transaction, thus leading to a potentially high abort rate. The Sagas model [García-Molina and Salem 1987] was proposed to alleviate these drawbacks. In this model, a global transaction (called a *saga*) is composed of a sequence of sub-transactions, each of which follows the traditional model. Thus a saga S may be defined as follows:

$$S = (T_1; T_2; \dots T_n)$$

where the T_is are the sub-transactions of S.

A saga only commits if all of its sub-transactions commit; thus a saga is atomic. However, the execution of a saga can be interleaved with the execution of other sagas. As a consequence, sagas do not have the isolation property, since a saga may "see" the results of another, partially completed, saga.

If a sub-transaction of a saga aborts, the entire saga aborts, which means that the effect of the sub-transactions executed up to the current point needs to be canceled. However, since these transactions have committed, a special mechanism must be introduced: for each sub-transaction T_i, one defines a *compensating transaction* C_i whose purpose is to cancel the effect of the execution of T_i.

Thus the effect of the execution of a saga $S = (T_1; \dots T_n)$ is equivalent to either that of

$$T_1; T_2; \dots T_n,$$

in which case the saga commits and achieves its intended purpose, or that of

$$T_1; T_2; \dots T_k; C_k; C_{k-1} \dots C_1 \qquad (k < n),$$

in which case the sub-transaction T_k has aborted, and its effect, as well as that of the preceding sub-transactions, has been compensated for[12].

Recall that isolation is not maintained for sagas. Two consequences follow:

- Compensation of a saga S does not necessarily bring the system back to its state before the execution of S, since sub-transactions of other sagas may have taken place in the meantime.
- Other sagas may have read the results of committed sub-transactions of a saga S, before they were compensated for. Applications are responsible for dealing with this situation.

[12]Note that there is no need for a compensating transaction for the last sub-transaction T_n.

The main practical problem with building applications based on sagas is to define compensating transactions. This may be difficult, or even impossible if a sub-transactions has executed "real" actions.

The main current application of the "advanced transactions" models is in the area of web services transactions (see 9.6.2).

9.6 Transactional Middleware

As seen above, the notion of a transaction has evolved from the traditional model to a variety of "advanced" models, in order to comply with the requirements of complex, composite applications. Transaction support software has evolved accordingly. In centralized databases, transactions were supported by a set of modules closely integrated within database management systems. As systems became distributed, and as transactions were applied to non-database services such as messaging, transaction management was isolated within specialized middleware components. A transaction processing system is typically organized into a transaction manager (TM) and a set of resource managers (RM). Each resource manager is in charge of a local resource, such as a database or a message queuing broker, and is responsible for ensuring transactional properties for access to the local resource. The transaction manager ensures global coordination between the resource managers, using the 2PC protocol.

Several standards have been developed over time to provide a formal support to this organization. The Open Group Distributed Transaction Processing (DTP) standard [X-OPEN/DTP] defines the interfaces between the application and the TM (TX interface), and between the TM and the RMs (XA interface), as shown on Figure 9.4.

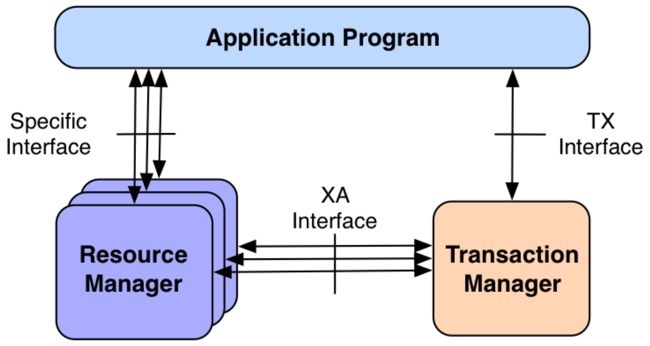


Figure 9.4. Transactional interfaces in X-Open DTP

In the rest of this section, we examine two frameworks that have been developed in recent years for transaction management. These frameworks are representative of two families of middleware systems. In 9.6.1, we describe the principles of transaction management in JEE, a platform for component-based programming. In 9.6.2, we present WS-TX, a standard proposed for web services transactions. We conclude in 9.6.3 with a brief outline of current advances in transactional middleware.

9.6.1 Transaction Management in JEE

JEE [JEE] is a platform for developing distributed, component-based applications in Java. It uses containers (7.5.1) to manage application-built components called *enterprise beans*[13]. Beans contain the code of the application and call each other using (possibly remote) method invocation.

Each bean is is managed by a container, which intercepts all method calls and acts as a broker for system services. There are two kinds of transactions, which differ by the method used for transaction demarcation.

- Bean-managed transactions, in which transaction demarcation instructions (*begin*, *commit*, *rollback*) are explicitly written by the developer of the bean, using the `javax.transaction.UserTransaction` interface.
- Container-managed transactions, in which transaction demarcation is done by the container, using declarative statements provided by the developer, as annotations attached to the bean.

In container-managed transactions, the main issue is to determine whether a method of a bean should be executed within a transaction, and if yes, whether a new transaction should be started. This is specified by a transaction attribute, attached to the method (a transaction attribute may also be attached to a bean class, in which case it applies to all the application methods of the class). The effect of the attribute is described in Table 9.2.

Transaction attribute	Calling activity	Method's transaction
`Required`	None	$T2$
	$T1$	$T1$
`RequiresNew`	None	$T2$
	$T1$	$T2$
`Mandatory`	None	Error
	$T1$	$T1$
`NotSupported`	None	None
	$T1$	None
`Supports`	None	None
	$T1$	$T1$
`Never`	None	None
	$T1$	Error

Table 9.2. Transaction attributes in container-managed transactions

The column "Calling status" indicates whether the call is done within a transaction (denoted as $T1$) or outside a transaction. The column "Method's transaction" indicates whether the method should be run within a transaction (which may be the calling transaction, $T1$, or a new transaction, $T2$).

For instance, suppose the called method has the `Required` attribute. Then, if it is called from outside a transaction, a new transaction must be started. If the method is called

[13]There are several kinds of beans (differing by persistence properties). We ignore these distinctions, which are not relevant in this presentation.

from within a transaction, it must be executed under that transaction. Some situations cause an error, in which case an exception is raised; if a transaction was running, it will automatically be rolled back.

In order to explicitly abort a container-managed transaction (for example, if the bean throws an application exception) the application should invoke the `setRollbackOnly` method of the `EJBContext` interface.

9.6.2 Web Services Transactions

Web services [Alonso et al. 2004] provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. In a more specific sense (see [W3C-WSA 2004]), Web services define a set of standards allowing applications to be integrated and executed over the Internet, following a service oriented architecture such as described in 3.3.4.

Web services (which would be more appropriately called Internet services) typically use a loosely-coupled mode of communication, in which applications execute over long periods and may negotiate the terms and conditions of their interaction. The conventional transaction model based on ACID properties proves too constraining for Web services. Therefore, new standards are being developed to support extended transaction models adapted to the Web services environment.

The emerging standard for Web services transactions is the set of Web Services Transaction specifications (collectively known as WS-TX) produced by OASIS (Organization for the Advancement of Structured Information Standards [OASIS]), a consortium that develops "open standards for the global information society".

The WS-TX collection is composed of three standards.

- WS-Coordination (WS-C) [OASIS WS-TX TC 2007c] This specification describes an extensible framework used to coordinate a number of parties participating in a common activity. The framework has two parts: (a) a generic part, which allows participants to share a common context and to register for a common activity, possibly controlled by a coordinator; and (b) a specific part, which defines a particular protocol in the form of a "coordination type". WS-Coordination is the base on which the transaction services described below are built, as specific coordination types. WS-Coordination is presented in 6.7.

- WS-AtomicTransaction (WS-AT) [OASIS WS-TX TC 2007a]. This specification defines an atomic transaction protocol, to be used for short duration transactions between participants with a high degree of mutual trust. It is based on 2PC (9.4.2), with two variants: Volatile 2PC, for participants managing volatile resources (e.g., caches), and Durable 2PC, for participants managing durable resources (e.g., databases).

- WS-BusinessActivity (WS-BA) [OASIS WS-TX TC 2007b]. This specification defines a protocol (in the form of a set of coordination types) to build applications involving long-running, loosely coupled distributed activities, for which the conventional ACID transaction model is inadequate. WS-BusinessActivity is based on an extended transaction model (9.5) in which the results of an activity may be visible before its completion, and the effect of a completed activity may be undone by means of compensation, such as defined in the saga model (9.5.2).

Separating WS-Coordination from the specific transaction protocols has two main benefits:

- Separation of concerns. The functions related to coordination are isolated and may serve for other purposes than transaction management.
- Extensibility. New transaction protocols, in addition to WS-AT and WS-BA, may be added in the future as coordination types in WS-C.

While WS-AT is essentially a conventional 2PC protocol for ACID transactions, its main advantage is to allow interoperability between applications that participate in a common transactional activity, by wrapping them into Web services. Thus independently developed applications, using different platforms and standards, may be made to interoperate in a closely coupled environment.

Since WS-BA addresses applications that integrate loosely coupled, independent participants running in different domains of trust, flexibility is an important requirement. It is achieved through the main following features:

- Nested scopes. A long running activity may be structured as a set of tasks, each of which is a short duration unit of work, which may typically follow a conventional transaction model. A task may itself be organized in a hierarchy of (child) tasks. Each task, which uses a collection of Web services, defines a *scope*. Scopes can be arbitrarily nested. This model is close to that of nested transactions (9.5.1), but is more flexible, since a parent may catch an error in a child task, and decide on whether to compensate, to abort, or to produce a non-atomic outcome.
- Flexible coordination model. WS-BA supports two coordination types, which define the behavior of the coordinator of a transaction. In the AtomicOutcome type, the coordinator must direct all participants either to close (i.e., to complete successfully) or to compensate. In the MixedOutcome type (optional), the coordinator may direct each individual participant to either close or compensate.
- Custom termination protocol. Two protocols are defined, each of which can use one of the two above coordination types. The difference between these protocols is about which entity decides whether a participant's activity is terminated. In the BusinessAgreementWithParticipantCompletion (BAwPC) protocol, the decision is made by the participant, which notifies the coordinator. In the BusinessAgreementWithCoordinatorCompletion (BAwCC) protocol, the decision is made by the coordinator, which has information about the tasks requested from the participant. In both cases, according to its decision about the outcome of the global activity, the coordinator requests the participant either to close (successfully complete) or to compensate the task.

The life-cycle of a transaction, as seen by the coordinator or by a participant, is represented as a state diagram. Transitions between states are triggered by messages sent either by a participant or by the coordinator.

In the BAwPC protocol (Figure 9.5), when an active participant decides that it has completed its work, it notifies the coordinator by a *Completed* message and goes to the Completed state. The coordinator then decides to either accept the participant's work (message *Close*) or to request that the work be compensated for[14] (message *Compensate*). In both cases (in no error occurs during compensation) , the participant eventually goes to the *Ended* state, in which it forgets about the transaction. Other states shown on the figure are introduced to deal with abnormal events at various stages of the work.

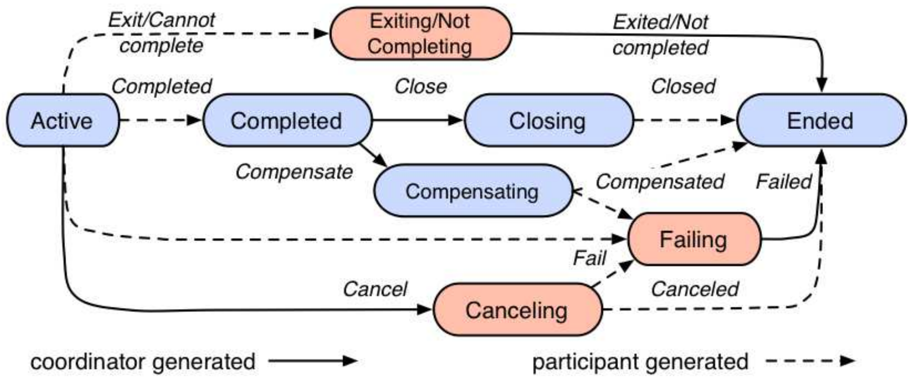


Figure 9.5. BusinessAgreementWithParticipantCompletion: state diagram (adapted from [OASIS WS-TX TC 2007b])

In the BAwCC protocol (Figure 9.6), the decision about termination is made by the coordinator, which sends a *Complete* message to the participant, which then goes to a Completing stage. In the absence of errors, the participant answers with a *Completed* message and goes to the completed state. The rest of the protocol is as described above. Various errors may occur during the Completing phase, leading to the additional transitions shown on the figure.

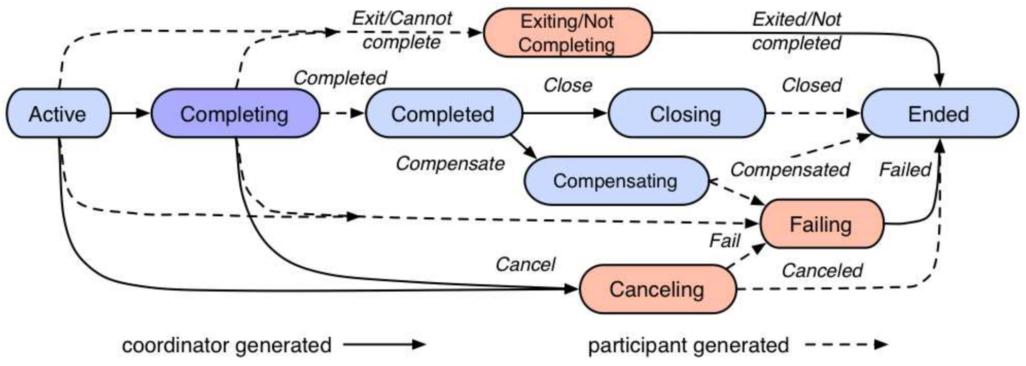


Figure 9.6. BusinessAgreementWithCoordinatorCompletion: state diagram (adapted from [OASIS WS-TX TC 2007b])

An open source implementation of the WS-TX protocols is underway at

[14]The WS-BA standard does not define the compensation operations, which are the responsibility of the application's logic.

[Kandula 2007]. It includes an extension to the WS-BA protocol, described in [Erven et al. 2007], which introduces a new role called the initiator to facilitate the development of applications using WS-BA.

9.6.3 Advances in Transactional Middleware

The emergence of new applications areas has motivated a variety of requirements regarding transactions. Several standards have been proposed to satisfy these requirements, which cover a wide spectrum. As a consequence, it has become clear that no single transaction model is able to cover all possible cases. Transactional middleware is therefore required to be adaptable and extensible. More precisely, it should satisfy the following requirements.

- Allowing various transactional services to be built on a common infrastructure, by composing a set of parts.
- Allowing the coexistence of several "personalities" of transaction support systems, conforming to various transaction standards and providing different guarantees to the applications.
- Allowing run-time extension and adaptation to react to changes in the environment.

To comply with these requirements, advances have been done in two complementary directions.

The first direction is to define new abstractions in order to identify commonalities in the transaction models and tools, and to better specify these models and tools. Separating coordination from transaction management in the WS-TX protocols (9.6.2) is a step in this direction. Another example of this approach is the abstraction of transaction demarcation (i.e., the policies that specify whether and how an action is to be executed under an active transaction), in [Rouvoy and Merle 2003].

The second direction is to design middleware infrastructures to support adaptable and extensible transaction systems, specially through the use of components. Two recent examples are the Argos framework [Arntsen et al. 2008] and the GoTM framework [Rouvoy and Merle 2007]. GoTM is presented in the next section.

9.7 Case Study: GoTM, a Framework for Transaction Services

GoTM (GoTM is an open Transaction Monitor) is a component-based framework for building highly adaptable and extensible transaction services. To achieve this goal, GoTM relies on an abstract architecture for transaction services, in the form of a set of common design patterns, which are reified as assemblies of fine-grained components based on the Fractal component model (7.6). Thus GoTM provides an extensible component library, which can be used to develop the main functions required from a transaction service.

The main aspects of a transaction service that may be subject to variation are the following:

- Transaction standard. Various standards have been defined for different environments. These standards specify the user interface of the transaction service. Examples include Object Transaction Service (OTS) for Corba, Java Transaction Service for JEE, and Web Services Atomic Transaction for Web Services. Some of these standards rely on common notions (e.g., the flat transaction model) or common mechanisms (e.g., transaction creation, context management).

- Transaction model. Various transaction models have been developed (see 9.5) to fit different application requirements. One major aspect of variation is the degree of isolation. Again, the goal is to allow different transaction models to coexist in a transaction service.

- Commitment protocol. As noted in 9.4.4, the two-phase commit protocol has a number of variants optimized for various execution environments. The abstract notion of a 2PC protocol captures the core notions that are common to these variants.

GoTM allows building transaction services that can be adapted according to the above criteria. This is achieved by the systematic use of reification of different notions in the form of fine-grain components. This reification process is applied at two levels:

- For the design patterns used to specify the architecture of the transaction service. Reifying the patterns in the form of components allows using component description and composition tools to adapt the structure of the transaction service.

- For the functions and the execution states of a transaction service. Since a fine-grain component represents an elementary aspect, it may be easily shared among different transaction services.

Thus the adaptation mechanisms of GoTM essentially apply to the architecture of the transaction services, rather than to the contents of the elements.

The next two subsections describe the architecture of the GoTM framework (9.7.1) and the main features of its implementation (9.7.2). The last subsection (9.7.3) shows how GoTM may be used for transaction service adaptation. This presentation only describes the main elements of GoTM. Refer to [Rouvoy 2006] and [Rouvoy and Merle 2007] for details.

9.7.1 Architecture of GoTM

GoTM is organized in two levels: a static part, the *Transaction Service*, whose function is to create transactions; and a dynamic part, the *Transactions* created by the transaction service. Both parts rely on design patterns, as shown on Figure 9.7.

The front-end components in both parts of GoTM are based on the FAÇADE design pattern [Gamma et al. 1994]. The role of this pattern is to provide a simple, unified interface to a complex system that may itself involve a number of interfaces[15]. GoTM uses FAÇADE to implement various transactions standards (e.g., JTS, OTS, WS-AT) in

[15]FAÇADE is thus related to the ADAPTER pattern (2.3.3). While ADAPTER gives access to an existing interface by way of another existing interface, FAÇADE builds a new interface to integrate a set of existing interfaces.

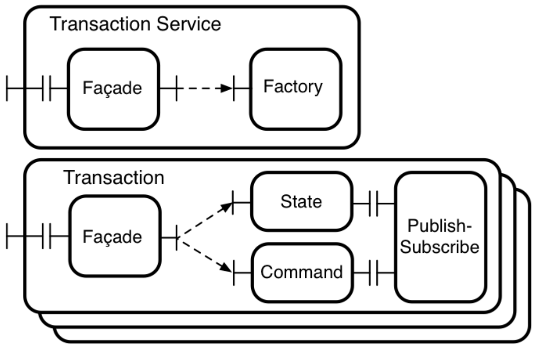


Figure 9.7. Patterns in the architecture of GoTM (adapted from [Rouvoy and Merle 2007])

terms of the interfaces exported by its component library. A specific FAÇADE component is automatically generated for each standard (e.g., *JTS-Façade*, *OTS-Façade*, etc.).

The core of the transaction service (Figure 9.8) is a *Transaction Factory* component, based on the FACTORY design pattern (2.3.2). This component creates transaction instances conforming to a specified transaction model (also represented as a component). The transaction model acts as a set of templates, which may be cloned to create components of transaction instances. For better performance, the factory is enhanced with caching and pooling facilities. Note that this construction makes use of the ability of the Fractal model to share components (the component *Factory* is shared between the three transaction services of the different standards).

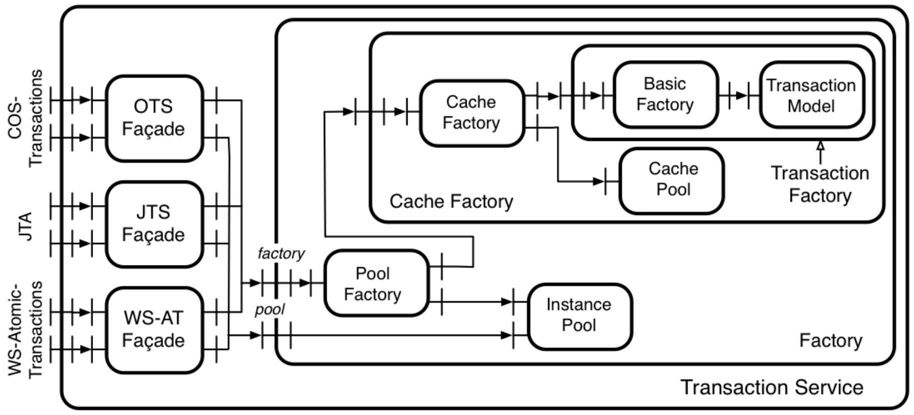


Figure 9.8. The GoTM transaction service (adapted from [Rouvoy and Merle 2007])

As an example, Figure 9.9 shows the transaction model component for JTS transactions. This model acts as a prototype to create instances of transactions using the JTS standard and its JTA interface. Note the conventions for the figures: a star denotes an interface of type *Collection*, and components shared with other models are enclosed by dotted lines.

In addition to FAÇADE, transaction instances use three main design patterns: STATE, COMMAND, and PUBLISH-SUBSCRIBE.

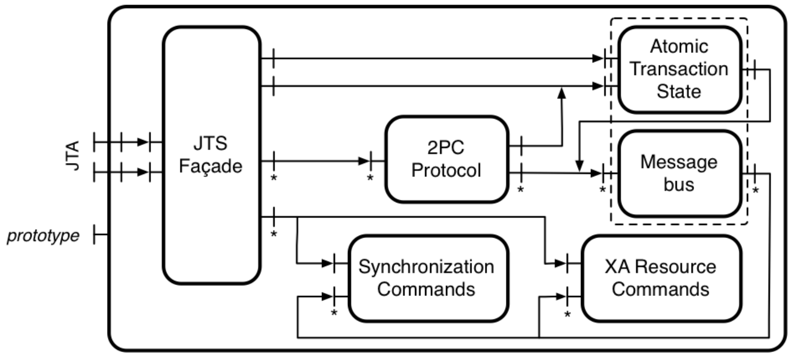


Figure 9.9. A prototype for JTS transactions in GoTM (adapted from [Rouvoy 2006])

The function of State is to represent and to control the different states of a transaction. Using this pattern (embodied in the component *AtomicTransactionState*), GoTM implements the state automaton defined by a transaction model (Figure 9.10). Each state is reified by a component, while the transitions between states are represented by bindings between these components (further comments on this technique in 9.7.2).

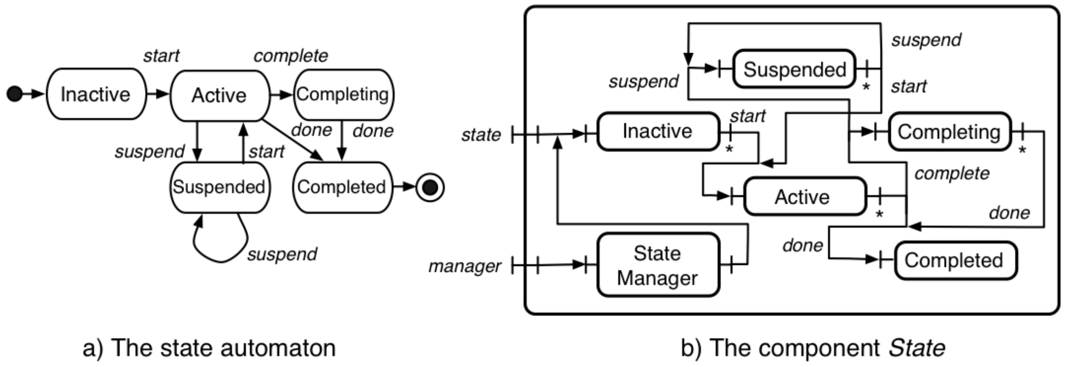


Figure 9.10. Controlling state in GoTM (adapted from [Rouvoy and Merle 2007])

The function of Command is to encapsulate a command (and its parameters) into an object. This provides an abstraction to build generic components, and facilitates the building of undoable commands (e.g. by storing the command objects into a stack).

In GoTM, Command is used to notify the participants to a transaction of significant events, defined as state changes. This applies both to actual participants (those involved in the transaction) and to "synchronization participants", which are only notified of the beginning and the end of the transaction.

The use of the Publish-Subscribe pattern is described in the next section as an illustration of the implementation techniques of GoTM.

9.7.2 Implementation of GoTM

The implementation techniques used in GoTM are intended to make evolution easy. Using a component-based architecture is a first step towards that goal. In addition, a special

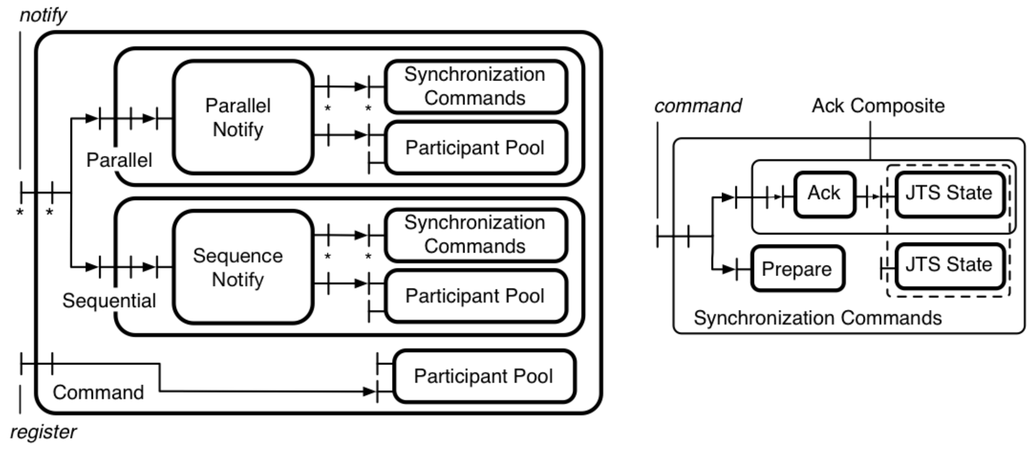


Figure 9.11. Using the COMMAND pattern in GoTM (adapted from [Rouvoy 2006])

effort has been made towards a very fine grain definition of the components and interfaces, guided by the principle of separation of concerns. Each function of a transaction service is reified in the form of a component. Within a transaction model, reusable functions are identified, and implemented as fine-grained shared components. The states that are defined by a specific transaction model are likewise reified. Thus a transaction service is described as an assembly of "micro-components".

This approach has the following benefits:

- The interfaces of the micro-components are simple (in practice, no more than four operations per interface).
- Component sharing and reuse is favored (and made technically easy by the component sharing facility of the Fractal model).
- Configuration attributes are reified into bindings, thus allowing the use of an ADL as an evolution tool.

This is illustrated by the example of the *Publish-Subscribe* component of transactions (Figure 9.12). *Publish-Subscribe* is used to synchronize the transaction participants during the execution of the Two-phase commit protocol (9.4.2).

In the implementation shown, the *publish* operation may be synchronous or asynchronous. Selecting one of these options could have been done through a parameter in the *publish* interface. Instead, the options are provided through two micro-components, *Synchronous* and *Asynchronous*, which have a common interface *publish*, of type *Collection*. The client binds to both components, through client interfaces (*pub-sync* and *pub-async*, respectively), and selects the component to use according to its needs (for instance, emphasizing safety or performance). Note that bindings may be modified at run time, e.g., to introduce a new option. The role of the *State Checker* components is to ensure that the execution of *publish* conforms to the state automaton that specifies the semantics of this operation. Components *State* and *Subscriber Pool* are shared (this is denoted by dotted lines on the figure).

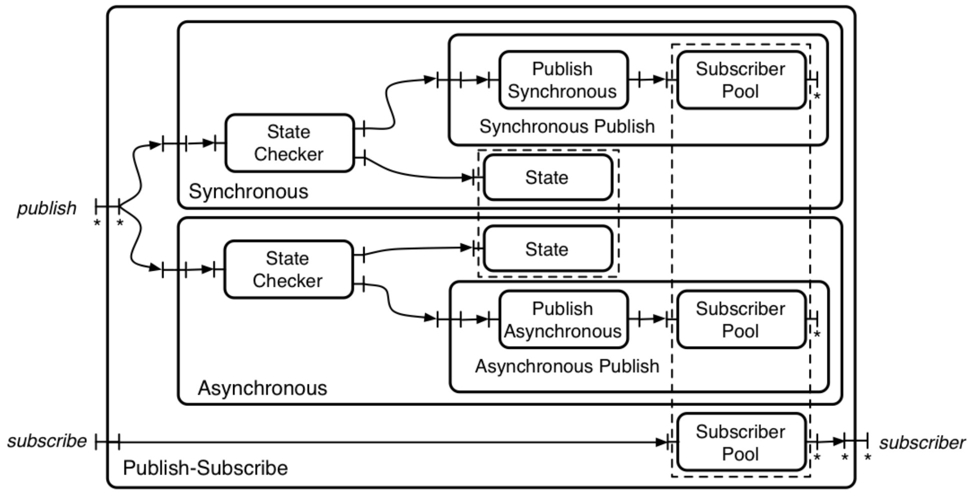


Figure 9.12. Publish-Subscribe in GoTM (from [Rouvoy 2006])

9.7.3 Using GoTM

We illustrate the adaptation capabilities of GoTM with three simple examples.

- Adapting the commitment protocol of a transaction. The algorithm of the commitment protocol is isolated through the STRATEGY design pattern. Thus, to modify this algorithm (e.g., to use 2PC-PA instead of 2PC-PC), one only needs to update the description of the component which implements the protocol. This is done (statically) by modifying one line in the ADL description of the framework. This may also be done at execution time (see [Rouvoy 2006] for details).

- Adapting the management of the participants to a transaction. This may be done at two levels. One may first add new commands in the *Synchronization Commands* components (each command is itself represented as a component, according to the COMMAND pattern). One may also replace the whole component *Synchronization Commands*, in order to modify the type of the participants of a transaction (i.e. the available commands). Both extensions are again performed by a simple, local, modification of the ADL description.

- Adapting the transaction standard. As shown in 9.7.1, this is done by providing a model for the new standard, together with a front end implemented as a new *Façade* component.

In conclusion, the separation of concerns which guided the design of GoTM, together with the systematic application of design patterns and the implementation techniques using micro-components achieve the goal of flexibility set up by the designers of GoTM. Experience reported in [Rouvoy et al. 2006a, Rouvoy et al. 2006b] shows that these benefits do not entail a significant performance overhead.

GoTM is described in [Rouvoy 2006, Rouvoy and Merle 2007]. It is available under an open source LGPL license at [GoTM].

9.8 Historical Note

The notion of a transaction as a well-identified processing unit acting on a set of data was empirically used in commercial exchanges, even before computers. With the advent of databases in the late 1960s and early 1970s, this notion was embodied in the first transaction processing monitors, such as CICS, developed by IBM. However, although atomicity was identified as an important issue, no formal model existed, and the implementations relied on empirical rules.

The first attempt at establishing a formal support for concurrency control is the landmark paper [Eswaran et al. 1976]. It lays the foundations of the theory of serializability, which is still the base of today's approach to concurrency control, and introduces two-phase locking. Fault tolerance issues were also beginning to be investigated at that time (see 11.10), and found their way into the database world. Thus [Gray 1978] introduces the DO-UNDO-REDO and WAL (write-ahead logging) protocols. The ACID properties are identified in the early 1980s (e.g., [Gray 1981]). An early survey on recovery techniques is [Haerder and Reuter 1983]. All these notions were applied to the design of large scale database management systems, among which System-R [Gray et al. 1981], which has been extensively documented.

Distributed commitment, the key problem of distributed transaction processing, is examined in [Gray 1978], which introduces two-phase commit (2PC). This stimulated research on non-blocking protocols, first leading to three-phase commit (3PC) [Skeen 1981], and later to other developments described further on.

The first distributed databases were developed in the early 1980s. R* [Mohan et al. 1986] was a distributed extension of System-R. Other influential distributed transactional systems include Camelot, later known as Encina [Eppinger et al. 1991] and UNITS (Unix Transaction System), later known as Tuxedo [Andrade et al. 1996] Both Encina and Tuxedo have evolved into products that are still in current use.

Transaction models relaxing the ACID rules (known as "advanced" models) were investigated in the 1980s. The most influential ones (9.5) are nested transactions [Moss 1985] and sagas [García-Molina and Salem 1987].

In the early 1990s, considerable progress was made in the understanding of the fundamental aspects of distributed transactions. Non-blocking atomic commitment was analyzed in detail [Babaoğlu and Toueg 1993, Guerraoui and Schiper 1995] and its relationship with consensus was clarified [Guerraoui 1995]. For recent advances in this area, see [Gray and Lamport 2006].

The development of middleware platforms for distributed applications, which started in the mid-1990s, called for advances in transactional middleware. After experience with technical solutions has been gathered, the main effort was devoted to the elaboration of standards. This work is still ongoing. "Advanced" transaction models, which are well adapted to the constraints of long transactions, are finding their way into web services.

The mainstream research in the field of transactions has now shifted to transactional memory, a mechanism using transactions to manage concurrent memory access, a potential bottleneck for applications using multicore processors. Transactional memory may be implemented by hardware, by software, or by a combination of both. See [Larus and Rajwar 2007] for a survey of this area.

Historical sources for various aspects of transactions may be found in the historical notes of [Gray and Reuter 1993]. A recent detailed historical account of transaction management is [Wang et al. 2008].

Chapter 10

Systems Management

The goal of systems management is to maintain a system's ability to deliver its service with a prescribed quality. The management of software systems and applications involves two main aspects: managing the resources needed by the system (observation, monitoring, mediation, allocation); setting up and adapting the system's structure to meet its requirements and to respond to changes in the environment (configuration, deployment, reconfiguration). This chapter mainly concentrates on these latter topics. Other aspects of management (fault tolerance, resource management, and security) are examined in detail in the following chapters.

10.1 Main Concepts and Terminology

10.1.1 Introducing Management

We start by recalling a few definitions from the introduction of Chapter 2. A *system* is an assembly of hardware and/or software parts, which is designed to fulfill a function (the provision of one or several *services*). The system interacts with an external environment; the services that it provides rely on assumptions about the services that it gets from the environment. Services are further discussed in 2.1.

The environment of a system comprises its human users, external software with which it may interact, and its physical environment, including devices that it is designed to monitor or to control. A system may include material components (processors, disks, routers, networks, sensors, etc.), a software infrastructure providing common services (operating systems, communication protocols, middleware, etc.), and applications delivering specific services. Each part of a system may also be considered in isolation, as the provider of an individual service; its environment then includes the rest of the system.

Some of the interactions of a system with its environment are part of its function; they are intended and predictable. Others, such as failures, attacks, unexpected variations of the demand, may be unwanted or unpredictable. We define *management* (or *administration*) as the function that aims at maintaining a system's ability to provide its specified services, with a prescribed quality of service, in the face of both wanted and unwanted interactions. Two approaches may be taken to achieve this goal.

- Prevention, in order to enable the system to perform its functions and to eliminate undesired interactions (or at least to reduce the probability of their occurrence).
- Detection and cure, in order to cancel or to lessen the effects of an adverse event or situation.

Detection and cure is an *adaptation* process, which may take various forms: changing the system's internal parameters; changing the system's structure; (re-)allocating resources. It may involve stopping and restarting the system, or it may be performed at run time, without service interruption. Adaptation is essentially a *control* activity, involving an event-reaction loop: the management system detects events that may alter the ability of the managed system to perform its function, and reacts to these events by trying to restore this ability. The "management" and "managed" systems may be defined at various levels of detail. In a complex system, the control loop may thus involve several levels of abstraction. In the current state of the art, most management tasks involve human intervention.

The above definition of management is wide enough to cover a variety of situations: managing users (accounts, rights, etc.); reacting to a failure, an attack or a peak in the demand; observing, monitoring, logging, collecting statistics on various parts of the system; managing hardware and software resources (bookkeeping, allocation, replacement, addition).

The term *resource* applies to any identifiable entity (physical or virtual) that is used for service provision. An entity that actually provides a service, using resources, is called a *resource principal.* Examples of physical resources are processors, memory, disk storage, routers, network links, sensors. Examples of virtual resources are virtual memory, network bandwidth, files and other data (note that virtual resources are abstractions built on top of physical resources). Examples of resource principals are processes in an operating system, groups of processes dedicated to a common task (possibly across several machines), various forms of "agents" (computational entities that may move or spread across the nodes of a network). Note that a resource principal may itself be viewed as a resource by a higher level principal, e.g., when a process manages sub-processes, etc.

When designing a management system, the question arises of how to separate the functions related to management from those of the managed system. This issue is akin to defining "extra-functional" vs "functional" properties (2.1.3). Actually, as implied by the definition, the management tasks essentially deal with extra-functional aspects (e.g., performance, security, fault tolerance). Thus the distinction between the management and the managed systems is in a sense arbitrary. However, the principle of separation of concerns favors a design in which the boundary between the two systems is clearly identified, with interfaces explicitly dedicated to management.

A remark on terminology: in an organizational context, "management" is about setting the rules, while "administration" is about seeing that the rules are followed. In the area of systems and networks, the distinction is less clear[1]. "Management" seems to be the preferred term for networks, while there is no distinct trend in the systems and applications area. In this chapter, we do not make a significant distinction between the two terms.

[1]In the 2003 LISA Conference (Large Installation System Administration), 3 session titles and 4 paper titles contained the word "management".

Most efforts for developing models and techniques for management have been devoted to common infrastructures such as operating systems and networks. Application-level management has received less attention. However, the situation is changing, because applications are subject to higher demands, and because of the trend towards more open systems and middleware.

In the context of this book, we cover all aspects of management, with specific emphasis on application-level. In the remainder of this section, we first briefly survey the general concepts of administration (10.1.2). We then analyze the main functions relevant to systems management (10.1.3). We conclude the section by introducing the main challenges of management and the notion of autonomic computing.

10.1.2 Designing and Implementing Management Functions

Management Model

The current conceptual approach to systems and network administration derives from the management standards defined between 1988 and 1990: the Simple Network Management Protocol (SNMP), which is the base of network management in the Internet world; the Common Management Information Services and Protocols (CMIS/CMIP), specified by the ISO and the ITU, which is mainly used in the telecommunications world. These standards are open, to allow independence from vendors. They are based on a common generic model (manager-agent) that is summarized below.

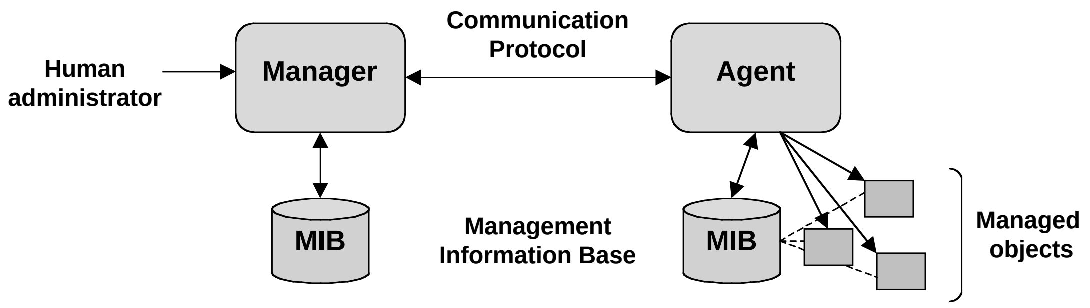


Figure 10.1. A generic model for network and systems management (simplified)

The model relies on the following entities (Figure 10.1).

- A *manager* acts as an interface for a human administrator, and usually resides on a host dedicated to administration. It interacts with one or more agents.
- An *agent* resides on an administered host. It acts as a local delegate of the manager for a set of managed objects, whose state it may consult of modify.
- A *Management Information Base* (MIB) defines the organization of the managed objects and the data model that allows access to these objects. It is used by both the manager and the agents.

- A *management communication protocol* defines the interaction between the manager and the agents. Typically, a manager may consult or modify the state of a managed object through requests to the appropriate agent. An agent may also asynchronously call a manager to report a change of state or an alarm condition in a managed object.

This model embodies the separation between the administration system and the administered entities. The associated standards have gone through several versions, from a centralized (single manager) to a weakly distributed organization. "Weakly distributed" refers to a hierarchical organization based on a tree of managers (in contrast with a fully distributed organization using a peer to peer interaction scheme).

There is also provision for the integration of legacy systems, through so-called proxy agents, which act as adapters (2.3.3) to translate the requests of the management communication protocol in terms of the appropriate primitives of the legacy system.

This generic model has been used in different contexts: computer networks, equipment management, telecommunication services. It has been specialized to fulfill various specific management functions, such as configuration, security, fault tolerance, observation, performance monitoring, and accounting.

One incarnation of this model is SNMP, the most widely used protocol for network management. The requests are essentially to `get` or to `set` the state of a managed object (e.g., routing parameters, timeout settings, traffic statistics, etc.); an agent may also asynchronously notify a manager (`trap`). The management communication protocol is implemented on top of UDP. The MIB defines a hierarchical organization and a common naming scheme for the managed objects on each node, and the types of the objects, which specify their interface for management related operations.

[Martin-Flatin et al. 1999] surveys distributed network and system management, while [Schönwälder et al. 2003] gives a perspective on the evolution of management technologies for the Internet.

Management Interfaces

A managed object needs to provide an interface for management related operations. As noted above, the principle of separation of concerns calls for an explicit separation between the interfaces dedicated to service provision (functional interfaces) and those used for management (management interfaces).

Typical operations of a management interface are the following.

- Operations related to the life cycle of the managed object: create a new instance (when applicable), remove an instance, start, stop or suspend the activity, etc.
- Operations acting on the state of the managed object: read the value of an attribute, modify the value of an attribute, create or delete an attribute.
- Callback operations (usually asynchronous, e.g., in the form of an interrupt) that signal a change of state or an alarm condition.

If the managed object is a physical device, the management interface is usually provided by a software module that encapsulates the device. The description of the management interface of an object is part of the record that describes the object in the MIB.

Management Domains

The notion of a *domain* has been initially introduced to define areas of protection through shared access rights, and has later been extended as a means of grouping together a set of entities having some common properties. Domains have been introduced in system management [Sloman and Twidle 1994] to group a set of administered resources, be they physical or logical, with the following goals.

- To define common management policies, rules of usage and operations for the objects in the domain.
- To define a naming context (3.1.1) for the objects in the domain (a similar use to that of domains as defined in the Domain Name System).

Various criteria may be selected to define domains, depending of the intended use. A domain may be associated with a geographical location (node, building, etc.). The organization of domains may reflect that of a company or of a department, or the structure of a complex application. A domain may also group all the computers that use a certain operating system, or all printers in a building, etc.

As a consequence, an object may be present in several domains defined with different criteria; thus domains may overlap. A domain may itself be an administered object: for instance, changing the policy attached to a domain *D* is an operation of a domain that administers *D*.

Figure 10.2a shows the user's view of a set of objects and domains. Figure 10.2b shows the implementer's view of the same set. It may be noted that a domain does not actually include objects, but links to objects; remember that a domain acts as a naming context for the objects of this domain.

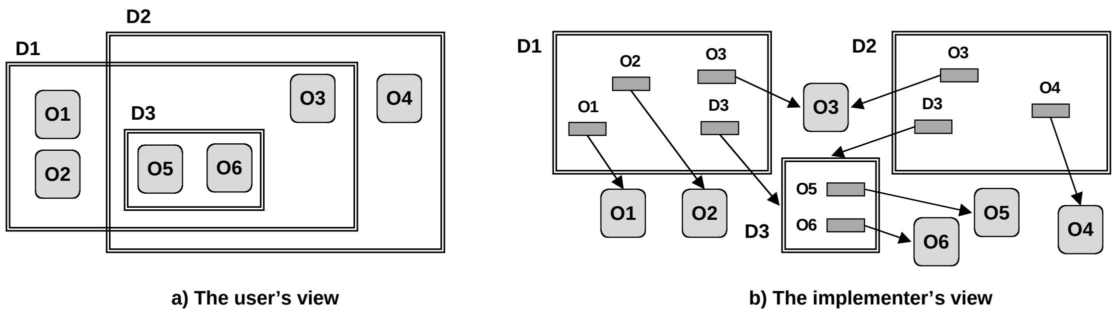


Figure 10.2. Management domains (from [Sloman and Twidle 1994])

This is illustrated by the difference between the status of objects *O3* and *O5* (Figure 10.2). *O3* is shared between domains *D1* and *D2*, i.e., both *D1* and *D2* "see" *O3* through a direct link. This is not the case for object *O5*: both *D1* and *D2* only "see" *O5* through an indirect link through *D3*. The shared entity is *D3*.

Since overlapping domains may share objects (including other domains), there is a possibility of conflict between different policies that apply to a shared object. Such conflicts must be resolved at a higher level, i.e., within a domain administering the conflicting domains.

Domains do not define a specific model for system management, but they may be used as a framework to help organizing the entities defined by such a model. Thus domains may be used in several ways in conjunction with the manager-agent model to organize both the managers and the managed objects.

- Domains may be used to group a set of managed objects according to various criteria (physical location, logical division of applications), or to subdivide this set into subsets of manageable size. Domains may also be used to group objects controlled by an agent. A manager is associated with a domain and implements the policy attached to the domain.

- A hierarchical organization may be defined between managers through a hierarchy of domains associated with different management levels. The policies defined in each domain specify its relationship with other management domains, e.g., as regards security (allowed operations on objects in other domains, etc.).

- To ensure fault tolerance, several managers may be associated with a domain. The collaboration between these managers is organized using one of the methods discussed in Chapter 11.

10.1.3 Basic Functions of Systems Management

A system is an assembly of components (in a broad sense: we only need to assume that the interfaces between the parts are well identified). The first task is to configure and to deploy the system, i.e., to select and assemble the components, and to place them on the physical sites. Further management tasks are done on a running system, and may be performed in three ways.

- By calling specific management interfaces provided by individual components (if applicable).

- By changing the configuration of the system and/or its placement on the sites.

- By acting on the resource allocation to the components.

While operating systems and middleware are usually equipped with interfaces explicitly dedicated to management, this is not always the case for legacy applications. In that case, management "hooks" may be created by interception techniques, at least for such functions as observing the performance of the systems and monitoring the use of its resources. This black-box approach is illustrated, for example, in [Aguilera et al. 2003]. It may apply to a system as a whole and to systems built as an assembly of black box components.

In the rest of this section, we briefly review the main functions related to management.

Observation and Monitoring

Knowing the state and the evolution of a system is a prerequisite for administering it. This includes several aspects.

- information on system configuration;
- information on current resource allocation;
- information on different aspects of load and performance;
- information on system evolution, i.e., on a specific set of events occurring in the system.

This information may be provided in different forms: instantaneous, for example in graphical form, or as a set of records (a log) to be exploited off line, etc. An important aspect of observation is the specification of the information to be collected. This would be used, for example, to set up filters that eliminate the data or the events considered irrelevant.

The term "observation" has a general meaning, referring to any operation of information collection such as described above. The term "monitoring" refers to the observation of a specific function or activity, usually to compare it with an expected or intended behavior, and with the intent of applying corrective measures if needed.

Observation may be active, i.e., initiated by the management system, for instance at fixed intervals, or it may be passive, i.e., triggered by some event on the managed system such as the crossing of a threshold by a monitored variable.

Observation and monitoring are examined in more detail in Section 10.3.

Configuration and Deployment

To produce and to start a working version of an application built as an assembly of components, the following tasks must be performed.

1. selecting the components to be used (some may exist in several versions) and possibly setting initial parameters;
2. verifying the consistency of the system, e.g., as regards dependencies between its components;
3. determining the sites on which the system should be installed, placing each component on the corresponding site, and setting up the connections between the components;
4. starting the components in an appropriate order.

The selection and placement of the components are done according to a chosen policy, which may be expressed in different ways, usually in a declarative form. The policy may directly specify the components and locations to be used, it may express preferences, or it may define higher level goals (e.g., in terms of performance or availability), from which the management system attempts to derive choices that contribute to these goals.

These operations are actually a form of binding (3.3), in which unresolved parameters progressively acquire a value. For example, consider a system made up of three components *A*, *B* and *C* (Figure 10.3). In a first step, a version of each component (e.g., *A1*, *B3* and *C2*) is selected; if necessary, the version is created from a generic template by assigning adequate values to parameters. In the next step, these components are placed on the sites and connected to each other (i.e. bound together). In this example, *A1* and *C2* are placed on site *Site 1* and *B3* on site *Site 2*. Note that *A1* uses an interface provided by *Site 1*, which constrains the placement of *A1* if this interface is not available elsewhere. Finally, the components are activated in a prescribed order, by invoking a specified operation in their interfaces.

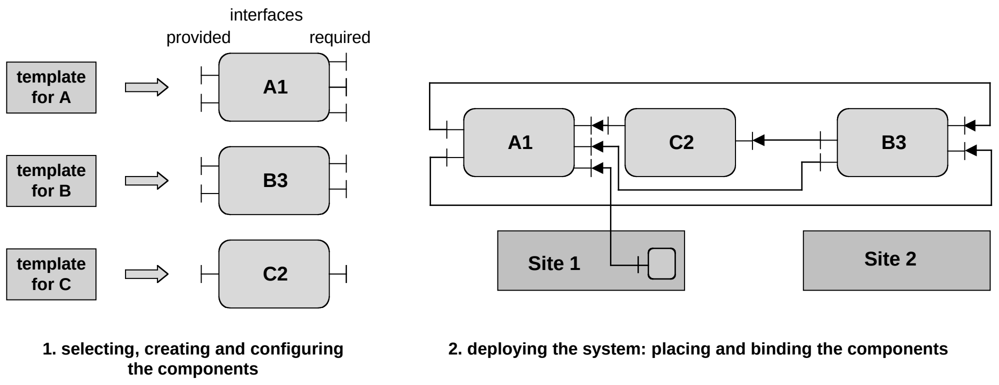

interfaces
provided
required
template for A
A1
template for B
B3
template for C
C2
A1
C2
B3
Site 1
Site 2
1. selecting, creating and configuring the components
2. deploying the system: placing and binding the components

Figure 10.3. Configuration and deployment

One may identify a *configuration* phase (steps 1 and 2) and a *deployment* phase (steps 3 and 4). In step 3, binding and placement may be intermixed (i.e., some of the bindings may be done before placement and some after); some bindings may also be left unresolved at that stage, to be completed dynamically at run time. Step 4 (launching) is subject to ordering rules (e.g., a server must be started before its clients).

Configuration and deployment are examined in more detail in Section 10.4.

Reconfiguration

Reconfiguration is defined as the modification of an existing, deployed configuration. Such a configuration is made up of a set of components residing on a set of sites. Thus the following operations are relevant to reconfiguration.

- changing a component's attributes on a site, using its management interface;
- modifying bindings between existing components;
- adding, removing or replacing a component on a site, which entails modifying the bindings that involve that component;

- moving a component from a site to another one, while maintaining its bindings to other components.

A complex reconfiguration may combine these operations. Reconfiguration may necessitate that the system (or a part of it) be stopped. It also may be performed dynamically, which entails special care to preserve the system's availability for service.

Common examples of reconfiguration, in response to changes in the environment, include:

- replicating a software component on several nodes to accommodate a peak in the system's load;
- moving a server to a new site because of a failure or of a programmed maintenance operation;
- reconfiguring a dynamic network by downloading new programs in the routers.

In the case of mobile systems, i.e., systems whose components are not permanently tied to a specific host or location, the difference between configuration and reconfiguration tends to disappear.

Reconfiguration is examined in more detail in Section 10.5.

Fault Tolerance

A system is subject to a *failure* whenever it deviates from its expected behavior. A failure is the consequence of the system being in an incorrect state; any part of a system's state that fails to meet its specification is defined as an *error*. An error, in turn, may be due to a variety of causes: from human mistake to malfunction of a hardware element to catastrophic event such as flood or fire. Any cause that may lead to an error, and therefore may ultimately occasion a failure, is called a *fault*.

A number of measures may be taken to reduce the probability of faults; however, it is a fact of life that faults cannot be totally eliminated. Therefore systems must be designed to continue providing their service in spite of the occurrence of faults. The relevant factor of quality of service is *availability*, defined as the fraction of time during which a system is ready to provide service. The goal of fault tolerance is to guarantee a specified degree of availability, e.g., 99.999%.

High availability is ensured through two main approaches, which are usually combined: reducing the probability of service interruption, through redundancy; reducing the time to repair a failed system, i.e. to bring it back to a state in which it is ready for service.

Fault tolerance is examined in more detail in Chapter 11.

Resource Management

In order to perform its function, a computing system uses various resources such as processors, main and secondary storage, I/O and network bandwidth. Resource allocation, i.e., the sharing of a common set of resources between applications contending for their use, has a direct impact on the performance of an application, as well as on its cost-efficiency. In the traditional view of a computing system, resource allocation was a task performed by the operating system, and user applications had little control over this process. This situation has changed due to the following causes:

- the increasing number of applications subject to strong constraints in time and space, e.g., embedded systems and applications managing multimedia data;
- the growing variability of the environment and operating conditions of many applications, e.g., those involving mobile communications;
- the trend towards more open systems, and the advent of open middleware.

Thus an increasing part of resource management is being delegated to the upper levels, i.e., to the middleware layers and to the applications themselves, while leaving the lower system layers with the responsibility of fair overall resource sharing between various classes of services. The rationale is that the application is in a better position to know its precise requirements and may dynamically adapt its demands to its needs, thus allowing global resource usage to be optimized, while guaranteeing a better service to each individual application. New interfaces for resource management are therefore needed between the operating system and the applications.

The use of large scale networked systems, such as clusters and grids, adds an additional level of complexity to resource management, introducing the need for new abstractions, both for the resource principals and for the resources themselves.

Resource management is examined in more detail in Chapter 12.

Security

Security is the quality of a system that allows it to resist to malevolent attacks. Such attacks aim at compromising various properties of the system, namely:

- *Confidentiality*: preventing information to be let out to unauthorized parties.
- *Integrity*: ensuring that any modification of information is intended and explicitly authorized.
- *Access rules enforcement*: ensuring that a service is denied to unauthorized parties and actually available to authorized ones.

To ensure these properties, several subgoals are defined, such as:

- *Authentication*: ensuring that a party is actually what it claims to be.
- *Certification*: ensuring that a document, or a proof of identity, is actually valid (i.e., it has not been forged and is not obsolete).

The above properties imply that the various parties, or actors, involved in a system (e.g., people, organizations, or their software delegates) may act as authorities. The role of an *authority* is to deliver authorizations and/or to provide guarantees on other entities. Specifying and implementing security relies on a relation of *trust* between the actors of a system. The notion of a domain (10.1.2) plays an important part as a unit of mutual trust between its member entities.

Security is examined in more detail in Chapter 13.

10.1.4 The Challenges of Management

The increasing complexity of computing infrastructures and the emergence of service-oriented computing with high QoS requirements are pushing the management systems to their limits. We briefly examine three aspects of this challenging situation: the limitations of the current model; the emergence of architecture-based management; the trend towards autonomic computing.

Beyond the Manager-Agent Model

While the manager-agent model has been in use for about 15 years, its limitations are now apparent. The model relies on a centralized architecture, which restricts its ability to scale up, to cope with overload, and to resist failures. In addition, the increased complexity and the dynamic evolution of the managed systems are not well supported by the static, inflexible structure of the manager-agent model. Finally, current systems tend to be organized in multiple heterogeneous management domains associated with different aspects (security, configuration, quality of service, etc.), with independent authorities for policy definition and enforcement. This calls for a decentralized, flexible and reactive management structure.

Another trend is the increasing integration of the worlds of telecommunications, networking, and service provision. Each of these activities has defined its own methods and tools for management. There is a growing effort for a better integration in this area, e.g., using advances in middleware technologies for managing networking infrastructures.

Architecture-based Management

The notion of a system model is taking an increasing importance in the design, the development and the management of software systems. A *system model* is a formal or semi-formal description of a system's organization and operation, which serves as a base for understanding the system and predicting its behavior, as well as for its design and implementation.

In Chapter 7, we have introduced the notion of *system architecture*, as a framework in which a complex system is described as an assembly of elementary parts. We have defined the basic elements of the model that underlies system architecture: components, connectors, and configurations. These entities have different concrete representations according to the specific architectural model being used, but share common properties.

As system architecture is pervading the area of systems design, it has been realized that its constructs also form an adequate base for systems management. In particular, components may be conveniently used as units of deployment, of fault diagnosis, of fault isolation, as well as domains of trust; reconfiguration is adequately represented by component replacement and connector rebinding. The notion of *architecture-based management* captures this trend. It promotes the use of architectural models and formal or semi-formal system descriptions as guidelines for various management functions. Such descriptions are becoming commonly available, as a result of the current efforts to develop model-driven systems architecture (see e.g., [OMG MDA]).

Towards Autonomic Computing

Up to now, administration tasks have mostly been performed by persons. A large fraction of the knowledge needed for administration tasks is not formalized and is part of the administrators' know-how and experience.

As the size and complexity of the systems and applications are increasing, the costs related to administration are taking a major part of the total information processing budgets, and the difficulty of the administration tasks tends to approach the limits of the administrators' skills. As a consequence, there is a trend towards automating (at least in part) the functions related to administration. This is the goal of the so-called *autonomic computing* movement [Kephart and Chess 2003, Ganek and Corbi 2003].

Autonomic computing aims at providing systems and applications with self-management capabilities, including self-configuration (automatic configuration according to a specified policy), self-optimization (continuous performance monitoring), self-healing (detecting defects and failures, and taking corrective actions), and self-protection (taking preventive measures and defending against malicious attacks).

The general principles of autonomic computing are examined in the next section. The following sections develop some of the management functions previously outlined: observation and monitoring, configuration and deployment, reconfiguration.

10.2 Autonomic Computing

.

Currently, human administrators perform management actions to ensure the desired operation of the system, using appropriate tools. Autonomic computing is viewed as an evolution of this practice, along the following lines.

- The overall management goals are expressed at a high level, and their translation into technical terms is performed by the management system.
- The management system observes and monitors the managed system; as mentioned in 10.1.3, the observation may be active or passive.
- On the base of the observation results, the management system takes appropriate steps to ensure that the preset goals are met. This may entail both preventive and defensive actions, and may necessitate some degree of planning.

In a more detailed view, the system is made up of a set of *autonomic elements*, each of which follows the above overall scheme. The organization of such an element is sketched on Figure 10.4. A closer view is presented in 10.2.2.

From this general organization, it appears that autonomic computing is closely related to control. While control methods and tools have been applied to computing systems for a long time to solve specific problems, specially in the area of networking, control theory is only beginning to penetrate the area of systems management. In the next section, we introduce the main concepts of control. We then show how these concepts may be applied to the design of self-managing systems.

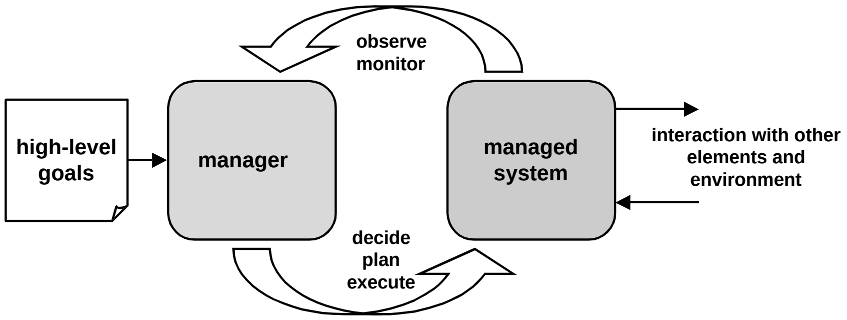


Figure 10.4. Overall view of autonomic computing

10.2.1 Systems Management as a Control Process

Before considering the control aspects of systems management, let us briefly recall the main definitions related to control. Consider a physical system (the controlled system) that evolves in time and is subject to external disturbances. The goal is to control the behavior of this system by making it conformant with a prescribed policy. We need to assume that the system provides an interface (the actuator) that allows its behavior to be influenced through commands. Two main approaches may be taken:

- Open loop (or feed-forward): assuming that the disturbance can be known (or estimated) before it causes the system to deviate from the prescribed behavior, determine the command to apply to the system in order to take a corrective action (Figure 10.5 a).
- Closed loop (or feedback): assuming that some parameters that characterize the behavior of the system may be observed (by means of sensors), determine the command to apply in order to keep the value of these parameters consistent with the prescribed behavior (Figure 10.5 b).

The applicability of the feed-forward approach is limited, since it assumes that the disturbances can be predicted and since it implies that an accurate model of the system is available (in order to determine the action to be taken). The feedback model is more widely applicable, as it relies on weaker assumptions. The two approaches may be combined if some of the disturbances can be predicted, and if the corresponding corrective actions can be determined, at least approximately. The measurements and commands may take place at discrete instants in time, or may be continuous functions of time. The vast majority of applications of control to computer systems are based on a discrete feedback model.

The main advantage of the feedback approach is that it does not require a detailed model of the controlled system, although it may benefit from the existence of such a model, which may be either derived from a formal analysis or determined experimentally by identification methods. The main risk associated with feedback control is instability.

The controller's task is to fulfill an objective, which may take several forms [Diao et al. 2005].

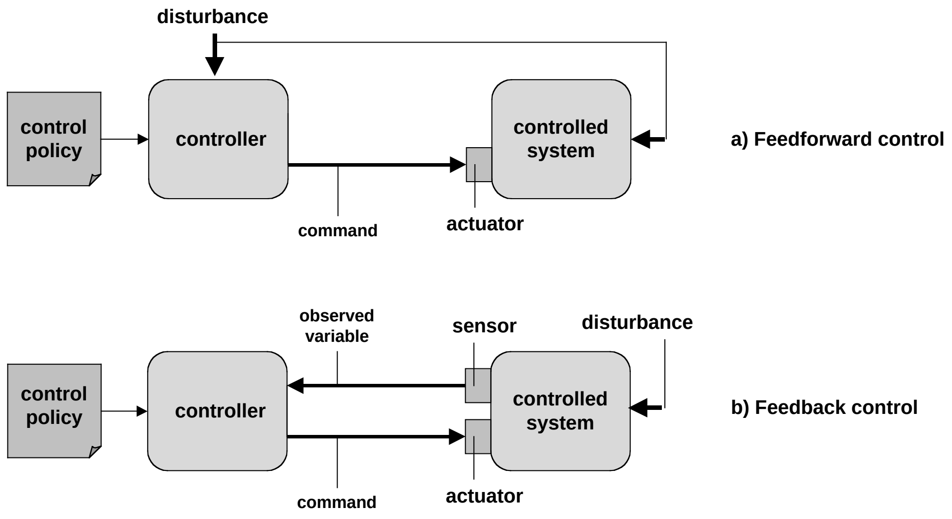


Figure 10.5. Two approaches to control

- Regulatory control: ensure that the measured output is equal (or close) to a reference input. For example, a specified value may be set for the utilization rate of a processor. Other examples may be found in Chapter 12.

- Disturbance rejection: ensure that disturbances acting on the system do not significantly affect the measured output.

- Optimization: ensure that the measured output is equal (or close) to an optimal value (with an appropriate definition of "optimal", e.g., minimal average response time, maximal use of a resource, etc.).

The system's operation is subject to some constraints, which are related to the quality of the control. The most important constraint is *stability*, i.e., bounded inputs should always produce bounded outputs. Other desirable properties include accuracy (closeness of the measured output to the reference input), short settling time (quick convergence), and absence of overshoot (e.g., accurate response to changes in the reference input).

Detailed examples of the use of feedback control for fault tolerance and for resource management may be found in Chapters 11 and 12, respectively. In Section 10.2.3, we examine the principles that govern the use of feedback control for self-managing systems.

10.2.2 Architectural Aspects of Autonomic Computing

Following the architecture-based management approach, we now examine in more detail the organization of an autonomic element, the building block of autonomic systems. An autonomic element implements a control loop that regulates a part of the system, which we call a *managed element.* A managed element (ME) may consist of a single elementary hardware or software component, or may be a complex system in itself, such as a cluster of machines, or a middleware system. In order to be included in a control loop, a managed element must provide a management interface, which includes a sensor interface and an actuator interface[2]. These are used by a controller, also called an *autonomic manager* (AM) to regulate the managed element through a feedback control loop. The *autonomic element* (AE) is the ensemble including the managed element and the control loop, i.e. the controller and the communication system that it uses to access the management interface (Figure 10.6).

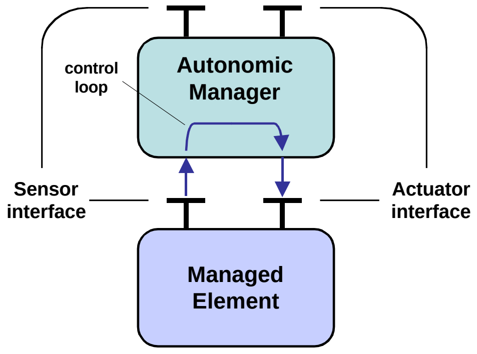


Figure 10.6. An autonomic element

An autonomic manager may itself be equipped with a management interface, thus becoming in effect a managed element. Thus allows a hierarchical organization of AMs. In the same vein, an elementary managed element, such a hardware device like a disk unit, may itself include embedded, built-in control loops, making it an autonomic element, even if these control loops are not directly accessible through the ME's management interface.

The controller that regulates a ME in an autonomic element usually deals with a single control aspect, e.g., security, fault tolerance or performance. A given ME may then be part of several AEs, each of which deals with a specific aspect; each of these AEs has a specific AM and may be regarded as a different management domain, in the sense defined in 10.1.2 (Figure 10.7 (a)).

The AMs managing different aspects of a common element have different criteria and may take conflicting decisions. An approach to resolving such conflicts is to coordinate the AMs that manage different aspects through a new AM (the coordinator), which applies a conflict management policy (Figure 10.7 (b)). An example of such a policy might be as follows, to arbitrate between a repair manager and a performance optimizing manager: give priority to the repair manager, except when this would degrade the quality of service of a specified client. The coordination of multiple autonomic managers is examined in [Cheng et al. 2004]. An overview of the architectural aspects of autonomic computing may be found in [IBM 2003].

The notion of an autonomic element is thus seen to cover a wide range of situations, at different levels of granularity. In the above discussion, we came across three typical levels, in increasing order of abstraction:

- An elementary component, with embedded internal control.

[2]The complexity of these interfaces depends on that of the managed element.

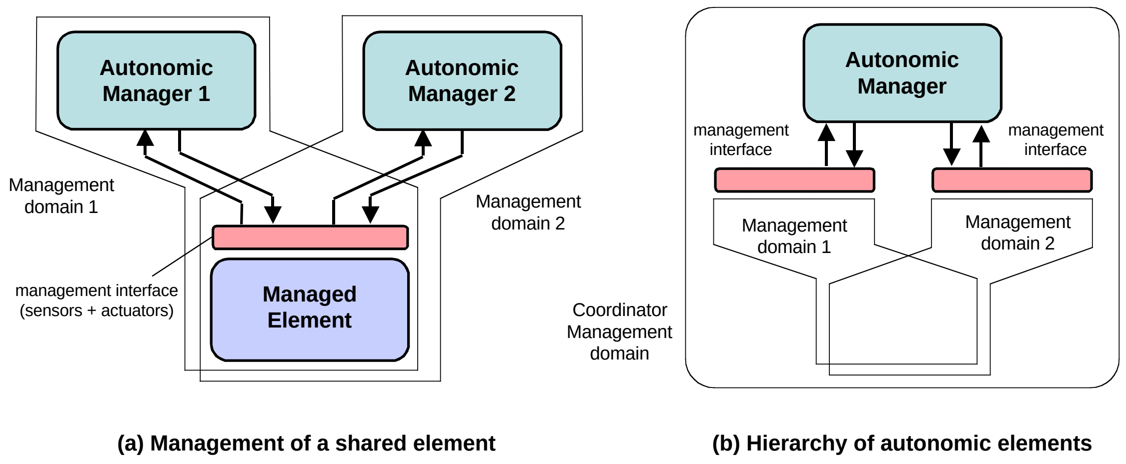


Figure 10.7. Multiple domains for autonomic management

- A mid-level manager, at the application or middleware level, controlling an aspect such as QoS, security, or fault tolerance.
- A coordination manager, whose role is to arbitrate between several aspect-specific managers in charge of a common set of resources (a shared ME).

As the level of abstraction increases, so does the mean period of sampling, which determines the time scale of the feedback control loop.

A managed element, be it elementary on complex, needs to provide a management interface, including sensors and actuators. There is currently no standard defining this management interface, and the existing experimental platforms have developed their own set of tools (see e.g., [Jacob et al. 2004]). When dealing with legacy software, not initially intended to be externally managed, a common solution for providing a management interface is to use a wrapper (2.3.3). The function of the wrapper is to make a bi-directional translation between the interface provided by the legacy components and the specified management interface. This may be a best effort attempt, depending on the available interfaces of the legacy software.

More details on the management interface may be found in the next section.

10.2.3 Principles and Patterns of Self-Management

The main elements of a feedback control loop used for managing an autonomic element or system are the controller, the sensor, and the actuator. While autonomic management applies to systems with widely varying characteristics, it is possible to identify a few common patterns for these three elements. We present them in turn.

Controller Design and System Modeling. The organization initially proposed by IBM [Kephart and Chess 2003] for autonomic managers is the so-called MAPE-K (Monitor, Analyze, Plan, Execute - Knowledge) model, which may be described as follows.

- Monitor. Collect data that characterize the behavior of the managed system.

- Analyze. Interpret the data, using knowledge about the managed system.
- Plan. Determine a course of action, based on the results of the analysis, using a specified algorithm or heuristic.
- Execute. Implement the plan, by sending commands to the managed system's actuators.

The "knowledge" includes a model of the managed system's behavior, which may take a a variety of forms, as indicated below.

In the Monitor and Execute phases, the manager communicates with the managed element by interacting with the sensors and actuators, respectively. More details on this interaction are provided later.

The Analyze and Plan phases are the central part of the manager. They use the model embodied in the "knowledge" to interpret the inputs and to determine the reaction according to the chosen strategy (which may itself be adapted). Several approaches may be used for the design of the manager, depending on the accuracy of the model, as discussed in [Diao et al. 2005]:

- A purely empirical approach, relying on a primitive model, which does not embody any knowledge of the managed system's operation. This may be quite effective, as exemplified by the early mechanisms used for load leveling in multiprogrammed operating systems (12.4).
- An approach based on a a black box model. The managed system is considered as a black box, assuming to follow a known behavior law (e.g., linear, time-invariant). The numerical values of its parameters are determined by identification (measuring its response to a set of inputs). Then the model is used to drive the actuators, based on the sensors' input. An example of the use of this technique may be found in 12.5.4.
- An approach based on queueing theory, in which the system is represented by a network of queues. This class of models is specially useful for performance analysis and control. An example of the use of this technique may be found in 12.5.5.
- An approach based on the detailed knowledge of the managed system's structure and internal operation, allowing one to build a specific model. An example of the use of this technique may be found in 12.5.6.

General methods for system adaptation (2.4) are of course applicable. In particular, a managed element may be reflective, i.e., maintain a causally connected representation of itself, and provide primitives for introspection and intercession, which may be included in the management interface and used as sensors and actuators, respectively.

Sensors and Actuators. Sensors and actuators are the main facets of the management interface provided by a managed element (other facets include, for example, management of attributes that are not relevant to the autonomic control aspect).

Sensors and actuators have widely varying characteristics, since they interact with different specific aspects of the systems they manage, which themselves use a variety of hardware and software technologies. One way to deal with this heterogeneity is to abstract a set of common functions in a generic, uniform interface, and to specialize the implementations of this interface to accommodate specific technologies and needs. This approach is illustrated in some of the examples that follow.

Two other aspects need to be considered when designing sensors and actuators.

- Deployment of sensors and actuators. Sensors and actuators may be either statically inserted or dynamically deployed in the managed element. Dynamic deployment offers more flexibility, but poses problems similar to those of reconfiguration (10.5).
- Communication modes. There are two different modes of communication, depending on which part the initiative lies. Both modes may be used in a given framework.

  Sensors may periodically send their results to the *Monitor* module (active mode), or that module may consult the sensor when data are needed (passive mode). In the case of actuators, the *Execute* module may directly activate the actuators (direct mode), and an actuator may also callback *Execute* to report completion and to request further instructions (callback mode). See Figure 10.8.

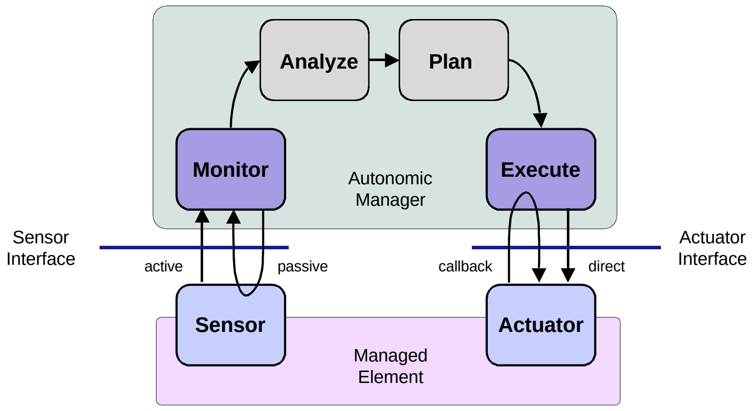


Figure 10.8. Interfaces for sensors and actuators

Actuators are the subject of the next subsection. Sensors are examined in Section 10.3.

10.2.4 Frameworks for Actuators

Several proposals have been made to specify a generic framework for actuators. Such a framework is typically defined as a set of classes that may be specialized. We briefly present three examples that illustrate this approach.

Fractal. In the Fractal component model (7.6) each component is equipped with a management interface, in the form of an extensible set of controllers. This interface may be used as a base for the implementation of actuators that have an effect on the component. For instance, if the component is composite, the *ContentController* allows new components to be instantiated within the component, and the *BindingController* allows changing the bindings between the included components. Thus an actuator for reconfiguration is directly available. Any other form of actuator may be specified either as a new controller, or as an extension of an existing controller.

IBM Autonomic Management Engine. In the IBM Autonomic Management Engine (AME), part of the Autonomic Computing Toolkit [Jacob et al. 2004], any specific actuator code that performs an action on a managed element is embedded in a wrapper, called an *action launcher*, which presents a uniform Java interface, `ActionLauncher`, to an autonomic manager. This interface allows the manager to trigger a specific action, by enabling a predefined condition associated with that action. The set of condition-action associations is established by the manager through an operation also provided by the interface of the action launcher.

KX. KX (short for Kinesthetics eXtreme) [Valetto et al. 2005] also provides a set of generic primitives that form the interface between a manager and a managed element. The underlying model is based on the observation that the interaction of a manager with an actuator on a managed element may be decomposed in a set of generic operations: instantiate an actuator, or bind to a pre-existing instance; pass parameters to it, to configure it for a specific condition; invoke it to effect the action; and report back the results, if needed. The generic (Java) API provided to a manager includes primitives for all of the above operations.

Wrapping techniques are needed to use this generic API with a plug-in that implements a specific actuator. In addition, the methods of the generic API may be specialized in order to provide various interaction modes between the manager and the actuator (e.g. event-based, synchronous, using native Java invocation or mobile code, etc.).

10.2.5 Conclusion

Autonomic computing, in 2005, is more a vision of the future than an industrial reality. Several large-scale initiatives have been launched by the industry to meet the challenge of complex systems management, and autonomic computing has become an active area of research, with a number of prototypes being developed and evaluated.

As the effort is in its early stages, no standards have yet emerged. However, incremental progress is being made in the understanding of the fundamental problems, in the elicitation of relevant patterns, and in the introduction of autonomic features into working systems. Some examples may be found in the following chapters.

Autonomic computing is a multi-disciplinary area, which draws on control theory, resource management and optimization theory, artificial intelligence, man-machine interaction, and systems architecture. A survey of the main research challenges of autonomic computing is presented in [Kephart 2005].

10.3 Observation and Monitoring

10.3.1 Requirements for Observation and Monitoring

The requirements for distributed systems observation are similar to those applying to measurement in general, with a few specific adaptations.

Minimal intrusiveness. The observation devices should minimally interfere with the observed system. In particular,

- The semantics of the observed system should be preserved. Thus the process of observation may cause additional events to occur, but it should not alter the order of application-related events.
- The overhead caused by the observation should be kept to a minimum. If the observed system is time-critical, the observation should not cause it to violate its time-related specifications, e.g. to miss a deadline.

Accuracy. An observation should deliver accurate results. Accuracy applies to the physical quantities being measured (e.g., execution time, memory occupation, number of requests executed per unit time, etc.), and may be specified in absolute or in relative terms.

Flexibility of use. It should be possible to dynamically reconfigure the observation system during operation, for instance to change its parameters (e.g. focus, selectivity, or sampling period), or to concentrate on a different part of the observed system. It should be possible to dynamically insert, start, stop and remove observation devices, without having to restart or to recompile the observed system.

If the observed elements consist of events rather than physical quantities, it should be possible to define various “objects of interest”, e.g., composite events (combinations of primitive events), according to specified event composition patterns.

Scalability. The observation system should be as scalable as the observed system. If the latter grows, so should the former.

Multi-level interpretation. The results of observation may be interpreted at various levels, corresponding to levels of abstraction of the observed system. While raw measurements may be collected at a low level, they may be processed (e.g., by smoothing, integration, aggregation) to reflect a state corresponding to a higher level view of the observed system.

10.3.2 Patterns for Observation and Monitoring

The current approach to observation patterns relies on the principle of separation of concerns, which leads to the following design rules.

- Multi-level interpretation is achieved through a layered design, which separates devices for low-level data collection (the *probes*) from devices for data integration at various levels (the *gauges*). Generic interfaces are provided at all levels. A communication infrastructure (e.g., an event bus) is used to propagate information between the levels.
- The administration of the observation devices, i.e., their instantiation, configuration and deployment, is separated from the process of data collection.

We now examine these aspects in more detail. The material on probes and gauges is inspired by the work of the DARPA DASADA program [DASADA], which developed standards for adaptable distributed systems with special attention to system observation.

Probes

A *probe* is a sensor attached to a running program, in order to collect information about the execution of that program. Probes are characterized by a number of properties:

- The nature of the information being collected: CPU consumption, memory use, I/O rate, number of operating system calls, etc., either at a specific point in time or over some period.
- The time at which the probe is attached to the observed program: compile time, load time, run time.
- The way in which the collected information is propagated: an active probe sends events (periodically, or when some observed value crosses a threshold), while a passive probe responds to an explicit request.

Dynamic probe insertion (i.e., without recompiling or restarting the program) is more complex that compile time or load time insertion, but allows more flexibility, specially when the observed system is itself subject to dynamic reconfiguration. In addition, the interesting observation points are not always known in advance. Dynamic probe insertion allows "exploratory probing", i.e., determining the information to collect and the observation points by trial and error on a running system.

The life-cycle of a dynamically inserted probe may be described by the following operations, which trigger state transitions:

- *Deploy*: instantiate the probe on the host that supports the observed program.
- *Install*: attach (or bind) the probe to the observed program.
- *Activate*: enable the probe to register the observed data, according to its specification.

Inverse commands (*Undeploy*, *Uninstall*, *Deactivate*) reverse the effect of the above commands. Once activated, a probe may either send events or respond to queries, according to whether it it active or passive.

These commands make up a generic interface, and may be adapted to the specific form of an existing probe by means of a wrapper.

Gauges

Gauges are software devices used to deliver a higher level view of measurement data. Thus a gauge gathers, aggregates and analyzes the raw data provided by probes. A gauge is associated with a model of the measured system, and the results it provides are only significant with respect to that model. As a consequence, several gauges may be used to process a given set of raw measurement data, in accordance with different views of the measurement system, e.g., at various levels of detail. Since they operate at a separate level, gauges may be decoupled both from the probe infrastructure (which operates at a lower level) and from the transport mechanism.

An example of use of gauges for architecture-based monitoring is given in [Garlan et al. 2001]. In this approach, gauges are linked to architectural descriptions, i.e., a gauge delivers data that are significant in terms of an architectural model of the observed system (e.g., throughput on a link between components, mean response time for a method call at a component's interface, etc.). The following simple example illustrates the difference between a probe and a gauge in a client-server system: a probe measures the bit rate on a connection, while a gauge gives a throughput in terms of file transfer, and may thus be used to estimate the transfer time of a file as a function of its size.

Linking gauges to an architectural description opens interesting perspectives, such as integrating gauge descriptions into an architecture description language (ADL) and automatically generating and deploying gauges, using such descriptions. An example of such facilities may be found in [Garlan et al. 2001].

Event Dissemination

The main aspects of event dissemination are presented in 6.3. The main requirements are scalability, robustness, and manageability. These properties may be achieved through a decentralized organization, as may be seen in the examples presented in the next section.

10.3.3 Frameworks for Observation and Monitoring

We illustrate the notions on observations presented above with two examples.

Astrolabe. The Astrolabe system [van Renesse et al. 2003] combines a set of distributed gauges (data aggregation functions) with an event transport system. The function of Astrolabe is to monitor the dynamically changing state of a collection of distributed resources, reporting summaries of this information to its users. A typical example of a system that may be monitored with Astrolabe is a distributed collection of data centers hosting Web services applications, together with a set of client computers using these applications over a network. The results of the observation may be used to gather statistics on the use of the system, or as an input to autonomic managers in charge of adapting the system to a changing load.

The resources observed by Astrolabe are organized in a hierarchy of domains (10.1.2), called zones in Astrolabe. A zone is recursively defined to be either a host (a single machine) or a set of non-overlapping zones. A list of attributes is associated with each zone; this list may be seen as a form of a MIB (10.1.2). This MIB has different forms for leaf zones (consisting of a single host) and for non-leaf zones. The MIB of a leaf zone is written by local probes attached to systems or applications running on the host; it may represent the data associated with a given computer, with a specific application, etc. The MIB of a non-leaf zone is generated by aggregation functions (the zone's gauges) on the MIBs of the zone's children. A zone is designated by a path name, much like in a hierarchical file system. This organization is described on Figure 10.9.

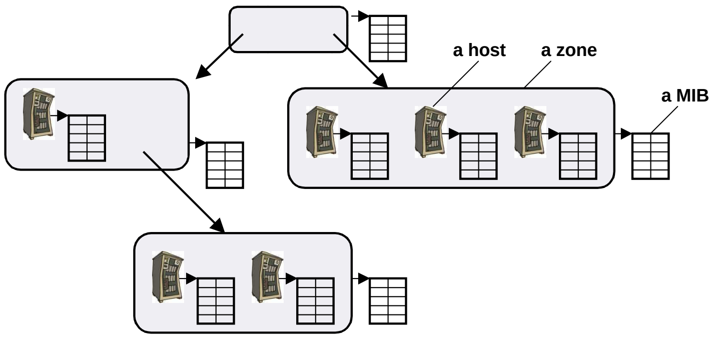


Figure 10.9. An example of an Astrolabe zone tree (adapted from [van Renesse et al. 2003])

The attributes' values collected in the MIBs of the leaf zones may be highly dynamic, and their variations must be propagated as fast as possible to be processed by the gauges. This propagation uses a randomized epidemic protocol: each host runs a process called an Astrolabe *agent*, whose function is to propagate information to other agents. Each agent picks some other agent at random and exchanges state information with it. The information relates to the zone that is the least ancestor of the agents' zones (or their common zone if they are in the same zone).

For efficiency, each zone keeps a copy of the MIBs of the zones along that zone's path to the root, and the MIBs of the siblings of those zones. At a given time, some copies may be out of date, but the copy update protocol guarantees *eventual consistency*: if no update occurs for a "sufficiently long" period, the values of the copies will become identical.

The gauges in Astrolabe implement aggregation, using data mining and data fusion. These operations are performed continuously, using the flow of propagated data as an input. Astrolabe may thus be seen as a large, hierarchically organized, relational database, updated using the gossip protocol described above. The tuples in the database are the MIBs described above.

Aggregation functions are programmable. The code of these functions (e.g., expressed as SQL queries) is mobile, and may thus be sent to a zone to modify its behavior. For security, the code is embedded in signed and timestamped certificates, which are installed in the MIBs as specific attributes.

Simulation experiments (reported in [van Renesse et al. 2003]) have shown that the information dissemination latency grows logarithmically with the number of hosts. This scalability is achieved through the hierarchical zone structure and through the gossip protocol.

Ganglia. Ganglia [Massie et al. 2004] is a scalable distributed monitoring system for large high performance computing systems such as clusters and grids. The main challenge is achieving scalability, for the system itself as well as for the monitoring system. Other desirable properties include manageability, extensibility, and robustness.

Like in Astrolabe, scalability is achieved through a hierarchical design. The monitored system is a federation of clusters, each of which is composed of a set of interconnected nodes. Ganglia is based on a two-level organization (Figure 10.10).

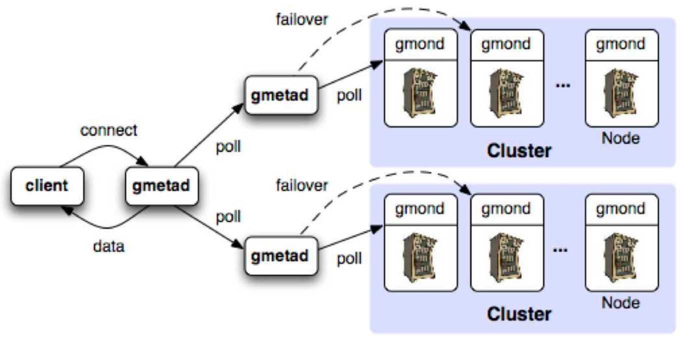


Figure 10.10. Organization of the Ganglia monitoring system (adapted from [Massie et al. 2004])

Within a cluster, Ganglia uses heartbeat messages (Chapter 11:) on a well-known multicast address as the basis for a membership protocol (detection of failed nodes). This multicast address is also used by the monitoring system to send monitoring data collected by the local probes on each node. Local probes may be attached to the local resources as well as to applications (the origin of the collected data is marked by a specific field in the multicast packets). Al nodes listen to both types of metrics on the well-known multicast address; thus all nodes have an approximate view of the state of the entire cluster, which allows this state to be reconstructed after a partial crash.

The implementation of Ganglia relies on two daemons, `gmond` and `gmetad`. The Ganglia monitoring daemon, `gmond`, runs on each node of a cluster and implements intra-cluster monitoring, using the announce/listen protocol on the well-known address. This daemon responds to client request by returning a standard representation of its monitoring data (encoded in XML). The Ganglia meta daemon, `gmetad`, acts as a gauge, by aggregating information coming from the federated clusters.

10.4 Configuration and Deployment

One consequence of the increasing role played by system models, as noted in [van der Hoek et al. 1998], is the emergence of a close relationship between the areas of software architecture and configuration management, which have initially evolved separately. The representation of a system in terms of a set of interacting components is the common base for methods and tools in both areas. We acknowledge this trend by taking an architecture-based approach to configuration and deployment.

In this section, we first introduce the main notions related to configuration and deployment, and the deployment life-cycle. We then present the principles of the architecture-based approach, and the related emerging patterns. We conclude by illustrating these patterns with typical configuration and deployment frameworks.

The current approach to reconfiguration also makes use of architectural models, and often extends the notions present in configuration frameworks. Reconfiguration is presented in a separate section (10.5)

10.4.1 Definitions

As defined by IEEE-Std-610, "Configuration Management is a discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements". Thus configuration management controls the evolution of software during its development phase. A software product must be allowed to exist in a number of versions to accommodate a variety of needs and environments. Configuration management tools must provide a means to archive and to identify software components in multiple versions, to specify configurations as assemblies of well-identified components to fulfill a specific purpose, and to enforce specified policies with respect to consistency, completeness, sharing and security.

Deployment, in the restricted sense that we use in this chapter, is the process of installing and starting a specific instance of a software product; in a wider sense, this term covers the whole part of a product's life-cycle that follows development, thus including update, adaptation and reconfiguration.

Deployment was initially regarded as a simple extension of configuration management and was not considered a respectable subject of study. Deployment tools were built ad hoc, in the form of installation scripts using low-level component identities such as file names; coordination between scripts on different machines was often done manually. As a result, inconsistencies due to configuration and deployment errors came to be identified as one of the main causes of failures in large scale distributed services [Oppenheimer et al. 2003]. The increasing size and complexity of both hardware platforms and software systems, and the need for constant evolution are bringing about a new generation of tools based on sound design principles and using higher level, more explicit, system representations.

The next section describes the main operations related to the various steps of configuration and deployment.

10.4.2 Configuration and Deployment Life-cycle

The following elements are common to all deployment frameworks.

- The *software product* to be deployed (the term "product" is generic and covers anything from basic operating system to end-user application). The parts that make up the product may be provided in various formats (source language, relocatable modules, executable units), and may exist in a number of versions.

- The hardware platform, called the *target*, on which the product is to be deployed. The target usually comprises a set of machines connected by a network. Some initial software (e.g. operating systems, router software, common utilities, etc.) may be pre-installed on various parts of the target.

- A *repository* containing the elements of the product to be deployed. The repository may be located on a central site, which may or not be part of the target platform; or it may be distributed on the sites of the target platform. Regardless of the physical setup, the important point is the logical distinction between the repository and the target.

An overall view of the whole life-cycle of a product is given in Figure 10.11. It shows two main phases. Phase 1 is the configuration process. Successive versions of the product's components are released for installation, and made available in the repository. The product can then be configured, i.e., parameters may be set, alternate versions may be selected, etc. Obsolete versions are retired. Phase 2 is the deployment process proper, in which the components of the product are installed on the target and the various bindings are established (the bindings may involve components of the product as well as elements present on the target). In the extended view of deployment, this phase also includes the update and reconfiguration operations that may take place during the later life of the product. Thus one needs to express the invariant conditions that a product must satisfy in order to remain consistent after a change, and to develop methods and tools that ensure consistency preservation.

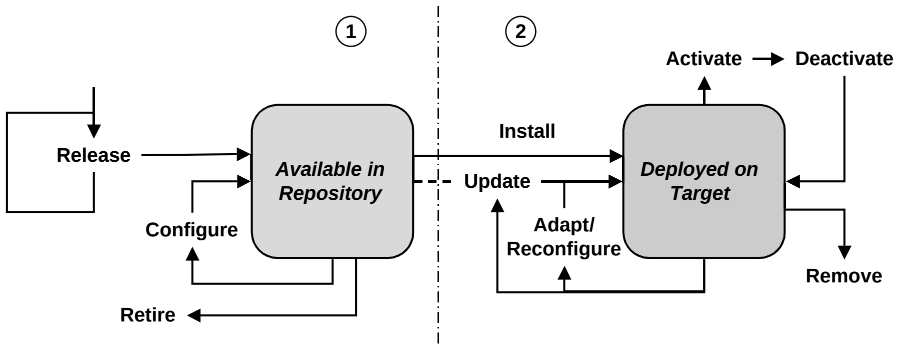


Figure 10.11. Configuration and deployment life-cycle

This separation into phases is convenient for presentation, but not strictly enforced in practice. Thus some operations pertaining to configuration (i.e., selecting component versions, setting up parameters) may also take place at installation.

Let us define the operations that appear on Figure 10.11. These definitions are adapted from [Hall et al. 1999], [Hall 2004], to which we refer for further details and examples of deployment frameworks.

- The ***release*** and ***retire*** operations are related to software development. Releasing a component is making it available for further use as an element of a software assembly. A component may be released in different forms (e.g., source code, intermediate code, loadable binary for a specific architecture, etc.). It may be packaged, i.e., put into a specific format usable by code management and installation tools (e.g. `.jar` archives, RPM modules, etc.). Retiring a component makes it unavailable for further use and support (however, installed instances of the component may continue being used).

- The ***configure*** operation covers any form of variability that may be applied before installation (e.g., selecting a specific component version, setting a compile-time option, etc.).
- The ***install*** operation performs the main deployment action. It sets up a product as an assembly of selected components on the target platform, executing any required bindings. After installation, the product is ready for execution. Execution is launched by the ***activate*** operation, which starts the appropriate entry points at the various sites, in a prescribed order. Execution is stopped by the ***deactivate*** operation.

  The ***remove*** operation (also called ***uninstall***) undoes the action of ***install***. It should therefore restore the target platform into a consistent state (this specially applies to shared resources such as libraries).
- ***adapt***, ***update*** and ***reconfigure*** are performed after installation. ***adapt*** denotes a generic form of reaction to a change detected in the system's environment. ***Reconfigure*** enacts a change in the product's composition or organization, which may be triggered by ***adapt*** or requested by a user (see 10.1.3; details in 10.5). ***update*** is a special case of reconfiguration which consists in replacing or adding newly developed components.

Note that there is no universal agreement on the above terminology. For instance, in the OMG configuration and deployment model [OMG C&D], "installation" is the loading of software into a repository and the "preparation" and "launching" phases cover what we call "installation".

10.4.3 Requirements and Critical Issues

We now discuss the main requirements and problems that are common to configuration and deployment.

Requirements

The raison d'être of configuration management is to allow variability. In the context of software products, *variability* is the ability to modify a system's behavior according to evolving needs. This modification may be done at various steps in the system's life-cycle: changing design decisions, setting compile time options or configuration files, dynamically linking new plug-ins. Variability may occur both in space (generating various versions of a given system for different operating environments) and in time (evolving a product to adapt to changing requirements). Variability is discussed in [van Gurp et al. 2001], [van der Hoek 2004].

The main requirement associated with variability is flexibility, which applies to several aspects, namely:

- the ability to apply changes at any point of the product's life-cycle;

- the ability to delay change up to the latest possible stage;
- the ability to use any policy for change management, and to change policies during the product's life;
- the ability to have several versions of a given component coexist in a product.

Depending on the specific environment, these requirements may or not apply, and may have different priorities.

Critical Issues

Configuring and deploying a large distributed system is itself a complex process, which entails the issues of scalability and fault tolerance. In addition, meeting the requirements listed above raises two specific issues: identification and consistency.

The first problem is *keeping track of the multiplicity of possible configurations of a product*. This turns out to be surprisingly hard, for several reasons. Traditional system configuration management keep track of versions or variants for components or for entire products, but do not usually manage architectural descriptions at a high level. This makes it difficult to identify the different variants of a system deriving from changes in its global architecture, because such changes are not adequately reflected in the system's identification. Another problem stems from ambiguous or over-constrained designation: for example an identifier for a library routine may refer to different versions depending on the library vendor; or a routine is only identified by its absolute location in the file system, which prevents different versions of it from coexisting.

The second, most important problem, is *maintaining consistency* in the face of change. A complex application exhibits many dependencies, both internal (between its components) and external (between the application and its environment, or between different applications). Frequent problems stem from the fact that these dependencies are not always explicitly stated in a product's description. Thus an apparently innocuous change may prevent an application from working because of its hidden reliance on a specific version of some library routine; if this routine is updated to correct the problem, this may cause another application to fail if different variants are not allowed to coexist, etc. Also the dependencies of a product may vary according to the binding time (e.g., the dependency of a product on a library has different implications whether the library is statically or dynamically linked).

These common issues suggest taking a unified approach to configuration and deployment [van der Hoek 2001], since changes made at one of these phases eventually need to be reflected in the other one. The need for an explicit statement of all dependencies calls for a unified representation of a system's structure throughout its life-cycle, which is a further argument for an architecture-driven approach, the subject of the next section.

10.4.4 Architecture-based Deployment

The main idea of architecture-based deployment is to use an architectural model of a system as a guide for the description, the construction, and the deployment of system configurations. Recall (7.3) that an architectural model describes the system in terms of entities such as components and connectors, whose precise definition depends on the model. Component models have been introduced in 7.4. For the general discussion of deployment, we do not need specific assumptions on the component model. For the examples, assumptions are specified as needed.

Principles

Architecture-based deployment follows two main principles.

- *The configuration of a system and its deployment parameters are described using the elements of the system's architecture. This description is separate from the code of the system.*
- *The description of a system's configuration is used as a base for automating the process of configuration and deployment.*

Note that these principles are yet another instance of separation of concerns (1.4.2). All information related to configuration or placement (e.g., which version of a particular component should be used, on which machine it should run) is taken out of the code of the application. On the other hand, the configuration description is not tied to a specific mode of use (e.g., it may be compiled or interpreted, depending on the run-time system being used).

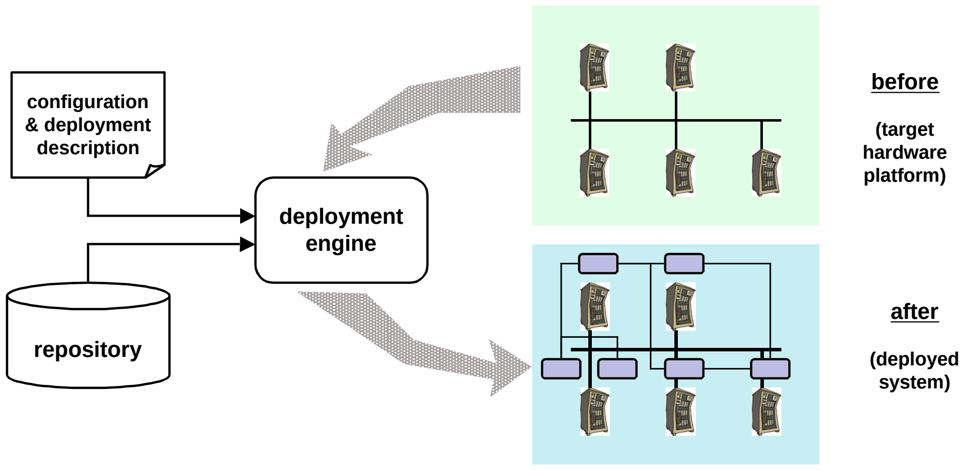

configuration & deployment description
deployment engine
repository
before
(target hardware platform)
after
(deployed system)

Figure 10.12. An overview of the deployment process

Figure 10.12 gives a global picture of the process of architecture-based deployment. It shows the repository and the target platform, introduced previously, and two additional elements: the configuration and deployment description, and the *deployment engine*. The engine carries out the configuration and deployment process, as specified by the description. The result of this process is a running instance of the installed system, on the provided target platform.

This is a very high-level view that makes no assumptions on the actual organization and operation of the whole process, whose specific details may take a number of concrete forms :

- The format of the description is not specified at this point. It usually takes a declarative form, which may be based on an architecture description language (ADL), possibly extended with constructions that capture specific notions related to configuration and deployment. The description is usually stored in the repository, although it is represented separately on the figure.
- Like the repository, the deployment engine may be either centralized or distributed on the sites of the target platform. The engine may use the description in several ways: it may interpret it to control the configuration and deployment process; or it may take it as an input to generate code that performs configuration and deployment.

Defining models and patterns for the configuration and deployment of component-based systems is an active research area, whose results are not yet stabilized. Therefore, the material that we present in the rest of this section is still in a tentative form. We examine in turn the repository, the configuration description, and the deployment engine.

Deployment Repository

As indicated, the repository may take different physical forms. Here we concentrate on its logical organization, which is defined by the following elements: the units for the various functions to be performed; the identification scheme; the expression of dependencies. Information related to these functions is stored as meta-data in the repository. We examine these in turn.

We first consider the deployment unit, i.e., the minimal item that may be individually deployed. In the absence of a standard terminology, we use the term *unit* for "deployment unit". The requirements for units, derived from those in 10.4.3, are quite similar to those for components (Chapter 7), namely:

- A unit must be uniquely identified; if a versioning scheme is used, each individual variant must have a different identifier.
- A unit must explicitly define its dependencies; as a consequence, units must contain explicit export and import lists. These lists must identify provided and required items associated with a unit's identity.
- Units must be composable, i.e., it should be possible to combine several units into a single one, by matching provided and required items; the compound unit should be usable exactly as a primitive (non compound) one. For flexibility, a hierarchical composition scheme should be preferred to a flat one.

The problem of installing a set of units satisfying specified dependency constraints has proven surprisingly difficult. In fact, it has been shown [Di Cosmo et al. 2006] that this problem is NP-complete in the general case. In current situations, however, the problem is tractable, but requires a careful specification of the dependencies and the use of elaborate tools such as constraint solvers. Examples may be found in section 10.4.5 (the Nix system) and in [Boender et al. 2006].

In practice, deployment units are also units for identification, for storage, and for packaging. Packaging a unit is setting its contents (code and meta-data) in a standard representation, which may be shipped between sites and used by appropriate tools. Note however that the unit for code sharing is distinct from the deployment unit. Sharing may apply to subparts of deployment units (e.g., a common library may be shared, but not independently deployed).

Units appear under various forms and names in different system, and do not always satisfy the above requirements. Examples are Java archives (*.jar* files) in Java based applications, bundles in OSGi, components in various component-based systems, packages in software distribution schemes such as RPM (see examples in 10.4.5).

Deployment Description Language

The function of a deployment description language is to specify the goals and constraints that apply to the deployment process.

As mentioned in 10.4.3, we suggest taking a unified approach to configuration and deployment. This applies to deployment descriptions, as illustrated by a simple example. Consider a component *C* that may exist in two versions *C1* and *C2*, which provide the same interface to their users but require two different versions of a library, say *L1* and *L2*, respectively. If *L1* is only available on host *H1* and *L2* on host *H2*, then component *C* will have to be deployed on different hosts according to the version in use. As a consequence, current deployment descriptions usually mix configuration and deployment information.

A deployment language should specify the following points:

- *The definition of the product to be deployed*, in terms of its elementary components (units). This description may use the designation of the units in the repository, and should specify all variants of the units being used. It should also identify the variant of the deployed system. This information is similar to that provided by an ADL.

- *The requirements and constraints* that apply to the deployed product, in relation with the target platform. This may include constraints specifying that a given unit should be deployed on a specified site, that two units should or should not be co-located, etc.

- *The tools to be used to perform the deployment.* This description may take a variety of forms, depending on the organization of the deployment engine: using a pre-deployed engine that is capable of interpreting the description; defining scripts that perform the deployment tasks, to be executed by specified interpreters, e.g., shells on various sites; defining scripts that generate the deployment engine itself.

Deployment description is still in its infancy. Due to lack of experience, there is no consensus yet on the definition of a deployment description language. Most existing systems use ad hoc, low level descriptions, often in the form of attribute-value pairs expressed in XML for convenience.

The evolution of deployment description languages is linked to that of the ADLs (**??**). Like ADLs, they should evolve towards a more abstract form expressing high-level QoS goals, from which a low-level description could be generated.

Examples of deployment descriptions are given in 10.4.5.

Deployment Engine: Design

The notion of a deployment engine, i.e., a device that carries out the various tasks of deployment, has been present in a number of systems since the early 1990s. However, until recently, there has been no unified view of the requirements and architecture of a deployment engine; each system defined its own ad hoc organization, often as a set of shell scripts linked by coordination code.

The current view of the deployment engine is that of a workflow engine. Recall that a *workflow* is a set of tasks T_i together with a list of precedence constraints, such as "$start(T_k)$ must follow $end(T_m)$ and $end(T_n)$". The tasks may then be executed in sequence or in parallel, subject to the constraints. The function of a workflow engine is to orchestrate the execution of the tasks, using events (Chapter 6) to mark changes of state and to trigger task inception. In a general workflow model, the execution of tasks may generate new tasks and the associated constraints, which are added to the existing sets.

Current efforts aim at defining a unified, generic model of a deployment engine, in order to reuse experience and to reduce development time. The most promising path is based on Model Driven Architecture (MDA), an approach promoted by the OMG [OMG MDA] that separates application logic from the underlying technology by defining abstract platform independent models (PIM). By analyzing the configuration and deployment process, one first identifies the generic steps that define the workflow tasks. The model may then be customized for a specific component model and implemented on several platforms.

This approach is taken by the OMG for its Configuration and Deployment Model standard proposal [OMG C&D][3], and by several research projects such as [Flissi and Merle 2005], which we briefly summarize below.

The abstract (platform independent) model defines a deployment workflow as a set of elementary tasks, namely:

- releasing and configuring the components that make up the application;
- uploading the components on the execution sites and generating component instances as needed; also registering any components that need further access by name;
- performing any remaining (site specific) configuration operations;
- performing all needed bindings (within each site and between sites); some bindings may be delayed until execution;
- activating the components through their provided entry points.

This is actually a refinement of the life-cycle illustrated on Figure 10.11.

[3]still under development at the time of writing.

These tasks are subject to precedence constraints, which derive from various dependency relations. Some of these relations are explicit, e.g., the dependencies between components, as derived from the *provides* and *requires* declarations. Others result from the data-flow, when a task needs data that are generated by another task (e.g., a factory creates a component instance that is further used in a binding). Others are specified by the application logic, e.g. the order in which the components must be started on the various sites. Most dependencies are explicitly or implicitly expressed in the deployment description.

The first step in the construction of the deployment engine is to specify the abstract deployment model by two sets of generic classes which define the data and the tasks, respectively. For instance, in one of the models defined in [Flissi and Merle 2005], the data classes comprise *System*, *Factory*, *Instance* and *Interface*, and the task classes include, among others, *Installation*, *Instantiation*, *Configuration*, and *Activation*. Dependencies between tasks are defined as indicated above.

The next step is to customize the model for a specific component model, e.g., Fractal or CCM, by specializing the generic classes of the abstract model. This is done through inheritance, by adding the needed attributes and operations to the generic classes. The model is still a PIM, since it does not depend on an underlying infrastructure.

The final step is to generate an implementation of the model for a specific platform. This in turn is done in two sub-steps:

- Deriving a platform specific model (PSM) from the customized PIM, through transformation rules. These rules allow the operations and data of the above specialized classes to be replaced by their equivalent form for the target platform. Note that the target platform may be different from the component model being deployed (e.g., a CCM application may well be deployed on a Fractal infrastructure).

- Generating the actual code of the deployment engine, based on the PSM. Part of this process may be automated, but some of the translation of the abstract methods into code must be done by hand.

This work is still experimental, but it contributes to a good understanding of the deployment process and opens the way for a systematic approach to the construction of deployment engines, as opposed to the current lack of standards, which hinders reuse and interoperability.

Deployment Engine: Operation

We now describe the operation of the deployment engine whose organization has been presented. We assume that the components to be deployed are equipped with a management interface, which provides administrative operations on the component, such as described in Chapter 7. In the case of legacy software, we need to assume that at least some of these operations exist in the form of "hooks" and may be exported through appropriate wrappers, which make the software available in component form.

We describe the operation of the deployment engine as an extension of a pattern described in [Schmidt et al. 2000] under the name Component Configurator. The deployment engine consists of a number of deployment tasks, orchestrated by a workflow engine. As indicated above, each deployment task is dedicated to a specific function such as configuration, installation, binding, etc. To perform its function, a task is guided by the deployment description and acts on the components to be deployed. Such actions includes moving a component to a remote site, creating a binding between several components, setting configuration parameters, etc. Most of these actions are executed by calling operations of the components' management interface. This is an instance of inversion of control (2.1.1). The workflow engine receives messages from the tasks, signaling completion of a deployment step, and in turn activates other tasks according to the precedence constraints derived from the deployment description. The deployment process is started by activating an initial task. The whole process is illustrated on Figure 10.13.

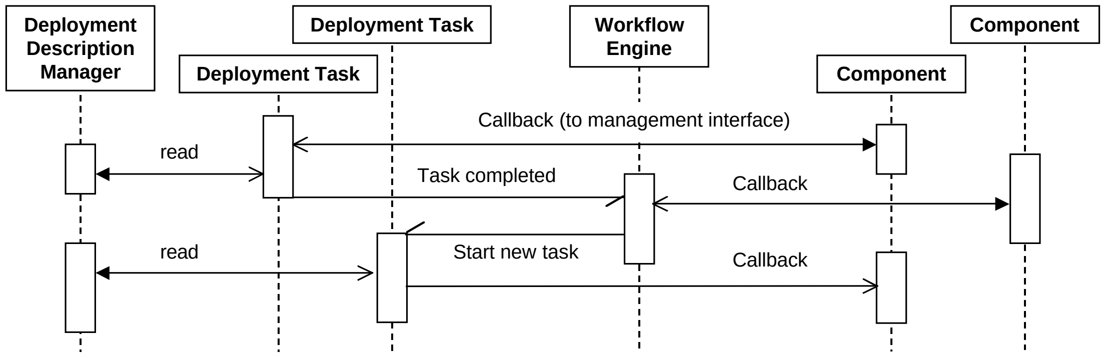


Figure 10.13. Overall deployment pattern

We have introduced a process called *Deployment Description Manager* to mediate access to the deployment description by the deployment tasks and the workflow engine.

The detailed description of the operations being performed by the callbacks depends on the specific component model being used. A generic framework that can be personalized for various component models is described in [Flissi and Merle 2004].

10.4.5 Frameworks for Architecture-based Deployment

In order to illustrate the concepts introduced above, we present a brief review of two current frameworks for component-based configuration. SmartFrog is an example of a working system that has been used for several years in an industrial environment. Nix is a more recent research project, which proposes a rigorous approach to the dependency problems arising in configuration management. Both systems are available in open source form.

SmartFrog: A Configuration Framework for Distributed Applications

SmartFrog (Smart Framework for Object Groups) [Goldsack et al. 2003, SmartFrog 2003] is a framework for describing, deploying, launching and managing distributed applications. SmartFrog has been developed at HP Labs and is being used in production systems.

SmartFrog supports Java-based applications in a distributed environment. It roughly follows the generic model described in 10.4.4.

- *Deployment description.* The description language provided by SmartFrog allows the following aspects of a system to be described in a declarative form: the components that make up the system, their configuration parameters, their connections, their intended location, and the elements of their life-cycle (e.g., in which order they should be started).
- *Repository and component model.* There is no separate repository in a physical sense; the components are hosted on the JVMs supported by the target platform. The components must conform to a specified model, which defines a base class from which any component must derive and a specific life-cycle. As usual, legacy applications may be integrated into SmartFrog by means of appropriate wrappers.
- *Deployment engine.* The deployment engine is implemented by a set of cooperating demons located on the sites of the target platform. In addition to the deployment proper, the engine continues to manage the application while it is running, and is responsible for its orderly shutdown.

We now describe these elements in more detail.

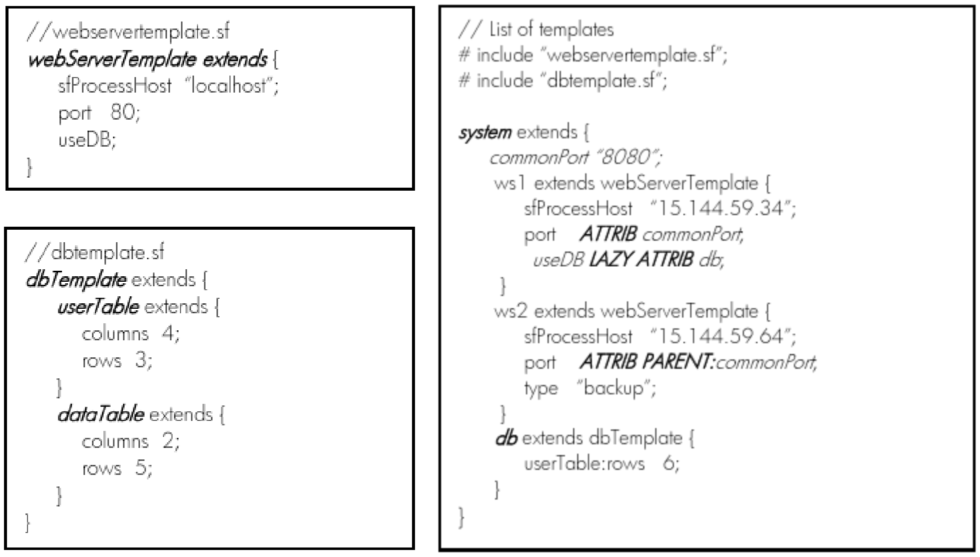
1. *Description Language.* The SmartFrog description language is a means for creating a declarative description of a system to be deployed. A component description is an ordered collection of attributes; an attribute is a name-value pair. Values may be of three kinds: (a) a basic, immediate value (e.g., an integer, a character string, etc.); (b) a reference to another attribute, through its name (the naming system is described below); and (c) a component description. The last form allows a tree of components to be built (the leaves of the tree being basic values), and defines a containment relationship (component A *contains* component B if the description of A includes an attribute whose value is B).

The language is untyped and prototype-based. Any attribute whose value is a component description may be considered as a prototype, or template, from which other components may be derived by extension (inheritance). The extension mechanism allows overriding the values of existing attributes and adding new attributes.

Figure 10.14 gives an example of SmartFrog component descriptions. The box on the right side shows the description of a system, which uses two templates, the components described by the two boxes on the left side. The components *ws1*, *ws2*, and *db* defined in *system* illustrate the extension mechanism: some attributes are added (e.g. *type* in *ws2*), and some values of existing attributes are overridden (e.g., *sfProcessHost* in both *ws1* and *ws2*). Note that the attribute named *useDB* in *webServerTemplate* does not define a value (a value may be added in an extension, as shown in the program of *system*).

The containment relationship defines a hierarchical naming system, in which each component defines a naming context, allowing composite names to be defined, like in a hierarchical file system. An instance of a simple name is *port*; an instance of a hierarchical name is *userTable:rows*, in the context *dbTemplate*.

We finally examine *references*, an important part of the language. A reference is a means for binding attribute values, both variables and components. For example, in



```
//webservertemplate.sf
webServerTemplate extends {
    sfProcessHost "localhost";
    port 80;
    useDB;
}
```

```
//dbtemplate.sf
dbTemplate extends {
    userTable extends {
        columns 4;
        rows 3;
    }
    dataTable extends {
        columns 2;
        rows 5;
    }
}
```

```
// List of templates
# include "webservertemplate.sf";
# include "dbtemplate.sf";

system extends {
    commonPort "8080";
    ws1 extends webServerTemplate {
        sfProcessHost "15.144.59.34";
        port   ATTRIB commonPort;
         useDB LAZY ATTRIB db;
     }
    ws2 extends webServerTemplate {
        sfProcessHost "15.144.59.64";
        port   ATTRIB PARENT:commonPort;
        type "backup";
     }
    db extends dbTemplate {
        userTable:rows 6;
    }
}
```

Figure 10.14. An instance of a SmartFrog description, adapted from [Goldsack et al. 2003]

the *system* component, both attributes "*port ATTRIB commonPort*" and "*port ATTRIB PARENT:commonPort*" refer to the same attribute, namely "*commonPort* `"`*8080*`"`" in the enclosing component. The *LAZY* keyword in attribute *useDB* means that the binding should be delayed until run time.

Note that the description language includes information related to both configuration proper (e.g., assigning values to attributes) and deployment (e.g., choosing a host site, defining a binding as dynamic).

2. *Component Model and Life-cycle.* A SmartFrog (primitive) component is a single Java object that implements a specific management API, in addition to its own functional interface. This interface, called `Prim`, defines a life-cycle controller, composed of three operations: *Deploy*, *Start*, and *Terminate* (Figure 10.15(a)). The life-cycle itself is shown on Figure 10.15(b).

An application is made up of a set of components, organized in a hierarchy by a parent-child relationship. The tree structure associated with this hierarchy is defined as follows: leaf nodes are instances of primitive components (as defined above), while non-leaf nodes are instances of compound components, which extend a `Compound` component. `Compound` is capable of monitoring a set of a collection of children, which can themselves be compound or primitive.

Thus, in the example of Figure 10.14, if we replace the line "`system extends {`" by "`system extends Compound {`", we define a component hierarchy in which the primitive components *ws1* and *ws2* are children of the compound component *system*. These children share the same life-cycle, which is monitored by the parent. They synchronously undergo the life-cycle steps of Figure 10.15(b), i.e., they terminate together, etc.

The component hierarchy may thus be used to define the activation and termination sequence of an application, through a sequence of downcalls and upcalls in the hierarchy,

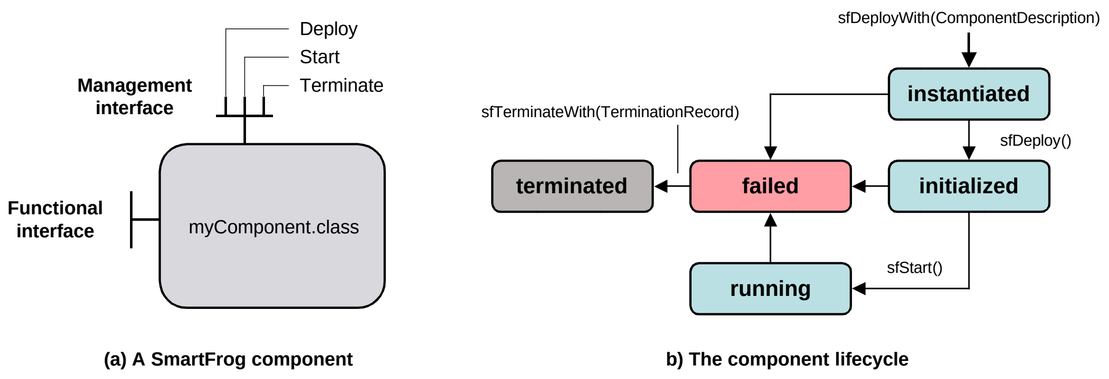


Figure 10.15. The SmartFrog component model, adapted from [Goldsack et al. 2003]

using the management interface. We do not discus the details of this process.

In order to assist deployment and management, SmartFrog includes a set of standard components, providing services such a component sequencing, naming, dynamic discovery, etc.

3. *Deployment Engine.* The SmartFrog deployment engine is a distributed workflow manager that uses the descriptions and component definitions to orchestrate the various tasks contributing to deployment. These tasks use the management methods provided by the SmartFrog component model to act on the components. The workflow manager itself is a fully distributed system, which may be started from any node and runs without a central point of control, using process group coordination mechanisms to manage its own operation.

The role of the engine is not limited to the deployment phase. The engine remains active during the application's run time, and is responsible for the shutdown of the application. The engine may be used as a generic workflow driver, which is able to orchestrate various run time management tasks, such as monitoring the application's state, reacting to failures, etc.

Since the SmartFrog engine is able to upload components to remote sites and to start their execution, care must be taken to prevent its being used for malicious actions, such as virus propagation, service denial, etc. Therefore the engine is complemented by a security framework, which organizes the target platform into a number of independent security domains, allowing responsibility to be partitioned. Inter-domain checks limit unwanted interaction. For instance, configuration descriptions, which trigger remote actions, must be signed by the issuing domain, and are authenticated before execution. In addition, secure channels are used for communication between nodes.

SmartFrog has been used for several years in an industrial environment, allowing some experience to be gathered. It provides a convenient alternative to low-level deployment techniques based on scripting. The configuration language allows a large degree of flexibility, from static descriptions to dynamic models using run time binding and discovery services. The environment has been shown to scale reasonably well, owing to the fully distributed, loosely coupled design of the deployment engine.

The component model and the life-cycle are fairly simple, but powerful enough to accommodate large, evolving applications. Although SmartFrog is developed in a Java environment, it may accommodate other components through wrapping techniques.

SmartFrog has some limitations. It does not have an organized repository. It does not have self-configuration facilities, and it assumes that an initial environment is present. Finally (like many existing deployment frameworks), it has been developed to respond to practical needs, and neither its language nor its component model have an underlying formal or conceptual base.

Nix: Providing Mechanisms for Safe Deployment

Nix [Dolstra et al. 2004a] is a framework which provides mechanisms that can be used to define a variety of deployment policies. It has been developed as a research project at the University of Utrecht. The main contribution of Nix is a model for safe and flexible deployment, based on a thorough analysis of the dependencies between components in the deployment process. Safety means that all dependencies between components are satisfied; flexibility means an unconstrained choice of deployment policies, and allows the possible coexistence of several versions of a component.

The proposed approach [Dolstra et al. 2004b] relies on an analogy between component deployment and memory management. Two main problems are identified:

- Unresolved dependencies. The *closure* of a component is the set of components on which it (transitively) depends for its correct operation. An unresolved dependency, the memory equivalent of a dangling pointer, occurs when a component is deployed without its complete closure.

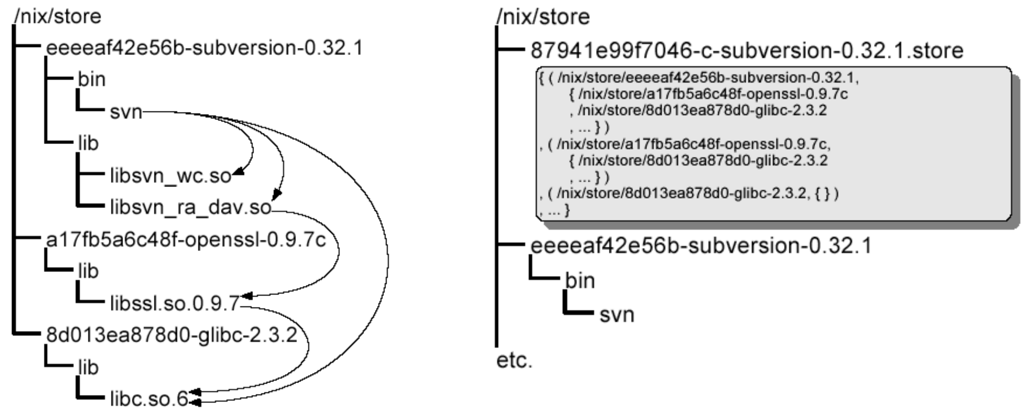
- Collisions. This refers to the inability of having two different versions of a component coexist because they occupy the same location in the file system.

The proposed solution is based on organizing the component repository as an address space (Figure 10.16(a)), in which each stored object has a globally unique name (called a *store path*), similar to a symbolic name in a hierarchical file system.

A component is typically represented by a directory in this space, under which are stored its elements such as source code, binary code, libraries, etc. A store path uniquely identifies the component and includes a prefix, which is a cryptographic hash of all inputs involved in building the component. Thus any change in one of these inputs generates a different store path.

In order to express dependencies, and thus to allow closures to be computed, the following discipline is applied: if object A depends on object B, then A contains the store path of B[4]. For example, the arrows on Figure 10.16(a) show the dependencies of a *subversion* component. The value of the corresponding closure is represented on Figure 10.16(b), as an object in its own right.

[4]Should this discipline be violated (e.g., by encoding store paths representations in a non-recognizable form), the dependency analysis would fail. The authors of Nix claim that this does not occur in practice.


/nix/store
eeeeaf42e56b-subversion-0.32.1
bin
svn
lib
libsvn_wc.so
libsvn_ra_dav.so
a17fb5a6c48f-openssl-0.9.7c
lib
libssl.so.0.9.7
8d013ea878d0-glibc-2.3.2
lib
libc.so.6
/nix/store
87941e99f7046-c-subversion-0.32.1.store
{ (/nix/store/eeeeaf42e56b-subversion-0.32.1,
{ /nix/store/a17fb5a6c48f-openssl-0.9.7c
, /nix/store/8d013ea878d0-glibc-2.3.2
, ... })
, (/nix/store/a17fb5a6c48f-openssl-0.9.7c,
{ /nix/store/8d013ea878d0-glibc-2.3.2
, ... })
, (/nix/store/8d013ea878d0-glibc-2.3.2, { })
, ... }
eeeeaf42e56b-subversion-0.32.1
bin
svn
etc.

(a) Organization of the store **(b) Closure value for *subversion***

Figure 10.16. The Nix store, from [Dolstra et al. 2004b]

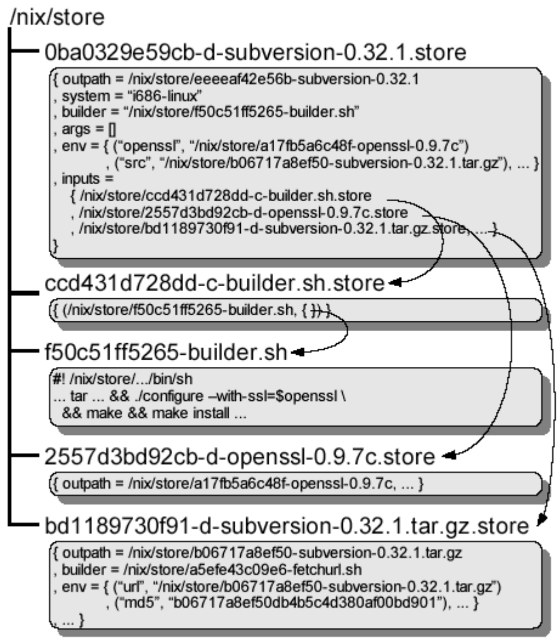
Closures are automatically determined by interpreting *derivation values*. As an example, the derivation value used to determine the *subversion* closure value is shown in the upper box of Figure 10.17.

Note that, in our terminology (10.4.4), a derivation value includes both a deployment description (e.g., the names of the components, a description of the target platform) and the program of some of the deployment tasks (in the form of shell scripts whose store paths are called *builders* in Nix). A derivation value is interpreted by the equivalent of a deployment workflow engine, which builds the closures and performs the actions specified by the builders. Input values are obtained by using the appropriate builders (e.g., fetch a component from a specified URL).

Path names and derivation values in the form shown above are hidden from clients, through various mechanisms. For instance, the notion of a user environment provides a window on the store, in a user-friendly form, by using the equivalent of symbolic links in a file system. Derivation expressions are not intended to be written by hand; they are generated from a higher level language.

The components to be deployed may be provided either in source form or in binary form. In the first case, binary forms are built using the process described above. In the second case, a mechanism called the *substitute* is used: a substitute describes a pre-built (binary) component, together with an access path (e.g., an URL) to it. Thus component provision in a binary form may be viewed as an optimization; if the binary form cannot be found, the process falls back on the construction from source form. Another optimization is the use of binary patches. A *binary patch* [Dolstra 2005] describes a set of edit operations that transform a base component into a target component (the equivalent of `diff` interpretation). Thus if both the base component and the patch are available, the target component may be built quickly, which is specially useful when upgrading to a new version of a product.

As a conclusion, the contribution of Nix is a deployment model based on a rigorous organization of the repository and a mechanism for expressing and enforcing dependencies. This makes the deployment process a purely functional one: the content of a component,



```
/nix/store
0ba0329e59cb-d-subversion-0.32.1.store
{ outpath = /nix/store/eeeeaf42e56b-subversion-0.32.1
, system = "i686-linux"
, builder = "/nix/store/f50c51ff5265-builder.sh"
, args = []
, env = { ("openssl", "/nix/store/a17fb5a6c48f-openssl-0.9.7c")
          , ("src", "/nix/store/b06717a8ef50-subversion-0.32.1.tar.gz"), ... }
, inputs =
    { /nix/store/ccd431d728dd-c-builder.sh.store
    , /nix/store/2557d3bd92cb-d-openssl-0.9.7c.store
    , /nix/store/bd1189730f91-d-subversion-0.32.1.tar.gz.store, ... }
}
ccd431d728dd-c-builder.sh.store
{ (/nix/store/f50c51ff5265-builder.sh, { }) }
f50c51ff5265-builder.sh
#! /nix/store/.../bin/sh
... tar ... && ./configure –with-ssl=$openssl \
  && make && make install ...
2557d3bd92cb-d-openssl-0.9.7c.store
{ outpath = /nix/store/a17fb5a6c48f-openssl-0.9.7c, ... }
bd1189730f91-d-subversion-0.32.1.tar.gz.store
{ outpath = /nix/store/b06717a8ef50-subversion-0.32.1.tar.gz
, builder = /nix/store/a5efe43c09e6-fetchurl.sh
, env = { ("url", "/nix/store/b06717a8ef50-subversion-0.32.1.tar.gz")
          , ("md5", "b06717a8ef50db4b5c4d380af00bd901"), ... }
, ... }
```

Figure 10.17. Derivation value for *subversion* in Nix, from [Dolstra et al. 2004b]

as stored in the repository, only depends on the inputs used to build it, and never changes after being built. Various optimizations [Dolstra 2005] allow the process to remain efficient in spite of the potentially large amount of work needed to rebuild a product after a change on a component located deeply in the dependency hierarchy.

Nix does not appear to have been used for deploying applications on a distributed target platform (clusters or grids), although nothing in its architecture prevents this extension.

10.4.6 Conclusion on Deployment

The area of configuration and deployment has received increasing attention in recent years, owing to the impact of these aspects on the overall systems management process, and to the high error rate of current practices. While some fairly elaborate experimental deployment environments have been developed (e.g., [Burgess 1995], [Anderson 2003]), many production systems use ad hoc solutions, due to the lack of established standards.

The current trend is towards a higher level approach to deployment, based on underlying models. We have presented two significant experiences based on component architecture. Other relevant research is described in [Roshandel et al. 2004], [Dearle et al. 2004]. Work is also being done on the foundations of the deployment process [Parrish et al. 2001, Di Cosmo et al. 2006]. As experience is gathered, standards are being prepared and should be in effective use in the coming years.

10.5 Reconfiguration

Reconfiguration is the process of modifying a deployed system or application. Reconfiguration is an adaptation process, which is a useful tool in the following situations.

- Mobility. In an increasing number of applications, the physical location of users, devices, and data is constantly changing. This calls for a corresponding adaptation of the application's structure.
- System evolution and maintenance. A system evolves by addition of new elements and replacement of existing elements. Maintenance operations may cause parts of a system to be temporarily stopped or disconnected. In both cases, reconfiguration is needed to avoid stopping the whole system and to preserve its correct operation.
- Fault tolerance. After an error has been detected, faulty components may need to be eliminated, and the tasks of the system must be reassigned among the remaining components (see 11.1.2).
- Dynamic monitoring. Observation and measurement facilities (10.3) may be dynamically inserted, if they have not been provided in the initial implementation. They may also be removed when unnecessary, thus reducing overhead.
- Autonomic computing. An autonomic system (10.2) responds to changes in its environment by adapting its structure or operating parameters. Reconfiguration is one of the tools available for this adaptation.

Reconfiguration may be static (stopping the system, performing the changes, and restarting the system); or it may be dynamic (reconfiguring the system while keeping it active, at least in part). We are specifically interested in *dynamic* reconfiguration, which aims at preserving service availability.

In this section, we first introduce the main requirements for dynamic reconfiguration. We then discuss the main relevant patterns, and the mechanisms needed to actually perform the operations needed for reconfiguration. We conclude with a few examples that illustrate the current state of the art.

10.5.1 Introduction and Requirements

In the architecture-based approach that we are taking, a system is an assembly of components, and reconfiguration is described in terms of operations on these components and on their bindings (or connectors). As indicated in 10.1.3, such operations include :

1. Changing a component's attributes, using its management interface.
2. Modifying bindings between existing components.
3. Adding or removing a component on a site, which entails modifying the bindings that involve that component.

4. Moving a component from a site to another one, while maintaining its bindings to other components.

5. Replacing a version of a component by a different version.

Operation 1 is internal to a component, and part of its planned operation. As such, it does not pose specific problems. Operations 2 and 3 modify the structure of the system, while operation 4 preserves the system's structure, and changes its geometry. Operation 5, called implementation update, may modify the component's implementation, the component's interface, or both.

Dynamic reconfiguration is subject to the following requirements, as presented in [Kramer and Magee 1990].

- Changes should be described in terms of the system's structure (a recall of the principle of the architecture-based approach). This requirement excludes fine-grain changes (i.e., in the internal structure of a component), which would be too complex to specify and to manage efficiently (such changes are performed by replacing the whole component).

- Change specifications should be declarative. This requirement separates the specification of the changes (the responsibility of the user) from their actual implementation (the responsibility of the reconfiguration framework).

- Change specifications should be independent of the algorithms and protocols of the application. This requirement favors the design of generic, reusable patterns and mechanisms as building blocks of reconfiguration systems.

- Changes should leave the system in a consistent state. This requirement calls for a "transparent" reconfiguration process, i.e., one that allows the reconfigured system or application to continue its normal operation after the reconfiguration has taken place. This may be expressed in the language of transactions: the execution of the application takes the system from a consistent state to a new consistent state, where consistency is expressed by a specified invariant; if the system is reconfigured during execution, the invariant should be preserved.

- Changes should minimize the disruption to the application system. In concrete terms, the set of nodes affected by the change should be the minimal set for which the reconfiguration specifications are respected; also performance degradation should be minimized.

Most of these requirements essentially embody the principle of separation of concerns: separating specification from implementation, defining application-independent reconfiguration specifications, isolating the reconfiguration process from the normal course of the application.

As reconfiguration is applied to large scale systems, new requirements appear:

- Scalability. The reconfiguration process should be scalable, i.e., its cost, in terms of performance and service disruption, should remain acceptable when the size of the system increases.

- Non-interference. If several reconfiguration operations are concurrently activated, their respective effects should be mutually isolated (a requirement similar to that of concurrent transactions).

Defining consistency criteria for reconfiguration is still an open issue. Current investigations use an architecture-based formal approach, along the following lines.

- Use an architecture description language (ADL) to describe the structure of the system as an assembly of components.
- Extend this language with a formal notation specifying the integrity constraints to be maintained is the system is reconfigured.
- At run time, verify that a proposed reconfiguration operation actually respects the constraints (and forbid the operation if this is not the case).

Examples of this approach are [Georgiadis et al. 2002, Warren et al. 2006]. Both experiments use the Alloy language [Jackson et al. 2000] to specify the integrity constraints.

10.5.2 Patterns for Dynamic Reconfiguration

In most cases, a complex reconfiguration operation may be described as a sequential or concurrent combination of a small number of patterns, each of which describes an elementary reconfiguration step. These patterns are: dynamic interposition, dynamic component replacement, component migration, dynamic change of network topology.

The actual embodiment of these patterns uses several basic mechanisms, or tools: quiescence detection, dynamic loading, state transfer, dynamic link update. These mechanisms are implemented using the APIs of the underlying operating systems and communication networks.

We first examine the constraints needed to preserve consistency. When a component is involved in a reconfiguration process, it must be in a consistent state[5] at the beginning of the operation, and remain in a consistent state. We assume that the components communicate through synchronous method calls[6], and that several threads may be simultaneously executing methods within that component. A component is in the *active* state when at least a thread is currently executing one of its methods. It is in the *passive* state when no thread is executing one of its methods. However, when a component is observed to be passive, there is no guarantee that it will remain passive in the future, because one of its methods may be called by a thread currently executing within another component. A stronger property is needed: a component is in the *quiescent* state if (i) its internal state is consistent; (ii) the component is passive; and (iii) it will remain passive, i.e., no further calls to methods of this component will be issued by other components[7]. Quiescence is the precondition to be satisfied by a component involved in a reconfiguration.

In this section, we give an outline of the main reconfiguration patterns. The basic mechanisms needed for their implementation are discussed in Section 10.5.3.

[5]We need to assume that consistency is defined by application dependent constraints, e.g., within a transaction framework.

[6]The notions presented here can be extended to asynchronous communication schemes.

[7]A similar condition is introduced in distributed termination detection.

Dynamic Interposition

The simplest form of dynamic interposition is the insertion of an interceptor (2.3.4) in front of a component C, while the system is running. Therefore all bindings whose target is C need to be rebound to point to the interceptor, which must be itself bound to C (Figure 10.18).

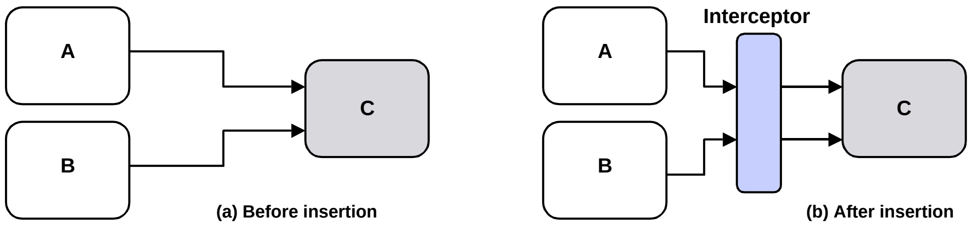


Figure 10.18. Dynamic interposition

This rebinding operation must ensure consistent method execution: all the calls directed to C before the rebinding must return to the initial caller, while all the new calls must return to the interceptor.

Once dynamic interposition is available, it may be used for dynamically inserting a wrapper around a component [Soules et al. 2003]. The interceptor must be programmed to embed the call to a method of component C between a pre-call and a post-call (Figure 10.19). Specific wrapping code may be supplied for each method of the component.

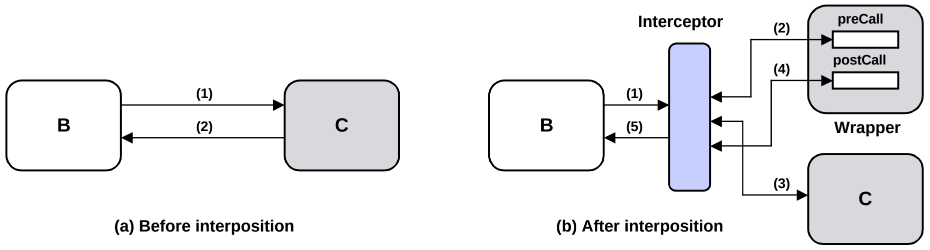


Figure 10.19. Dynamic wrapper insertion (adapted from [Soules et al. 2003])

A similar pattern has been implemented in CORBA using the Portable Request Interceptors defined by the OMG. A request interceptor is statically inserted, at the client and/or the server side. This interceptor redirects the request to another interceptor based on rules. A rule is a (condition, action) couple: if the condition is met, an action is taken. Rules may be inserted and removed at run time, which allows reacting to unanticipated conditions. See [Sadjadi and McKinley 2004] for technical details.

Dynamic Component Replacement

Dynamic interposition may be used to do dynamic component replacement. The goal is to replace a component C by a component C', which has a conformant interface, without stopping execution, and with minimal disruption to the system's operations.

The process uses an specific interceptor called the *mediator*, and is done in two phases (Figure 10.20), after the mediator has been inserted.

1. In the *blocking* phase, new calls to C (a and b on the figure) are blocked by the mediator, while calls in progress (? on the figure) are allowed to go to completion. When C is in the quiescent state, the next phase (*transfer*) is initiated.
2. In the *transfer* phase, the mediator performs state transfer between C and its new version C' (the state is guaranteed to be consistent since C is quiescent). It then updates the bindings whose target is C to point to C' (the mechanisms for rebinding are discussed in 10.5.3). Finally the mediator allows the blocked calls to resume, using the new version.

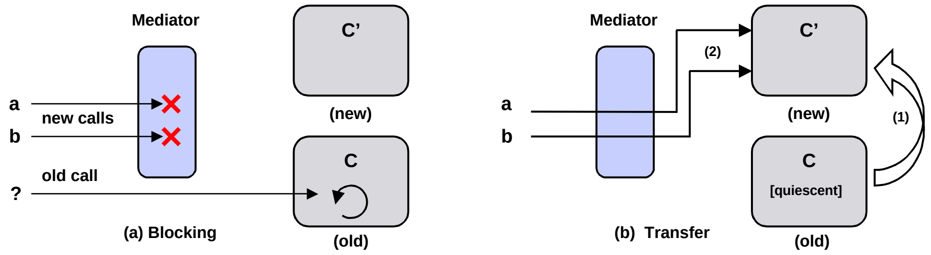


Figure 10.20. Dynamic component replacement (adapted from [Soules et al. 2003])

When the blocked calls have been redirected, the mediator and the old version of the component, C, may be deleted or garbage-collected.

This process may be optimized by splitting the *blocking* phase in two: a *forward* phase, and the *blocking* phase proper, as explained in 10.5.3.

Component Migration

Component migration (dynamically moving a component C from a site S to a site S') is a direct application of component replacement, in which component C' is a copy of component C located on site S'. The copy, which includes state transfer, must be done when component C is quiescent.

10.5.3 Implementation Issues for Dynamic Reconfiguration

We now examine the implementation of the basic mechanisms needed in the patterns described above.

Quiescence Detection

We first describe the principle of the solution used in the K42 operating system [Soules et al. 2003] to detect the quiescence of a component. This principle would apply to any system based on threads. We need to assume that any thread execution terminates in finite time (this condition must be ensured by the application).

The main idea is to divide the execution of threads into several epochs (in practice, only two epochs are needed). Suppose that a structural change (e.g., inserting an interceptor) occurs at time $t1$. Then we define two generations of threads:

- The threads that were started before $t1$ (the "non-current" generation); they run in the "old" structure (e.g., in the original component, before the interceptor was inserted).
- The threads that were started after $t1$ (the "current" generation); they run in the "new" structure (e.g., they run through the interceptor).

Therefore the threads of the non-current generation, numbered g, will terminate in finite time; let $t2 > t1$ be the time at which this occurs. Then g becomes the number of the current generation (and new threads created after $t2$ are assigned to it), while the generation that was current before $t2$ becomes non-current.

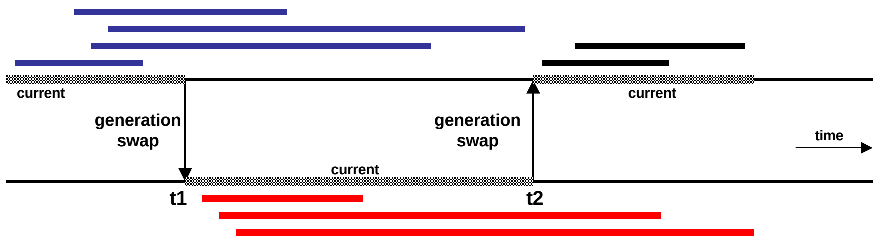


Figure 10.21. Quiescence detection

Thus two generation swaps are needed to establish that the current generation of threads have terminated (Figure 10.21). The termination of a generation of threads is detected by maintaining a thread counter, and is signaled by means of a callback mechanism to avoid active wait.

Consider now a component-based system using events (see Chapter 6). In such a system, the receipt of an event (represented by a message of a specified form) triggers a reaction (the execution of a code sequence in the component), which may in turn generate new events. Incoming events are queued while the component is busy processing current events. If the processing of an event (i.e., the execution of the reaction associated with it) is an atomic operation, then the component is in a consistent state whenever no event processing is in progress. Then, if all events directed to that component are intercepted and buffered, the component is quiescent and may be replaced or transferred, and its state may be copied. When the new component is reactivated, it starts by processing the buffered events.

Reference Management

Dynamic interposition and component replacement involve redirection of references from a component to another one. For instance, if an interceptor is inserted in front of component C, all references to C from its client components need to be updated.

There are two main approaches to this problem.

1. Reverse linking. All references from a client component to a component C are tracked: when a client component is bound to C, a reverse link to that client is created and associated with C. Thus, when an interceptor I is inserted in front of C, all references to C are found and are changed to references to I.

2. Indirection. All references to C point to an indirect link, which itself contains a reference to C. To insert an interceptor I, one only needs to update that link.

Indirection is much simpler to manage (in particular to perform a change atomically), but involves a constant (small) run-time overhead at each call. It is the most widely used solution. Usually, the indirect link is part of a local context maintained by each client. We illustrate this mechanism with two examples.

- The K42 operating system [Soules et al. 2003] uses a per-client object translation table, which is an array of indirect links to the components (objects in K42) accessed by this client.

- Enterprise Java Beans (EJB) systems use a double indirection to manage references between beans. Each bean has a local naming context. Names in this context point to entries in a global, system-wide, naming context, in which all beans are registered. These contexts are managed through the JNDI API and may be explored using the *lookup()* operation.

10.5.4 Conclusion on Reconfiguration

System reconfiguration at run time has long been considered of marginal interest, but is entering current practice due to the increasing need for dynamic adaptation.

A malfunction of the reconfiguration process may have potentially disruptive effects on a running application. Therefore safety is a major concern for reconfiguration. To ensure safety, one needs to specify the invariants to be maintained to keep the system in a consistent state, and to design algorithms that maintain these invariants through reconfiguration.

While a few patterns have been identified for implementing safe reconfiguration, a long term goal is to ensure safety through formal means. The current path of research relies on an architecture-driven approach, complemented by a rigorous definition of consistency. The inclusion of dynamic aspects into a architecture description language, augmented with consistency specification, would then allow the reconfiguration process to be fully automated.

10.6 Historical Note

Systems administration in the 1970s and early 1980s was essentially concerned with the management of mainframes, and of "data centers" made up of collections of mainframes. At that time, the field of systems administration was hardly considered as a subject for research. The first conference series dedicated to the topic, Usenix LISA (Large Installation Systems Administration Conference [LISA]), started as a workshop in 1987 and acquired conference status in 1990.

The mid-1980s were marked by the rise of the Internet, the development of client-server systems, and the wide diffusion of personal computers. The administration of distributed systems had to integrate the issues of network management and machine administration. The first administration models applicable to distributed systems (ODP, OSI) were developed in the late 1980s and early 1990s. See part V of [Veríssimo and Rodrigues 2001] for a detailed account of these models and of the platforms based on them.

The growth of both local-area networks and the global Internet in the 1990s made client computers and applications increasingly dependent on both networks and servers, and emphasized the issues of performance, availability, and security. However, the dominant administration models remained based on a manager-agent paradigm, and became less and less adequate for meeting these new demands.

In the 2000s, an increasing part of human activity, both in industry and in everyday life, is relying on Internet services. Deficiencies in system administration (and specially in configuration and deployment) are now identified as a major cause of failure for large scale Internet services [Oppenheimer et al. 2003], emphasizing the need for a new approach. Network and systems administration has become an active area of research, and the new standards that will replace those based on the manager-agent model should be shaped by the results of this research. It is likely that an increasing part of system administration will be automated, as a result of a better understanding of adaptable and self-managing systems.

Chapter 11

Availability

For a system that provides a service, *availability* is defined as the ability of the system to actually deliver the service. While several causes may hamper this ability, this chapter concentrates on the methods of improving system availability in the face of failures. *Fault tolerance* covers the concepts and tools that aim at achieving this goal.

The chapter starts with a review of the main concepts and techniques related to fault tolerance. It specifically examines the case of distributed systems, with special emphasis on group communication, which plays a central role in this area. It goes on with the application of these techniques to two main problems: improving the availability of servers, and improving the availability of data. It then examines the notions related to partial availability, and concludes with two case studies.

11.1 Main Concepts and Terminology

A system is *available* if it is ready to correctly deliver the service that it is designed to provide. Availability may be hindered by several causes, among which:

- failures, i.e., accidental events or conditions that prevent the system, or any of its components, from meeting its specification;
- overload, i.e., excessive demand on the resources of the system, leading to service degradation or disruption;
- security attacks, i.e., deliberate attempts at disturbing the correct operation of the system, causing malfunction or denial of service.

In this chapter, we examine how availability can be maintained in the presence of failures. Overload conditions are discussed in Chapter 12 and security attacks in Chapter 13.

This section presents a few elementary notions related to fault tolerance: availability and reliability measures (11.1.1); basic terminology (11.1.2); models of fault tolerance (11.1.3); availability issues for middleware (11.1.4). Section 11.2 examines the main aspects of fault tolerance: error detection, error recovery, and fault masking. This presentation is only intended to provide the minimal context necessary for this chapter, not to cover the subject in any depth. For a detailed study, see Chapter 3 of [Gray and Reuter 1993].

11.1.1 Measuring Availability

The measure of availability applies to a system as a whole or to any of its individual components (we use the general term "system" in the following). A system's availability depends on two factors:

- The intrinsic failure rate of the system, which defines the system's *reliability*.
- The mean time needed to repair the system after a failure.

The reliability of a system is measured by the mean time that the system runs continuously without failure. More precisely, consider a system being started (or restarted) at some instant, and let f be the length of time till its first failure. This is a random variable (since failures are mostly due to random causes). The mean (or expected value) of this variable f is called the Mean Time To Failure ($MTTF$) of the system.

Depending on the causes of the failures, the $MTTF$ of a system may vary with time (e.g., if failures are related to aging, the $MTTF$ decreases with time, etc.). A current assumption (not valid for all systems) is that of failures caused by a memoryless process: the probability rate of failure per unit time is a constant, independent of the past history of the system. In that case, the $MTTF$ is also a constant[1].

Likewise, the repair rate is measured by the expected value of the time needed to restore the system's ability to deliver correct service, after a failure has occurred. This is called the Mean Time To Repair ($MTTR$). One also defines Mean Time Between Failures ($MTBF$) as $MTTF + MTTR$. In the rest of this section, we assume that both the $MTTF$ and the $MTTR$ are constant over time.

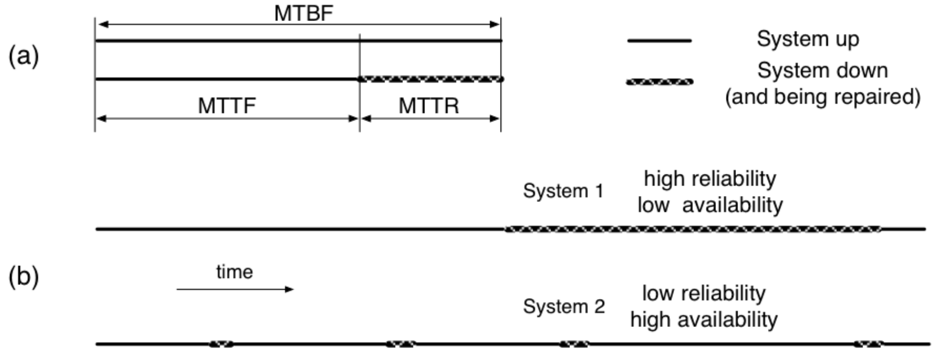


Figure 11.1. Reliability vs Availability

Assume a failure mode (the fail-stop mode, see 11.1.3) in which the system is either correctly functioning or stopped (and being repaired). Define the availability a of a system

[1]For a memoryless failure process, let p be the probability of failure per unit time. Then, if $p \ll 1$, $MTTF \approx 1/p$.

as the mean fraction of time that the system is ready to provide correct service. Then (see Figure 11.1a), the relationship between availability and reliability is given by:

$$a = MTTF/(MTTF + MTTR) = MTTF/MTBF.$$

The distinction between reliability and availability is further illustrated in Figure 11.1b.

To emphasize the importance of the repair time, note that the *un*availability rate of a system is $1-a = MTTR/(MTTF+MTTR) \approx MTTR/MTTF$, because in most practical situations $MTTR \ll MTTF$. Thus reducing $MTTR$ (aiming at quick recovery) is as important as increasing $MTTF$ (aiming at high reliability). This remark has motivated, among others, the Recovery Oriented Computing initiative [ROC 2005].

In practice, since the value of a for a highly available system is a number close to 1, a common measure of availability is the "number of 9s" (e.g., five nines means $a = 0.99999$, a mean down time of about 5 minutes per year).

11.1.2 Failures, Errors and Faults

In order to clarify the concepts of dependability (a notion that covers several qualities, including fault tolerance and security), and to define a standard terminology for this area, a joint committee of the IEEE and the IFIP was set up in 1980 and presented its proposals in 1982. A synthetic presentation of the results of this effort is given in [Laprie 1985]. A more recent and more complete document is [Avižienis et al. 2004]. These definitions are now generally accepted.

We briefly present the main notions and terms related to fault tolerance (a summary is given in 10.1.3).

A system is subject to a *failure* whenever it deviates from its expected behavior. A failure is the consequence of the system being in an incorrect state; any part of a system's state that fails to meet its specification is defined as an *error*. An error, in turn, may be due to a variety of causes: from human mistake to malfunction of a hardware element to catastrophic event such as flood or fire. Any cause that may lead to an error, and therefore may ultimately occasion a failure, is called a *fault*.

The relationship between fault and error and between error and failure is only a potential one. A fault may exist without causing an error, and an error may be present without causing a failure. For example, a programming error (a fault) may lead to the corruption of an internal table of the system (an error); however the table being in an incorrect state does not necessarily lead to a failure. This is because the incorrect values may never be used, or because the internal redundancy of the system may be such that the error has no consequence on the system's behavior. A fault that does not cause an error is said to be *dormant*; likewise, an undetected error is said to be *latent*. If the failure does occur, the time delay between the occurrence of the fault and the failure is called the latency. A long latency makes it difficult to locate the origin of a failure.

In a complex system, a fault which affects a component may cause the system to fail, through a propagation mechanism that may be summarized as follows (Figure 11.2).

Suppose component A depends on component B, i.e., the correct operation of A relies on the correct provision of a service by B to A. A fault that affects B may cause an error in B, leading to the failure of B. For A, the inability of B to deliver its service is a fault.

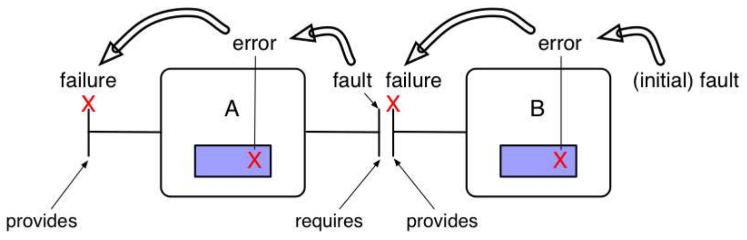


Figure 11.2. Error propagation

This fault may lead to an error in A, possibly causing the failure of A. If A is itself used by another component, the error may propagate further, and eventually lead to the failure of the whole system.

How to prevent the occurrence of failures? The first idea is to try to eliminate all potential causes of faults. However, due to the nature and variety of possible faults (e.g., human mistakes, hardware malfunction, natural causes, etc.), perfect fault prevention is impossible to achieve. Therefore, *systems must be designed to operate in the presence of faults*. The objective is to prevent a fault from provoking a failure, and, if a failure does occur, to recover from it as fast as possible. Methods for achieving this goal are presented in 11.2.

11.1.3 Models for Fault Tolerance

Recall (see 1.4.2) that a model is a simplified representation of (part of) the real world, used to better understand the object being represented. Models are specially useful in the area of fault tolerance, for the following reasons:

- It is convenient to have a framework to help classify and categorize the faults and failures, because of their wide variety.
- The tractability of several problems in fault tolerance strongly depends on the hypotheses on the structure and operation of the system under study. A model helps to accurately formulate these assumptions.

We represent the system as a set of components which communicate through messages over a communication system (see Chapter 4).

Here is a (simplified) classification of failures, in increasing degree of severity (see e.g., [Avižienis et al. 2004] for more details, and for a taxonomy of faults). The component affected by the failure is referred to as “the component”.

- The simplest failure mode is called *fail-stop*. When such a failure occurs, the component immediately stops working (and therefore stops receiving and sending messages). Thus, in the fail-stop mode, a component is either operating correctly or inactive[2].

[2]Since fail-stop is a simple, well understood behavior, a common technique is to force any failure to this mode, by stopping a faulty component as soon as an error is detected, and by signaling the failure to the users of the component. This technique is called *fail-fast*.

- In *omission* failures, the component may fail to send and/or to receive some messages. Otherwise its behavior is normal. This failure mode may be useful to simulate a malfunction of the communication system.
- *Timing* failures only affect the temporal behavior of the component (e.g., the time needed to react to a message).
- In the most general failure mode, the behavior of the failed component is totally unrestricted. For historical reasons [Lamport et al. 1982], this kind of failures is called *Byzantine*. Considering Byzantine failures is useful for two reasons. From a theoretical point of view, this failure mode is the most difficult to handle. From a practical point of view, solutions devised for Byzantine failures may be used in the most extreme conditions, e.g., for critical applications in a hostile environment, or for systems subject to attacks.

The behavior of the communication system has a strong influence on fault tolerance techniques. We consider two main issues: asynchrony and message loss.

Regarding asynchrony, recall (4.1.2) that a distributed system, consisting of nodes linked by a communication system, may be synchronous or asynchronous. The system is *synchronous* if known upper bounds exist for the transmission time of an elementary message, and for the relative speed ratio of any two nodes. If no such bounds exist, the system is *asynchronous*. Intermediate situations (*partially synchronous* systems) have also been identified [Dwork et al. 1988]. Some form of synchrony is essential in a distributed system, because it allows the use of timeouts to detect the failure of a remote node, assuming the communication link is reliable (details in 11.4.1). In an asynchronous system, it is impossible to distinguish a slow processor from a faulty one, which leads to impossibility results for some distributed algorithms in the presence of failures (details in 11.3.3).

Regarding message loss, the main result [Akkoyunlu et al. 1975], often known as the Generals' paradox [Gray 1978], is the following. Consider the coordination problem between two partners, A and B, who seek agreement on a course of action (both partners should perform the action, or neither). If the communication system linking A and B is subject to undetected message loss, no terminating agreement protocol can be designed. The communication protocols used in practice (such as TCP) do not suffer from this defect, because they implicitly assume a form of synchrony. Thus the loss of a message is detected by acknowledgment and timeout, and the message is resent till either it is successfully received or the link is declared broken.

11.1.4 Working Assumptions

We are specifically interested in availability support at the middleware level. Middleware sits above the operating system and networking infrastructure, and provides services to applications. Thus we need to examine (a) the assumptions on the properties of the infrastructure; and (b) the availability requirements placed by applications on the middleware layer.

Middleware most frequently uses the communication services provided by the network transport level (4.3.2). The guarantees ensured by these services vary according to the nature of the network (e.g., a cluster interconnect, a LAN, the global Internet, a wireless network). However, for all practical purposes, we make the following assumptions in the rest of this chapter:

- The communication system is *reliable*, i.e., any message is eventually delivered uncorrupted to its destination, and the system does not duplicate messages or generate spurious messages.

- The middleware system is *partially synchronous*, in the following sense: there exists bounds on the transmission time of a message and on the relative speed of any two processes, but these bounds are unknown a priori and only hold after some time.

  This assumption is needed to escape the impossibility results (see 11.3.3) that hold in asynchronous systems; its validity is discussed in 11.4.1.

Different assumptions may be needed in specific cases, e.g., message loss and network partition in the case of wireless networks.

We assume a fail-stop (or fail-fast) failure mode for components, be they hardware or software. Again, different assumptions may hold in specific situations. Since a system is made of a (potentially large) number of components, a common situation is one in which some part of the system is subject to failure. The goal is then to preserve the ability of the system to deliver its service, possibly with a degraded quality. This issue is developed in 11.8.

According to the "end to end principle" ([Saltzer et al. 1984], see also 4.2.2), some functions, including those related to fault tolerance, are best implemented at the application level, since the most accurate information about the application state and the goals to be pursued is available at that level. The role of middleware with respect to fault tolerance is to provide the tools to be used by the applications to achieve specified availability guarantees. To do so, the middleware layer uses the communication primitives of the transport level, such as specified above. For example (see 11.4), the middleware layer uses reliable point to point transmission to implement reliable broadcast and atomic broadcast, which are in turn used to improve availability at the application level.

What are the main causes of failures in actual systems? Two landmark papers are [Gray 1986], which analyzes failures in the Tandem fault-tolerant servers, and [Oppenheimer et al. 2003], which investigates failures in large scale Internet services. Both studies observe that system administration errors are the dominant cause of failure (about 40% for servers, 35% for Internet services). Most of these administration errors are related to system configuration. Software failures account for about 25% in both studies. The part of hardware is about 18% for servers, 5 to 10% for Internet services. The proportion of network failures in Internet services depends on the nature of the application, in the range of 15 to 20%, and may reach 60% for "read mostly" services. Contrary to the fail-stop assumption, network failures tend to be gradual, and are less easily masked than hardware or software failures.

11.2 Aspects of Fault Tolerance

Fault tolerance is based on a single principle, *redundancy*, which may take various forms:

- Information redundancy. Adding redundant information helps detecting, or even correcting, errors in storage or in transmission. In particular, our assumption on reliable transmission is based on the use of error detecting or correcting codes, together with sending retries, at the lower levels of the communication protocols.

- Temporal redundancy. This covers the use of replicated processing: the same action is performed in several instances, usually in parallel, to increase the probability that it will be achieved at least once in spite of failures. The application of this principle is developed in 11.6.

- Spatial redundancy. This covers the use of replicated data: information is maintained in several copies to reduce the probability of data loss or corruption in the presence of failures. The application of this principle is developed in 11.7.

There are many ways to exploit these forms of redundancy, leading to several fault tolerance techniques, which in turn can be combined to ensure system availability. The main basic techniques can be categorized as follows.

Error detection is the first step of any corrective course related to fault tolerance. It identifies the presence of an error (a system state that does not satisfy its specification), and triggers the further operations. Error detection is further detailed in 11.2.1.

Error compensation (or fault masking) consists in providing the system with enough internal redundancy so as to cancel the effects of an error, thus preventing the occurrence of a fault from causing the system to fail. Masking is further detailed in 11.2.3.

If failure actually occurs, *error recovery* aims at restoring the system's ability to deliver correct service by eliminating the errors that caused the failure. Two main approaches may be followed (more details in 11.2.2).

- *Backward recovery* consists in bringing the system back to a past state known to be correct. This is a generally applicable technique, which relies on the periodic saving of the (correct) system state (*checkpointing*).

- *Forward recovery* aims at reconstructing a correct state from the erroneous one. This technique implies that the system state contains enough redundancy to allow reconstruction, which makes it dependent from the specificities of the system. State reconstruction may be imperfect, in which case the service may be degraded.

As a practical conclusion of this brief review, there is no single universal method to ensure fault tolerance. The method chosen for each concrete case strongly depends on its specific aspects. Therefore, it is important to make all assumptions explicit, and to verify that the assumptions are indeed valid. For instance, fault masking by redundant processing is only effective if the replicated elements are subject to independent faults. Modularity is helpful for fault tolerance, since it restricts the range of analysis, and makes error detection and confinement easier.

Achieving fault tolerance remains a difficult proposition, as illustrated by such "horror stories" as the Ariane 5 [Lions et al. 1996] and Therac 25 [Leveson and Turner 1993] failures.

11.2.1 Error Detection

The goal of error detection is to identify the presence of an error, i.e., a system state that deviates from its specification. The motivation is to prevent (if possible) the error to cause the affected component(s) to fail, to avoid the propagation of the error to other parts of the system, and to better understand the fault that caused the error, in order to prevent further occurrences of the fault.

Two qualities are attached to error detection.

- *Latency*, the time interval between the occurrence of an error and its detection.
- *Coverage*, the ratio of detected errors to existing errors.

Error detection techniques depend on the nature of the error. For instance a fault in a processor may cause it to deliver incorrect output, or to stop. In the first case, the output needs to be analyzed by one of the techniques described below; in the second case, a failure has occurred, and failure detection techniques are needed. Failure detection is the subject of 11.4.1. Here we only examine the detection of errors in data.

A first category of methods uses redundancy. One common technique for detecting errors in data transmission relies on error-detecting codes. For example, one extra (parity) bit detects an error affecting one data bit. With more redundancy (error-correcting codes), an error may even be corrected. A textbook on error correction is [Blahut 2003].

For data production (as opposed to transmission), errors may be detected by comparing the results of replicated data sources. For example, bit by bit comparison of the output of two identical processors, with identical inputs, allows error detection. One needs to make sure that the replicated data sources have independent failure modes (e.g., separate power supply, etc.). Techniques based on comparison have a high cost, but ensure low latency and high coverage rate.

Another class of methods relies on plausibility checks. These methods are application dependent. For instance, the values output by a process may be known to be in a certain range, and this property may be checked. Alternatively, bounds may be known for the variation rate in a series of output values. Plausibility checks are usually lest costly than techniques based on redundancy. Their latency depends on the specific situation. Their weak point is their coverage rate, which is often small, because plausibility usually entails loose constraints.

Examples of the use of error detection techniques are presented in the case studies (11.9).

11.2.2 Error Recovery

As noted in 11.2, backward recovery is a general technique, while forward recovery is application-specific. In this section, we briefly examine the main aspects of backward recovery.

Stated in general terms, the principle of backward recovery is to replace an erroneous system state by a previously recorded correct state, an operation called *rollback*. The process of state recording is known as *checkpointing*. The state is copied on stable storage, a highly available storage medium.

The state of a distributed system consists of the local states of its components, together with the state of the communication channels, i.e., the messages that were sent but not yet delivered. This state must be *consistent*, i.e., it must be the result of an observation that does not violate causality. To explain this notion, consider a system made up of two components A and B, and suppose A sends a message m to B. If A registers its state *before* sending m, and B registers its state *after* receiving m, the global checkpointed state is inconsistent, because it records the receipt of a message that has not been sent.

There are two main techniques to construct a consistent state. In coordinated checkpointing, the components of the system register their state in a coordinated way, by exchanging messages, so as to ensure that the recorded state (including the state of the communication channels) is consistent. A coordination algorithm is described in [Chandy and Lamport 1985]. In uncoordinated checkpointing, components checkpoint their state independently. If recovery is needed, a consistent state must be reconstructed from these independent records, which implies that multiple records must be conserved for each component.

Coordinated checkpointing is usually preferred, since it implies less frequent state saving (state recording on stable storage tends to be the major component of the cost). In addition, a technique known as message logging allows a more recent state to be reconstructed from a consistent checkpoint, by recording the messages sent by each process, and replaying the receipt of these messages, starting from the recorded state. A survey of checkpoint-based recovery is [Elnozahy et al. 2002].

Backward recovery is illustrated by two examples, which are further detailed in this chapter.

- *Process pairs*, a technique introduced in the HP NonStop system (originally Tandem [Bartlett 1981]). In this system, all hardware components (CPU, bus, memory modules, controllers, disks) are replicated, and the operating system implements highly available processes, by representing a process by a pair of processes running on different processors. More details in 11.6.2.

- *Micro-reboot*, a technique used in multi-tier middleware [Candea et al. 2004]. This technique is based on the empirical observation that a software failure of undetermined origin is usually cured by rebooting the system (restarting from a "clean" state). The difficulty is to determine the environment to be rebooted: it should be large enough to repair the error, but as small as possible for efficiency. More details in **??**.

See [Bartlett and Spainhower 2004] for a review and a comparison of two highly available commercial systems using recovery (HP-Tandem NonStop and IBM zSeries).

11.2.3 Fault Masking

Fault masking (providing the system with enough internal redundancy to suppress the effect of a fault) has been used for a long time to improve the availability of circuits, through the technique of Triple Modular Redundancy (TMR). The circuit is organized in successive stages (the output of stage i is the input of stage $i + 1$), and the components of each stage are replicated in three instances, connected to a majority voter. The main assumption is that the replicated components are highly reliable and fail independently[3]. The voter compares the output of the three components. In the absence of failures, the three outputs are identical. If one output differs from the other two, it is considered faulty, and discarded. The probability of the three outputs being different (i.e., at least two faulty components) is assumed to be negligible. Since the voter itself is a component that may fail, it is also replicated in three instances. This system tolerates the failure of one component at each stage.

Another instance of hardware fault masking using voting is the design of the Tandem Integrity system, in which the voter compares the output of three replicas of the CPU, thus masking the failure of one replica. In contrast with the Tandem NonStop system (11.2.2), which relies on process pairs supported by a special purpose operating system, Tandem Integrity can use any standard system, such as Unix.

A general method for ensuring fault masking is described in 11.3.1.

11.3 State Machines and Group Communication

A generic model that captures the notion of redundancy, based on state machines, is described in 11.3.1. To use this model to actually implement fault tolerant systems, one needs group communication protocols, whose specification and implementation are respectively examined in 11.3.2 and 11.4.

11.3.1 The State Machine Model

A general model for a hardware or software system (or for a system component) is that of a *deterministic state machine* (SM), executed by a process which receives requests on an input channel and sends answers on an output channel. A number of such machines may be assembled to make up a complex system.

The behavior of the state machine may be described as follows.

Call S_0 the initial state of the SM and S_k its state after having processed the k^{th} request r_k ($k = 1, \dots$). The effect of the receipt of request r_i is the following:

(a) $S_i = F(S_{i-1}, r_i)$

(b) $a_i = G(S_{i-1}, r_i)$

(c) the answer a_i is sent on the output channel.

The functions F and G (the state transition function and the output function, respectively) define the behavior of the SM.

In order to ensure consistency, causal order[4] must be preserved for request transmission: if $send(r)$ and $send(r')$ are the events of sending two requests, and if $send(r)$ happens before $send(r')$, then r' cannot be delivered to the SM unless r has been delivered.

A system implemented as an SM can be made tolerant to a number f of failures, by the following method. The SM (together with the process that executes it) is replicated

[3]While this assumption is usually justified for hardware components, it needs to be reconsidered if the failures are due to environmental conditions, such as high temperature, shared by all replicated components.

[4]Causal order is defined by the *happened before* relationship [Lamport 1978b], which subsumes local order on a single site and "send before receive" order for message transmission between sites.

in N copies (or replicas), where $N = f + 1$ in the case of fail-stop failures, and $N = 2f + 1$ in the case of Byzantine failures (in this latter case, a majority of correct processes is needed). There are two main approaches to ensure that the system is available, as long as the number of faulty replicas is less or equal to f.

- *Coordinator-based replication.* In this approach, also called *primary-backup* [Schneider 1993a], a particular replica is chosen as coordinator (or primary); the other replicas are called back-ups. The requests are sent to the primary. The primary's task is (i) to process the requests and to send the replies; and (ii) to keep the back-ups consistent. If the primary fails, one of the back-ups becomes the new primary. As shown in 11.6, this technique relies on a particular communication protocol (reliable view-synchronous broadcast), which guarantees that, whenever the primary fails, the new primary is in a state that allows it to correctly fulfill its function.

- *Active replication.* In this approach, also called *replicated state machine* [Lamport 1978a, Schneider 1990], all replicas have the same role, and each request is directly sent to each replica. Since the behavior of the SM is deterministic, a sufficient condition for the N replicas to behave identically (i.e., to keep the same state and to send the same replies) is that *all replicas receive the same requests, in the same order*. Achieving fault tolerance with active replication therefore relies on a particular communication protocol (causal totally ordered broadcast), which is discussed in more detail in 11.3.2 and 11.4.2, together with other forms of group communication.

These approaches define two basic patterns that are recurrent in fault tolerance techniques. System implementations based on both approaches are described in more detail in 11.6.

11.3.2 Group Communication

Since fault tolerance is based on replication, process groups play a central role in the design of fault tolerant systems. For example, as seen in 11.2.3, a group of processes can be organized as a reliable implementation of a single process, using the replicated state machine approach. However, this places specific requirements on communication within the group (in this case, totally ordered broadcast is required).

In this section, we introduce the main concepts and terminology of process groups and we discuss the requirements of group communication and membership protocols. The implementation of these protocols is subject to impossibility results, which are examined in 11.3.3. Practical approaches to group communication are the subject of 11.4.

A *process group* is a specified set of processes (the members of the group), together with protocols related to:

- Group communication (broadcast or multicast), i.e., sending a message to the members of the group, with specified requirements.

- Group membership, i.e., changing the composition of the group. The system maintains knowledge about the current composition of the group, represented by a *view* (a list of processes).

A global picture of group protocols is shown in Figure 11.3.

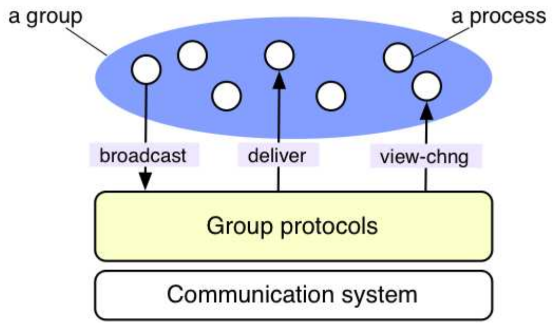


Figure 11.3. Group protocols

A typical API for group protocols includes the following primitives:

- $broadcast(p, m)$. Process p broadcasts message m to the group.
- $deliver(p, m)$. Message m is delivered to process p.
- $view\text{-}chng(p, id, V)$. Following a change in the group membership, a new view, V, identified by number id, is delivered to process p.

Additional primitives may be provided by specific systems (examples are given below).

Process groups are useful for implementing fault tolerant systems, and also for supporting information sharing and collaborative work.

Group protocols are an important subject of ongoing research, because two of their main aspects are still not well understood. First, specifying a group protocol is a non-trivial task. Group protocol specification is the subject of [Chockler et al. 2001], who note that most existing specifications (at the date of writing) are incomplete, incorrect, or ambiguous. Second, implementing fault tolerant group protocols is often algorithmically difficult, or, in some cases, impossible (see 11.3.3).

We now review the specifications of the most usual protocols.

Group communication protocols

In group communication primitives, a process (the *sender*) sends a message to a set of destination processes (the *receivers*). There are two main forms of group communication primitives, which differ by the definition of the receivers.

- *Broadcast*. The receivers are the processes of a single process set, which may be explicitly or implicitly defined, and which includes the sender. Examples: all members of a process group; "all" processes of a system.
- *Multicast*. The receivers are the members of one or several process groups, which may or not overlap. The sender may or not be part of the receivers.

The main problems specific to multicast are related to overlapping destination groups. We only consider here the case of broadcast.

The specifications of broadcast are discussed in detail in [Hadzilacos and Toueg 1993] and [Chockler et al. 2001]. Assume a fail-stop failure mode (without repair) for processes, and a reliable, asynchronous communication system. A process is said to be *correct* if it does not fail.

The minimal requirement for a broadcast primitive is that it be *reliable*, which is an "all or nothing" property: a message must be delivered to all of its correct receivers, or to none. More precisely, if a message is delivered to *one* correct receiver, it must be delivered to *all* correct receivers. A broadcast protocol that is not reliable is not of much practical use. The value of reliability lies in the shared knowledge that it implies: when a broadcast message is reliably delivered to a correct process, the process "knows" that this message will be delivered to all correct processes in the destination set.

A much stronger requirement is that of total order. A *totally ordered*, or *atomic*, broadcast is a reliable broadcast in which all receivers get the messages in the same order. More precisely, if two messages, m_1 and m_2, are delivered to a correct process p in that order, then m_2 may not be delivered to another correct process q unless m_1 has been first delivered to q.

The two above requirements are independent of the order in which messages are being sent. Another set of requirements involves the sending order:

FIFO broadcast: two messages issued by the same sender must be delivered to all receivers in their sending order. More precisely, if a process has broadcast m_1 before m_2, then m_2 may not be delivered to a correct process q unless m_1 has been first delivered to q.

Causal broadcast: If the sending events of two messages are causally ordered (see 11.2.3), then the messages must be delivered to any receiver in their causal order. More precisely, if the sending of m_1 happened before the sending of m_2, then m_2 may not be delivered to a correct process q unless m_1 has been first delivered to q. FIFO is a special case of causal broadcast.

These requirements are orthogonal to those of reliability and atomicity, which leads to the main six forms of broadcast summarized on Figure 11.4.

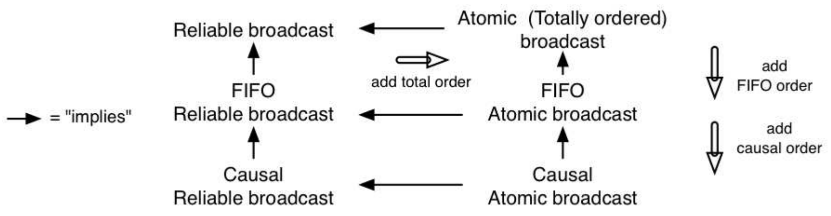


Figure 11.4. Requirements for broadcast protocols (adapted from [Hadzilacos and Toueg 1993])

An additional requirement, again orthogonal to the previous ones, is uniformity. In a *uniform* broadcast, we are also interested in the behavior of faulty processes. For example, reliable uniform broadcast is specified as follows: if a message m is delivered to a (correct or faulty) process, then it must be delivered to all correct processes. The motivation behind this requirement is that the delivery of a message may trigger an irreversible action, and a process may get a message and perform that action before failing.

The above specification assume that the group of processes is static, i.e., processes cannot join or leave the group (they may only crash). The specification may be extended to dynamic groups [Schiper 2006a], using the group membership service defined below.

Group membership protocols

Recall that a group (in the present context) is a set of processes, the members of the group, together with an API for communication and membership. We now assume that the composition of the group changes over time: members may leave the group, or fail; new processes can join the group, and failed processes can be repaired and join the group again. The aim of group membership protocols is to implement the join and leave operations, and to keep the members of the group informed about the current membership, by delivering a sequence of views.

There are two versions of group membership protocols. The *primary partition* version assumes that the sequence of views is totally ordered. This is the case if the process group remains connected (i.e., all its members can communicate with each other), or if one only considers a single partition in a disconnected group. In the *partitionable* version, the sequence of views is partially ordered. This applies to a partitioned group, in which several partitions are taken into account. We only consider the primary partition version.

As mentioned above, writing precise and consistent group membership protocol specifications is a surprisingly difficult task. Recall (1.4.2) that the required properties are categorized into *safety* (informally: no undesirable event or condition will ever occur) and *liveness* (informally: a desirable event or condition will eventually occur). These requirements are summarized as follows (see [Chockler et al. 2001] for formal statements).

Recall that the group membership service delivers views to the members of the group. A *view* is a list of processes that the service considers to be current members of the group. When a view v is delivered to process p, p is said to *install* the view.

The safety properties include:

- Validity. A process which installs a view is part of that view (self-inclusion property).
- Total order. The set of views installed by the processes is totally ordered (thus one may speak of the next view, of consecutive views, etc.).
- Initial view. There exists an initial view, whose members are explicitly defined.
- Agreement. The sequence of views installed by a process is a subsequence of *contiguous* elements of the global sequence of views (consistency property).
- Justification. A change of view (installing a new view) must be motivated by one of the following events: a process has joined the group, left the group, or failed.
- State transfer. The state of the group (i.e., the contents of a view) may be transferred to the next view in the sequence, except if all processes have failed. “State transfer” means that at least one process that installed the new view was included in the predecessor of that view.

Liveness is defined by the following property: any event associated with a given process (join, leave, or failure) will eventually be visible in a further view (except if a process fails after joining, in which case it may never install a view).

A recent proposal [Schiper and Toueg 2006] suggests to consider group membership as a special case of a more general problem, set membership, in which the processes of the group may add or remove elements of a set, and need to agree on the current composition of the set. The members of the set are drawn from an arbitrary universe. Group membership is the special case in which the members of the set are processes. This approach favors separation of concerns (1.4.2), by identifying two separate issues: determining the set of processes that are deemed to be operational, and ensuring agreement over the successive values of that set. Different algorithms are needed to solve these two problems.

Virtual synchrony

One important aspect of group protocols is the connection between group membership and communication through the notion of *virtual synchrony*, first introduced in [Birman and Joseph 1987]. Virtual synchrony defines a broadcast protocol (view synchronous broadcast) that is consistent with group membership, as implemented by the view mechanism.

The specification of view synchronous broadcast extends the agreement condition defined for group membership. The extended conditions are as follows.

- *Agreement* for view synchronous communication. Two properties hold:
  - The (correct) members of the process group install the same sequence of totally ordered views of the group.
  - The (correct) members of the process group see the same sequence of delivered messages between the installation of a view v_i and that of the next view v_{i+1}.
- *Justification* for view synchronous communication: any message delivered has actually been sent by some process .
- *Liveness* for view synchronous communication: any message sent by a correct process is eventually delivered.

These conditions are illustrated by the examples shown in Figure 11.5. In all cases, process p_1 broadcasts a message to a group including itself and p_2, p_3, p_4, all alive in view v_i. In case (a), there is no view change, and the system behaves correctly. In case (b), p_1 crashes during broadcast, and the message is delivered to p_3 and p_4 but not to $p2$. This violates virtual synchrony, since the surviving processes in view v_{i+1} have not received the same set of messages. This also happens in case (c), for a different reason: although the message eventually gets to p_4, it is only delivered after p_4 has installed the new view; p_4 is inconsistent in the meantime. Finally, virtual synchrony is respected in case (d), since the surviving processes in view v_{i+1} have received the same set of messages when the view is installed.

One common use of virtual synchrony is illustrated by the example of a replicated database. Each replica is under the control of a manager process, and updates are (atomically) broadcast to all managers. If this broadcast does not respect virtual synchrony,

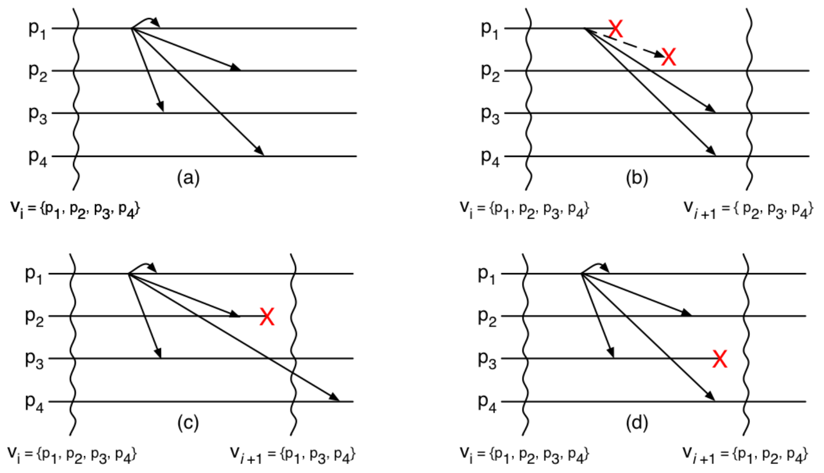


Figure 11.5. Virtual synchrony

one ore more replica(s) will become inconsistent after a view change following the crash of one of the managers. Virtual synchrony ensures that a reader can consult any of the replicas; it may get an out of date version (one that has not yet received the last update), but not an inconsistent one. Data replication is discussed in more detail in 11.7. Another application of virtual synchrony (fault tolerant servers) is presented in 11.6.2.

There is a close relationship between group membership and group communication. Actually, each of these services can be implemented on top of the other one. In most usual implementations, the group membership layer lies beneath the group communication protocol stacks. In [Schiper 2006a], a different approach is proposed, in which atomic broadcast is the basic layer, and group membership is implemented above this layer. The claimed advantage is simplicity, and better efficiency in most practical situations.

Both group membership and atomic broadcast can be expressed in terms of an agreement protocol, consensus, which we now examine.

11.3.3 Consensus and Related Problems

Consensus is a basic mechanism for achieving agreement among a set of processes. It can be shown that both the total order broadcast and the virtually synchronous group membership protocols are equivalent to consensus. The specifications and implementations of consensus therefore play a central role in the understanding of fault tolerant distributed algorithms. It has been proposed [Guerraoui and Schiper 2001] that a generic consensus service be provided as a base for building group membership and group communication protocols.

Consensus Specification

Given a set of processes $\{p_i\}$ linked by a communication system, consensus is specified as follows. Initially, each process p_i proposes a value v_i. If the algorithm terminates, each process p_i decides a value d_i. The following conditions must hold:

- *Agreement.* No two correct processes decide different values.
- *Integrity.* Each process decides at most once (i.e., if it decides, the decision may not be modified).
- *Validity.* A decided value must be one of those proposed (this condition eliminates trivial solutions, in which the processes decide some predetermined value).
- *Termination.* If at least one correct process starts the algorithm, all correct processes decide in finite time.

A process is correct if it does not fail.

Solving consensus in the absence of failures is a simple task. It may be done in two ways, corresponding to the patterns identified in 11.3.1.

1. One process is defined as the coordinator (or *primary*). On the primary's request, each process sends its proposed value to the primary. After it has received all values, the primary chooses one of them as the decided value, and sends it to all processes. When it receives a value from the primary, each process decides that value.
2. Each process sends its proposed value to every process (including itself). Once a process has received proposed values from all the p_is, it applies a deterministic decision algorithm (the same one for all processes) to choose a value among those received, and it decides that value.

In both cases, if we assume reliable communication, the algorithm terminates. This is true even if communication is asynchronous, although the termination time is not bounded in that case.

The two above approaches are the base of the algorithms used to solve consensus in the presence of failures. In that case, achieving consensus becomes much more complex, and in many cases intractable, as discussed below.

We now present (without proof) a few important results on the tractability of the problems associated with group protocols. The main impossibility results concern asynchronous systems and are linked to the impossibility of "perfect" failure detection in such systems. We then discuss the main approaches to the solution of agreement problems, which are the base of group protocols.

Limits of Group Protocol Algorithms

Reliable broadcast can be implemented in an asynchronous system, using a "flooding" algorithm. When a process receives a message, it transmits the message to all its neighbors (the processes to which it is connected by a direct link), and then delivers the message to itself. To broadcast a message, the sender sends it to itself, and follows the above rule. This algorithm implements the specification of reliable broadcast; in addition, it is uniform, i.e., if a message is delivered to a (correct or faulty) process, then it is delivered to all correct processes. Uniformity is guaranteed by the fact that a process delivers a message to itself only *after* having sent it to its neighbors. Since the system is asynchronous, there is no upper bound on the delivery time of a message.

This algorithm can be extended to achieve FIFO and causal reliable broadcast [Hadzilacos and Toueg 1993].

The central impossibility result, often known as FLP from its authors' names Fischer, Lynch, and Paterson [Fischer et al. 1985], is the following: in an asynchronous system, there is no deterministic algorithm that achieves consensus in finite time if one process (at least) may fail. Recall the assumptions: the failure mode is fail-stop, and communication is reliable. This impossibility extends to totally ordered broadcast and to view-synchronous group membership, since these algorithms are equivalent to consensus.

With the same assumptions, it has been demonstrated [Chandra et al. 1996b] that group membership cannot be solved in an asynchronous system.

How to live with these limitations? Three main tracks have been explored.

- Use randomization. The FLP result only applies to deterministic algorithms. Randomized algorithms have been devised to achieve consensus in an asynchronous system (see a review in [Aspnes 2003]).
- Relax the asynchrony assumption. Various forms of "partial synchrony" have been investigated [Dwork et al. 1988]. We use that specified in 11.1.4.
- Use an imperfect algorithm, i.e., one that may fail, but which is the best possible in some sense. Two main approaches have been proposed: the Paxos algorithm, and unreliable failure detectors. They are examined below.

Most practical approaches to group protocols combine the use of one of the above algorithms and a partial synchrony assumption. They are discussed in Section 11.4.

The Paxos consensus protocol

The first approach to solving consensus is known as the Paxos protocol. The failure hypotheses are weaker than indicated above: the network may lose messages, and the processes may fail by crashing, and subsequently recover. Paxos was introduced in [Lamport 1998]; an equivalent protocol was included in the replicated storage system of [Oki and Liskov 1988]. A pedagogical description may be found in [Lampson 2001].

The protocol works roughly as follows. The system goes through a sequence of views. A *view* defines a period during which one selected process, the primary, attempts to achieve agreement (using the same principle as the primary-based algorithm described in 11.3.2 in the absence of failures). After a view is started as the result of a view change, the protocol works in two phases. In the first phase, the primary queries the participants for their state (i.e., their replies in past views). Then the primary selects an "acceptable" value (in a sense defined later), and proposes that value[5]. If a majority agrees on that value, consensus is achieved, and the decision is propagated to the participants. Thus the protocol succeeds if the primary and a majority of processes live for a sufficiently long time to allow two and a half rounds of exchanges between the primary and the other processes. If either a majority cannot be gathered or the primary fails before agreement is reached or progress

[5]When the protocol is started by a client' request (as opposed to a view change), the value is that proposed by the client

is “too slow”, a new primary is elected[6], and a new view is started. The view change protocol is the choice of a new primary, followed by the first phase described above. The “normal” protocol achieves safety (agreement on a value), while the view change protocol guarantees liveness (i.e., progression).

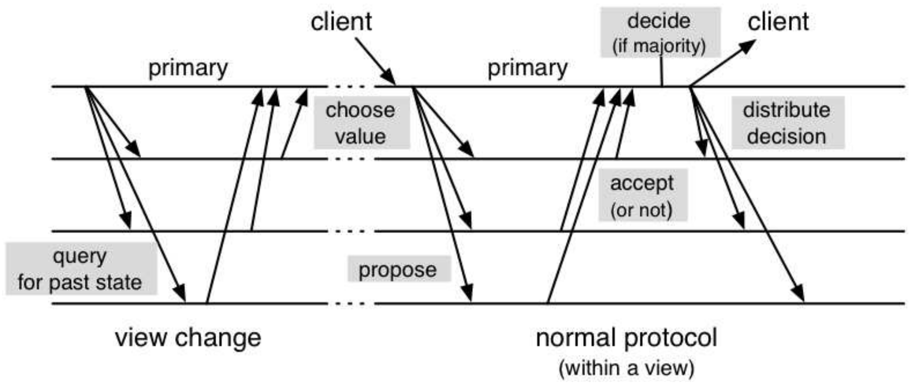


Figure 11.6. The Paxos consensus protocol

Paxos does not require that a single primary should be present at a time. Several views (and therefore several primaries) may coexist, depending on how the election process was started upon detecting failure to agree. However, Paxos imposes a total order on views, allowing processes to reject the values proposed by any primary that is not in the most recent view. A key point is that Paxos restricts the primary's choice of acceptable values: if a majority has once agreed on a value in some view (and a decision has not been reached in that view because of failures), only that value may be proposed in subsequent views. This stability property ensures that no two different values may be decided, a safety property.

The actual working of Paxos is much more subtle than shown in the above rough description (see references above). In addition, a number of optimizations may be made, which complexify the description. Paxos is not guaranteed to ensure agreement in finite time (since FLP still holds), but it is extremely robust in the presence of failures and unknown delays.

One practical use of Paxos is consistency management in replicated storage systems (11.7). See [Oki and Liskov 1988] for an early application, and [Chandra et al. 2007] for a recent one.

Unreliable failure detectors

The idea of *unreliable* failure detectors [Chandra and Toueg 1996] is based on the following intuitive remark: the intractability of consensus in asynchronous systems is linked to the impossibility of telling a faulty process from a slow one, i.e., to the impossibility of building a “perfect” failure detector. This leads to the following questions: (i) what are the properties of a perfect failure detector? and (ii) is consensus solvable with an “imperfect” failure detector, and, if so, what properties are required from such a detector?

[6]The election protocol may be very simple, as long as it selects one primary per view. Conflicts between coexisting views are solved as explain later.

A *failure detector* is a device that can be used as an oracle: it delivers upon request the list of processes that it suspects of being faulty. In a distributed system, a failure detector is itself a distributed program, made up of cooperating components, one on each site. To use the detector, a process consults the local failure detector component.

Two properties have been identified for a *perfect* failure detector: *strong completeness*, i.e., every faulty process is eventually suspected by every correct process; and *strong accuracy*, i.e., no correct process is ever suspected by any correct process. Thus, there is a time after which the list delivered by a perfect detector contains all the faulty processes, and only those processes.

These properties can be weakened[7], allowing several classes of imperfect (or unreliable) detectors to be defined. Weak accuracy means that *some* correct process is never suspected by any correct process. Both strong and weak accuracy may be *eventual*; in that case, there exists a time after which the indicated property holds.

One may then define the following classes[8] of detectors by their properties:

- *Perfect* (P): strong completeness, strong accuracy.
- *Strong* (S): strong completeness, weak accuracy.
- *Eventually Perfect* ($\diamond P$): strong completeness, eventually strong accuracy.
- *Eventually Strong* ($\diamond S$): strong completeness, eventually weak accuracy.

The detectors other than P are unreliable, i.e., they can make false suspicions.

The most important result [Chandra and Toueg 1996] is the following: in an asynchronous system with fail-stop failures, consensus can be solved using a detector in any of the classes P, S, $\diamond P$, $\diamond S$. Moreover [Chandra et al. 1996a], $\diamond S$ is the weakest detector allowing consensus.

More precisely, the solutions to the consensus problem among n processes, using failure detectors, have the following properties.

- With P: allows up to $n-1$ failures, needs $f+1$ rounds, where f is the number of failures tolerated.
- With S: allows up to $n-1$ failures, needs n rounds.
- With $\diamond S$: allows up to $\lceil n/2 \rceil - 1$ failures, needs a finite (but unbounded) number of rounds.

The consensus algorithm for a $\diamond S$ detector is based on the principle of the *rotating coordinator*. The coordinator tries to achieve agreement among a majority of processes, based on the proposed values. This may fail, because the coordinator may be falsely suspected or may crash. The function of coordinator rotates among the processes. Eventually (as guaranteed by the properties of $\diamond S$), a correct process that is never suspected

[7]We do not need to consider weak completeness (every faulty process is eventually suspected by *some* correct process), because a weakly complete detector is easily converted into a strongly complete one, by making each process reliably broadcast the list of suspected processes.

[8]A detector belongs to a given class if it has the properties specified for that class. In the following, we often use the term "detector C" to denote a detector of class C, where C is one of P, ... $\diamond S$.

will become coordinator, and will achieve agreement, provided that a majority of correct processes exists.

None of the above detectors may be implemented in an asynchronous system by a deterministic algorithm (if this were the case, it would contradict the FLP result). Implementations under a partial synchrony assumption are described in 11.4.1.

Paxos and unreliable failure detectors are two different approaches to solving consensus. A detailed comparison of these approaches is still an open issue. The interest of Paxos is that it ensures a high degree of tolerance to failures, (including message loss), and that it may be extended to Byzantine failures. The interest of unreliable failure detectors is that they allow a deep understanding of the origins of the difficulty of achieving consensus, and that they provide guidelines for actual implementations (see 11.4.1).

11.4 Practical Aspects of Group Communication

In this section, we examine the actual implementation of group protocols. We start by discussing implementation issues for failure detectors (11.4.1). We then look at atomic broadcast (11.4.2) and group membership (11.4.3). We conclude with a discussion of group protocols for large scale systems (11.4.4).

11.4.1 Implementing Failure Detectors

We first present the basic assumptions and mechanisms that underlie the implementations of failure detectors. We then discuss the quality factors of these detectors, and conclude with a few practical considerations.

Basic Mechanisms of Failure Detection

Recall (11.3.3) that none of the failure detectors P, S, $\diamond P$, $\diamond S$ may be implemented by a deterministic algorithm in an asynchronous system. The approach usually taken in practice is to relax the asynchrony assumption, by considering a particular form of *partial synchrony*: assume that bounds exist for the message transmission time and for the speed ratio between processes, but that these bounds are unknown *a priori*, and hold only after a certain time. This assumption is realistic enough in most common situations, and it allows the use of timeouts for implementing failure detectors. Recall that the failure model for processes is fail-stop, without recovery.

Failure detection relies on an elementary mechanism that allows a process q to determine if a process p is correct or faulty, assuming an estimated upper bound $\Delta_{p,q}$ is known for the transmission time[9] of a message between p and q. Two variants of this mechanism have been proposed (Figure 11.7).

- *Pull*, also called *ping*. Process q periodically sends a request "are you alive?" to process p, which answers with a message "I am alive". If no answer is received after $2\Delta_{p,q}$, q suspects p of having crashed.

[9] also assume, for simplicity, that $\Delta_{p,q} = \Delta_{q,p}$

- *Push*, also called *heartbeat*. Process p periodically sends to q (and possibly to other processes) a message "I am alive". Assume that p and q have synchronized clocks, and that q knows the times t_i at which p sends its message. If, for some i, q has not received the heartbeat message from p by time $t_i + \Delta_{p,q}$, q suspects p of having crashed.

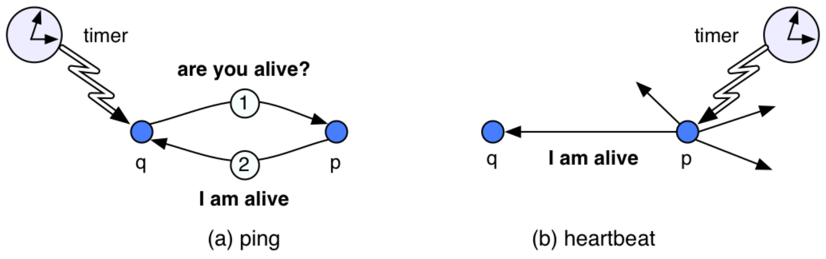


Figure 11.7. Principle of ping and heartbeat

The above description only gives the general principle, and various details need to be filled in. In particular, how is $\Delta_{p,q}$ estimated? One common technique is to dynamically determine successive approximations to this upper bound for each process, by starting with a preset initial value. If the value is found to be too small (because a heartbeat message arrives after the deadline), it is increased (and the falsely suspected process is removed from the suspects' list). With the partial synchrony assumption, this technique ensures convergence towards the eventual upper bound.

Another important design choice for a failure detection algorithms is the communication pattern between the processes. Two main patterns have been considered.

- Complete exchange (or all-to-all communication). Every process communicates with every other one. For n processes, the number of messages is thus of the order of n^2 per run (more precisely nC, where C is the number of faulty processes in the considered run).

- Exchange over a logical (or virtual) ring. Each process only communicates with some[10] of his successors or predecessors on the ring. The number of messages is now linear in n.

In addition, hierarchical patterns (with separate intra-group and inter-group communication) have been introduced for large scale systems (see 11.4.4).

Various combinations of the design choices indicated above (ping or heartbeat, determining the starting point for the watchdog timers, estimating an upper bound for the communication delay, choosing a communication pattern) have been proposed. A summary of some proposals follows.

In their seminal paper on unreliable failure detectors, [Chandra and Toueg 1996] propose an implementation of a failure detector $\diamond P$, assuming partial synchrony (as specified

[10]Typically, if an upper bound f is known for the number of tolerated failures, a process needs to communicate with its $f + 1$ successors to ensure it will reach a correct process.

at the beginning of this section). This implementation uses heartbeat, all-to-all communication, and adaptive adjustment of the transmission time bound, as described above. The main drawback of this implementation is its high communication cost (quadratic in the number of processes).

To reduce the number of messages, [Larrea et al. 2004] propose implementations of $\diamond P$ and $\diamond S$ using a logical ring, a ping scheme for failure detection, and again an adaptive adjustment of the transmission time bound. Each process q monitors (i.e., "pings") a process p, called the "target" of q; initially, this target is $succ(q)$, the successor of q on the ring. If q suspects p, it adds it to its suspects lists, and its new target becomes $succ(p)$; if q stops suspecting a process r, it also stops suspecting all processes between r and its current target, and r becomes its new target. The drawback of this algorithm is that each process only maintains a "local" (partial) suspects list. To build the global list (the union of all local lists), each local list needs to be broadcast to all processes. While this transmission may be optimized by piggybacking the local list on top of ping messages, this delays the detection, by a process q, of faulty processes outside q's own local list. Several variations of this scheme, still based on a logical ring but using heartbeat instead of ping, are proposed in [Larrea et al. 2007].

[Mostefaoui et al. 2003] take a different approach, by changing the assumptions on the communication system. They do not assume partial synchrony, but introduce assumptions involving an upper bound f on the number of faulty processes. These assumptions amount to saying that there is at least one correct process that can monitor some process (i.e., receive responses to its ping messages to that process).

There does not appear to exist a "best" failure detector, due to the wide range of situations, and to the variety of cost and quality metrics. The quality metrics are discussed below.

Quality Factors of Failure Detectors

A comprehensive investigation of the quality of service (QoS) factors of failure detectors is presented in [Chen et al. 2002]. This work was motivated by the following remarks:

- The properties of failure detectors (11.3.3) are specified as eventual. While this quality ensures long term convergence, it is inappropriate for applications that have timing constraints. The *speed* of detection is a relevant factor in such situations.
- By their nature, unreliable failure detectors make false suspicions. In practical situations, one may need a good *accuracy* (reducing the frequency and duration of such mistakes).

[Chen et al. 2002] propose three primary QoS metrics for failure detectors. The first one is related to speed, while the other two are related to accuracy. They also define additional quality metrics, which may be derived from the three primary ones.

The primary metrics are defined as follows, in the context described at the beginning of 11.4.1, i.e., a system composed of two processes p and q, in which q monitors the behavior of p, and q does not crash (we only give informal definitions; refer to the original paper for precise specification).

- Detection time. This is the time elapsed between p's crash and the instant when q starts suspecting p permanently.
- Mistake recurrence time. This is the time between two consecutive mistakes (a mistake occurs when q starts falsely suspecting p). This is analogous to the notion of $MTTF$ as defined in 11.1.1 (although the "failure" here is a faulty behavior of the detector).
- Mistake duration. This is the time that it takes to the failure detector to correct a mistake, i.e., to cease falsely suspecting a correct process. This is analogous to the notion of $MTTR$ as defined in 11.1.1.

Since the behavior of the system is probabilistic, the above factors are random variables. More precisely, they are defined by assuming the system has reached a steady state, i.e., a state in which the influence of the initial conditions has disappeared[11]. Note that these metrics are implementation-independent: they do not refer to a specific mechanism for failure detection.

[Chen et al. 2002] present a heartbeat-based algorithm, and show that it can be configured to reach a specified QoS (as defined by the above primary factors), if the behavior of the communication system is known in terms of probability distributions for message loss and transmission time. The algorithm makes a best effort to satisfy the QoS requirements, and detects situations in which these requirements cannot be met. Configuration consists in choosing values for the two parameters of the detector: η, the time between two successive heartbeats, and δ, the time shift between the instants at which p sends heartbeats and the latest instants[12] at which q expects to receive them before starting to suspect p. If the behavior of the communication system varies over time, the algorithm may be made adaptive by periodically reevaluating the probability distributions and re-executing the configuration algorithm.

Concluding Remarks

The results discussed above rely on several assumptions. Are these assumptions valid in practice?

Concerning the partial synchrony assumption, observation shows that a common behavior for a communication system alternates between long "stable" phases, in which there is a known upper bound for transmission times, and shorter "unstable" phases, in which transmission times are erratic. This behavior is captured by the partial synchrony model, since an upper bound for the transmission time eventually holds at the end of an unstable phase.

Concerning reliable communication, the "no message loss" assumption is justified for LANs and WANs, in which message retransmission is implemented in the lower layer protocols. It is questionable in mobile wireless networks.

[11]In practice, steady state is usually reached quickly, typically after the receipt of the first heartbeat message in a heartbeat-based detector

[12]The basic algorithm assumes that the clocks of p and q are synchronized; it can be extended, using estimation based on past heartbeats, to the case of non-synchronized clocks.

Concerning the fail-stop failure model, the assumption of crash without recovery is overly restrictive. In practice, failed components (hardware or software) are repaired after a failure and reinserted into the system. This behavior can be represented in two ways.

- Using dynamic groups. A recovered component (represented by a process) can join the group under a new identity.
- Extending the failure model. A crash failure model with recovery has been investigated in [Aguilera et al. 2000].

In all cases, recovery protocols assume that a form of reliable storage is available.

Concerning the basic failure detection mechanisms, both heartbeat and ping are used, since they correspond to different tradeoffs between efficiency (in terms of number of messages) and accuracy. Heartbeat seems to be the most common technique. Since ping uses more messages, it is usually associated with a specific organization of the interprocess links, such as a tree or a ring, which reduces the connectivity.

11.4.2 Implementing Atomic Broadcast

A large number of atomic broadcast and multicast algorithms have been published. [Défago et al. 2004] give an extensive survey of these algorithms and propose a taxonomy based on the selection of the entity used to build the order. They identify five classes, as follows.

- *Fixed sequencer.* The order is determined by a predefined process, the sequencer, which receives the messages to be broadcast and resends them to the destination processes, together with a sequence number. The messages are delivered in the order of the sequence numbers. This algorithm is simple, but the sequencer is a bottleneck and a single point of failure.
- *Moving sequencer.* Same as above, but the role of sequencer rotates among the processes. The advantage is to share the load among several processes, at the price of increased complexity.
- *Privilege-based.* This class of algorithms is inspired by the mutual exclusion technique using a privilege in the form of a token, which circulates among the processes. A process can only broadcast a message when it holds the token. To ensure message ordering, the token holds the sequence number of the next message to be broadcast. This technique is not well suited for dynamic groups, since inserting or removing a process involves a restructuring of the circulating pattern of the token. Also, special care is needed to ensure fairness (such as specifying a limit to the time that a process holds the token).
- *Communication History.* The delivery order is determined by the senders, but is implemented by delaying message delivery at the destination processes. Two methods may be used to determine the order: causal time-stamping, using a total order based on the causal order, and deterministic merge of the message streams coming from each process, each message being independently (i.e., non causally) timestamped by its sender.

- *Destinations Agreement*. The delivery order is determined through an agreement protocol between the destination processes. Here again, there are several variants. One variant uses a sequence of consensus runs to reach agreement over the sequence of messages to be delivered. Another variant uses two rounds of time-stamping: a local time-stamp is attached to each message upon receipt, and a global time-stamp is determined as the maximum of all local timestamps, thus ensuring unique ordering (ties between equal timestamps are resolved using process numbers).

In addition, [Défago et al. 2004] describe a few algorithms that are hybrid, i.e., that mix techniques from more than one of the above classes.

Two delicate issues are (i) the techniques used to make the algorithms fault-tolerant; and (ii) the performance evaluation of the algorithms. This latter aspect often relies on local optimization (e.g., using piggybacking to reduce the number of messages, or exploiting the fact that, on a LAN, messages are most often received in the same order by all processes). Therefore, the algorithms are difficult to compare with respect to performance.

Regarding fault tolerance, two aspects need to be considered: failures of the communication system, and failures of the processes.

Many algorithms assume a reliable communication system. Some algorithms tolerate message loss, using either positive or negative acknowledgments. For example, in a fixed sequencer algorithm, a receiving process can detect a “hole” in the sequence numbers, and request the missing messages from the sequencer. As a consequence, the sequencer needs to keep a copy of a message until it knows that the message has been delivered everywhere.

Concerning process failures, most of the proposed algorithms rely on a group membership service, itself based on a failure detector. Others directly use a failure detector, either explicitly, or implicitly through a consensus service. The relationship between atomic broadcast, group membership and consensus is briefly discussed in the next section.

11.4.3 Implementing Virtually Synchronous Group Membership

There are three main approaches to group membership implementation.

- Implementing group membership over a failure detector (itself implemented by one of the techniques described in 11.4.1). This is the most common approach. An example using it is described below.

- Implementing group membership over atomic broadcast (itself implemented by one of the techniques described in 11.4.2). [Schiper 2006a] gives arguments in favor of this approach.

- Implementing both atomic broadcast and group membership over a consensus service (itself usually based on a failure detector). See [Guerraoui and Schiper 2001].

We illustrate the first approach (using a failure detector to implement virtual synchronous group membership) with the example of JavaGroups [Ban 1998], itself derived from the Ensemble protocol stack [van Renesse et al. 1998]. The protocol uses the coordinator pattern and is based on the notion of stable messages. A message is said to be *stable* when it is known to having been delivered to every member of the current view of the group. Otherwise, it is *unstable*[13].

A view change is triggered by one of the following events (11.3.2): a process joins the group, leaves the group, or crashes (this latter event is observed by a failure detector). The coordinator is notified when any of these events occurs. It then initiates a view change by broadcasting a FLUSH message to the group. When a process receives the FLUSH message, it stops sending messages (until it has installed the new view), and sends all the messages it knows to be unstable to the coordinator, followed by a FLUSH_OK message. When it has received all FLUSH_OK replies, the coordinator broadcasts all the unstable messages (messages are uniquely identified within a view, by a sequence number, to avoid duplicates). Recall that the underlying communication system is reliable, so that the messages will reach their destination if the sender is correct. When all messages have become stable, the coordinator broadcasts a VIEW message, which contains the list of the members of the new view. When a process receives this message, it installs the new view.

In order to detect failures, the processes are organized in a logical ring, and each process pings its successor (as described in 11.4.1). If a process (other than the coordinator) crashes during this view change protocol, a notification is sent to the coordinator, which starts a new view change. If the coordinator crashes, its predecessor in the ring (which detected the crash) becomes the new coordinator, and starts a new view change.

The reliable message transmission layer uses both a positive (ACK) and negative (NAK) acknowledgment mechanism. Negative acknowledgment is based on the sequential numbering of messages, and is used when the received sequence number differs from the expected one. For efficiency, the default mode is NAK for ordinary messages. The ACK mode is used for sending view changes (the VIEW message).

This protocol does not scale well (the group size is typically limited to a few tens of processes). A hierarchical structure based on a spanning tree allows a better scalability. The processes are partitioned into local groups; in each local group, a local coordinator (the group controller) broadcasts the flush and view messages within the group. The controllers cooperate to ensure global message diffusion. This allows efficient communication for groups of several hundred processes.

Other communication toolkits include Appia [Appia] and Spread [Spread].

11.4.4 Large Scale Group Protocols

This section still unavailable

11.5 Fault Tolerance Issues for Transactions

11.5.1 Transaction Recovery

This section still unavailable.

[13]Typically, a message that was broadcast by a process that failed during the broadcast may be unstable

11.5.2 Atomic Commitment

This section still unavailable.

11.6 Implementing Available Services

In this section and the next one, we examine how available systems can be built using replication. This section deals with available services, while section 11.7 discusses available data. Both cases involve replicated entities, which raises the issue of keeping these entities mutually consistent. We examine consistency conditions in 11.6.1.

Recall (2.1) that a *service* is a contractually defined behavior, implemented by a component and used by other components. In the context of this discussion, we use the term *server* to designate the component that implements a service (specified by its provided interface), together with the site on which it runs. Making a service highly available is achieved by replicating the server. Two main approaches are used, corresponding to the two patterns identified in 11.3.1, assuming fail-stop failures: the primary-backup protocol (11.6.2) and the active replication protocol (11.6.3). These protocols are discussed in detail in [Guerraoui and Schiper 1997].

The case of Byzantine failures is the subject of 11.6.4.

11.6.1 Consistency Conditions for Replicated Objects

Defining consistency conditions for replicated data is a special case of the more general problem of defining consistency for concurrently accessed data. This problem has been studied in many contexts, from cache coherence in multiprocessors to transactions. Defining a consistency condition implies a trade-off between strong guarantees and efficient implementation. Therefore, a number of consistency models have been proposed, with different degrees of strictness.

Consider a set of shared data that may be concurrently accessed by a set of processes, which perform *operations* on the data. Operations may be defined in different ways, according to te application context. In a shared memory, operations are elementary reads and writes. In a database system, operations are transactions (see Chapter 9). In a (synchronous) client-server system (see Chapter 5), an operation, as viewed by the server, starts when the server receives a client's request, and ends when the server sends the corresponding reply to the client.

A commonly used consistency requirement for concurrently accessed data is *sequential consistency* [Lamport 1979], which specifies that a set of operations, concurrently executed by the processes, has the same effect as if the operations had been executed in some sequential order, compatible with the order seen at each individual process. In the context of transactions, this requirement is usually called *serializability* (9.2.1).

A stronger requirement, known as *linearizability* [Herlihy and Wing 1990], is used for shared objects. A shared object may be seen as a server that receives requests from client processes to perform operations. A partial order is defined on the operations, as follows: operation O_1 precedes operation O_2 (on the server) if the sending of the result of O_1 precedes the receipt of the request of O_2. An execution history (a sequence of operations on the object) is said to be linearizable if (i) it is equivalent to a legal sequential execution (legal means "satisfying the specification of the object"); and (ii) the sequential order of the operations is compatible with their ordering in the initial history. Linearizability thus extends sequential consistency, specified by condition (i). Intuitively, linearizability may be interpreted as follows: each operation appears to take effect instantaneously, at some point in time between its invocation and its termination; and the order of operations respects their "real time" order.

Linearizability thus appears to be more intuitive that sequential consistency, since it preserves the order of (non overlapping) operations. In addition, contrary to sequential consistency, it has a compositional property (a combination of separate linearizable implementations of objects is linearizable).

When dealing with replicated data, the above requirements are transposed as follows. The requirement corresponding to sequential consistency is *one-copy serializability* (1SR). This concept, introduced in database systems [Bernstein et al. 1987], captures two notions: (i) replication is invisible to the clients, thus maintaining the illusion that there exists a single (virtual) copy of a replicated object; and (ii) the operations performed on these virtual copies (actually implemented by operations on replicas) are sequentially consistent. Linearizability is likewise extended to replicated objects (in addition to one-copy serializability, the order in which the operations appear to be executed is compatible with the global ordering of operations).

The above criteria define *strong consistency*, in the sense that they maintain the illusion of a single object.

In *weak consistency*, by contrast, individual replicas are considered, and they may diverge, with some specified restrictions. Conflicts appear if these restrictions run the risk of been violated. Conflicts must be resolved (e.g., by preventing or by delaying an operation, etc.). The usual weak consistency condition is *eventual convergence*, also called *uniformity*: if no operations are performed after a certain time, all replicas must eventually converge to be identical.

Consider a service implemented by a set of replicated servers, following either of the patterns described in 11.3.1. Then a sufficient condition for linearizability is that (i) if a request is delivered to one correct replica, it must be delivered to all correct replicas; and (ii) if two requests are delivered to two correct replicas, they must be executed in the same order on both. Two schemes satisfying this condition are described in the next two subsections.

11.6.2 Primary-Backup Server Protocol

In the primary-backup scheme, all requests are sent to the primary, whose identity is known to the clients. In normal operation (i.e., if the primary is up), a request is processed as follows (Figure 11.8):

1. The primary executes the request, yielding a reply r and a new state S.

2. The primary multicasts r and S to all the backups. Each correct backup changes its state to S, stores r, and replies to the primary with an ACK message.

3. The primary waits for all the ACKs issued by the backups that it knows to be correct. It then sends the reply to the client, and waits for the next request.

Each request, and the corresponding reply, is uniquely identified by the client's identity and a unique sequence number for that client.

This scheme satisfies the linearizability condition, since a total ordering of the requests is imposed by the primary, and this order is followed by the backups[14].

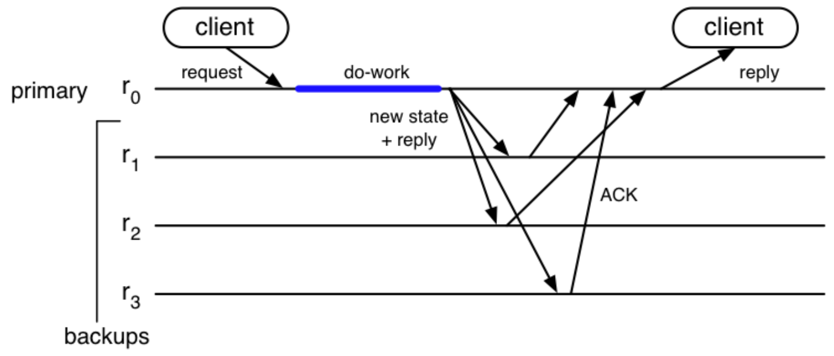


Figure 11.8. Primary-backup replication (adapted from [Guerraoui and Schiper 1997]

We now examine the case of failure. If a backup fails, there is nothing to be done, except that the primary needs to be informed of the failure (this is imposed by step 3 above: the primary does not wait for an ACK from a failed backup).

If the primary fails, a new primary must be selected among the backups, and its identity should be make known to both the clients and the remaining backups. The simplest arrangement is to specify a fixed (circular) ordering among all replicas. The new primary is the first correct replica that follows the failed primary in this ordering, which is known to both the servers and the clients. A client detects the failure of the primary by means of a timeout.

The key point of the recovery protocol is to ensure that the new primary can start operating with no request loss or duplication. The action to perform depends on the precise stage at which the failure occurred.

1. If the primary failed before multicasting the reply and new state of a request, no backup is aware of the request. Eventually, the client will time out, and resend the request to the new primary.

2. If the primary failed after the multicast, but before sending the reply to the client, the new primary will have updated its state; after the view change, it will be aware of its new role and it will send the stored reply to the client.

3. If the primary failed after having sent the reply, the client will receive a duplicated reply from the new primary. It will detect the duplication through the unique identification, and ignore the redundant reply.

[14]The backups do not actually *execute* the requests, but their state evolves as if they did.

A virtually synchronous group membership protocol answers all the demands of the above protocol. The primary uses reliable, view synchronous multicast to send the request and new state to the backups. This guarantees that there is no intermediate situation between the cases 1 and 2 of the recovery protocol (i.e., all the correct backups get the message, or none). A view change is triggered by the failure of a replica (including the primary). Thus the primary knows which backups it should expect an ACK from, while each backup knows the new primary after a failure of the current one.

When a failed replica is reinserted after repair, it triggers a view change and takes the role of a backup. Before it can actually act in that role (i.e., be able to become a primary), it needs to update its state, by querying another correct backup. In the meantime, any incoming messages from the primary should be batched for further processing.

The primary-backup replication scheme resists to $n - 1$ failures if there are n replicas (primary included). If $n = 2$ (single backup), a frequent case, the protocol is simplified: there is no need to wait for an ACK from the backup, and the primary sends the reply to both the client and the backup as soon as the work is done. This case was described in [Alsberg and Day 1976].

An early application of primary-backup replication is the process pair mechanism [Bartlett 1981] used in the Tandem NonStop system, a highly available computer system [Bartlett and Spainhower 2004]. A (logical) process is implemented by a pair of processes (the primary and the backup), running on different processors. A request directed to the logical process goes through a redirection table, which sends it to the process that is the current primary. After completion of a request, the primary commands the backup to checkpoint the request and the process state. When the primary (or its processor) fails, the backup process takes over, and the redirection table is updated. When the failed primary recovers, it becomes the backup.

11.6.3 Active Replication

Active server replication follows the replicated state machine approach described in 11.3.1. A client sends its request to all the replicas, which have the same status. Each (correct) replica does the required work and sends the reply to the client. The client accepts the first reply that it gets, and discards the others, which are redundant. The client is not aware of any failures as long as at least one replica is up; thus, like primary-backup, active replication with n servers resists to $n - 1$ failures.

As noted in 11.3.1, the requests need to be processed in the same order on all replicas, if the execution of a request modifies the state of the server. Therefore the client needs to use totally ordered (atomic) multicast to send its requests.

When a failed replica is reinserted after repair, it needs to update its state. To do so, it atomically multicasts a query for state to the group of replicas (including itself), and uses the first reply to restore its state. Atomic multicast ensures total order between requests for state and client requests. Thus all queried replicas reply with the same value of the state, and the reinserted replica, R, does not lose or duplicate requests, by using the following procedure. Let t be the instant at which R receives its own query for state. Client requests received by R before t are discarded; requests received between t and the first receipt of the state are batched for further processing.

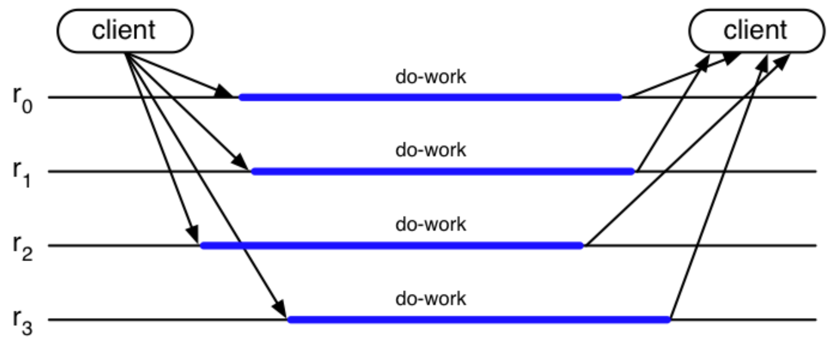


Figure 11.9. Active replication (adapted from [Guerraoui and Schiper 1997]

We now compare the primary-backup and active replication approaches. The main points to note are the following.

- Active replication consumes more resources, since the servers are dedicated to the processing of the requests. With the primary-backup scheme, backups may be used for low-priority jobs.
- Active replication has no additional latency in the case of failures, since there is no recovery protocol. In contrast, the view synchronous failure detection mechanism used in the primary-backup scheme may suffer from false failure detections, which add latency.
- Active replication is transparent for the clients, as long as one server is up. In contrast, clients need to detect the failure of the primary.
- Active replication, being based on the replicated state machine model, implies that the replicated servers behave in a deterministic way. There is no such constraint for the primary-backup scheme.
- Primary-backup uses view-synchronous group membership, while active replication uses totally ordered broadcast. In an asynchronous system with fail-stop failures, both communication protocols are equivalent to consensus. Thus the algorithmic difficulty is the same for both replication schemes.

The standard server replication scheme is primary-backup. Active replication is used in critical environments, when low latency and minimal client involvement are required.

11.6.4 Byzantine Fault Tolerance for Replicated Services

In the Byzantine failure mode, the failed component may have an arbitrary behavior. There are two main reasons to consider Byzantine failures: theoretical (this is the most general failure mode); and practical (the fail-stop hypothesis is not always verified; Byzantine failures cover the case of malicious attacks; and some critical applications require the highest possible degree of fault tolerance).

Early research on Byzantine failures has considered the problem of reliable broadcast. The main results are the following (f is the maximum number of faulty processes).

- With *synchronous* communication [Lamport et al. 1982], reliable broadcast is possible if the number of processes is at least $3f + 1$. The algorithm requires $f + 1$ rounds, and its cost (in terms of number of messages) is exponential in the number of processes. More generally, consensus can be achieved with the same degree of redundancy.

- With *asynchronous* communication [Bracha and Toueg 1985], a weak form of reliable broadcast can be achieved if the number of processes is at least $3f+1$: if the sender is correct, all correct processes deliver the value that was sent; if the sender is faulty, either all correct processes deliver the same (unspecified) value, or the algorithm does not terminate, and no correct process delivers any value.

In both cases the minimum redundancy degree is $3f + 1$. In synchronous systems, this factor may be reduced to $2f + 1$ if messages can be authentified.

For a long period, research on Byzantine failures did not have much practical impact, since the algorithms were considered too expensive. The situation has changed in recent years, and several practical fault tolerance protocols dealing with Byzantine failures have been proposed. Here is a brief summary of these efforts.

[Castro and Liskov 2002] introduced Practical Byzantine Fault Tolerance (PBFT), a form of state machine replication (11.3.1) using an extension of the Paxos consensus protocol to achieve agreement on a total order for requests among all non-faulty replicas. In order to tolerate f failures, the protocol uses $3f + 1$ replicas. We give an outline of the protocol below; see the references fo a detailed description.

Like in classic Paxos (11.3.3), the protocol goes through a sequence of views. In each view, a single primary starts an agreement protocol; if the primary fails, a new view is created, with a new primary. Safety is achieved by the agreement protocol, while liveness depends on view change. All messages are authentified to prevent corruption, replay and spoofing (which could be possible under the Byzantine fault assumption).

The agreement protocol has three phases: pre-prepare, prepare, and commit. The role of the first two phases is to totally order the requests within a view. The role of the last two phases is to ensure that requests that commit (i.e., are executed and provide a result) are totally ordered across views.

In the pre-prepare phase, the primary proposes a sequence number n for a request and multicasts it to the backups in a PRE-PREPARE message. If a backup accepts this message (based on cryptographic check and uniqueness of n within the current view), it enters the prepare phase by multicasting a PREPARE message, still including n, to all replicas (including the primary). If a replica has received $2f$ PREPAREs matching the PRE-PREPARE for a request r from different backups, it marks the request as *prepared*. The following predicate is true: no two non-faulty replicas can have *prepared* requests with the same n and different contents[15] (meaning different encrypted digests).

[15]The proof of this property goes as follows: from the definition of *prepared*, at least $f + 1$ non faulty replicas must have sent a pre-prepare or prepare message for request r. If two correct replicas have prepared requests with different contents, at least one of these senders has sent two conflicting prepares or pre-prepares; but this is not possible, since these replicas are non faulty.

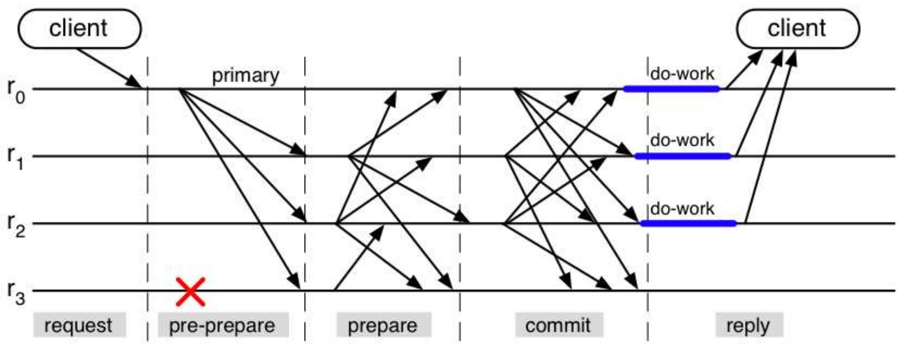


Figure 11.10. The PBFT protocol, normal case (adapted from [Castro and Liskov 1999])

When it has a *prepared* request, a replica multicasts a COMMIT message to the other replicas, thus starting the commit phase. A replica accepts this message if it is properly signed and the view in the message matches the view known as current by the replica. The commit phase ensures that the request is prepared at $f+1$ or more replicas. The key point here is that this condition can be checked by a *local* test at some replica[16] (see the original paper for a proof of this property).

After it has committed a request, a replica executes the work specified in the request and sends the result to the client. The client waits for $f+1$ matching replies. Since at most f replicas can be faulty, the result is correct.

A view change protocol detects the failure of the primary (by a timeout mechanism, which implies partial synchrony), and chooses a new primary. The new primary determines the status of pending requests and resumes normal operation.

This work has shown that Byzantine fault tolerance can be achieved at acceptable cost, and has stimulated further research with the goal of improving the performance of the protocol. Two main paths have been followed: improving throughput, by increasing the number of replicas and using quorums [Cowling et al. 2006, Abd-El-Malek et al. 2005]; improving latency, by reducing the number of phases through speculation (optimistic execution) [Kotla et al. 2007]. A generic, modular approach, which aims at combining the advantages of previous protocols, is presented in [Guerraoui et al. 2008].

11.7 Data Replication

Data replication is motivated by two concerns.

- *Availability.* Maintaining several copies of data on different systems increases the probability of having at least one available copy in the face of processor, system, or network failures.

- *Performance.* Replication allows parallel access to the systems hosting the replicas, thus increasing the global throughput for data access. Distributing replicas over a

[16]The local test verifies the following predicate: the replica has a *prepared* request, and has accepted $2f+1$ matching commits (possibly including its own) from different replicas.

network allows a client to access a replica close to its location, thus reducing latency. Both factors favor scalability.

In this section, we use the term *object* to designate the unit of replication, as chosen by the designer of the replication system. An object may be defined at the physical level (e.g., a disk block), or at the logical level (e.g., a file, or an item in a database). Unless otherwise specified, we assume that replication is based on logical objects.

The counterpart (and main challenge) of data replication is that the replicas of an object need to be kept *consistent* (as discussed in 11.6.1). Maintaining consistency has an impact on both performance and availability. Designing a data replication system thus involves a trade-off between these three properties.

The issue of data replication has been considered in two different contexts: distributed systems and databases. The main motivation of replication is availability for distributed systems, and performance for databases. The two communities have traditionally taken different approaches: database tend to closely integrate replication with data access protocols, while distributed systems use generic tools such as group communication. In recent years, efforts have been made towards a unified approach to data replication. [Wiesmann et al. 2000b] analyze the solutions developed in the two communities and compare them using a common abstract framework. This trend towards convergence has led to the emergence of a middleware-based approach to database replication, in which replication is controlled by a middleware layer located between the clients and the database replicas, and separate from the data access protocols. Separating these two concerns simplifies development and maintenance, and allows using techniques developed for distributed systems, such as group communication protocols. [Cecchet et al. 2008] examine the state of the art and the main challenges of middleware-based database replication.

In an often cited paper, [Gray et al. 1996] discuss the trade-off between consistency and performance in replicated databases. They distinguish between two approaches:

- *Eager replication*, also called *pessimistic* approach: keeping all replicas synchronized, by updating the replicas inside one atomic transaction (i.e., no access is permitted until all replicas are updated).
- *Lazy replication*, also called *optimistic* approach: performing update at one replica, and thereafter asynchronously propagating the changes to the other replicas.

Hybrid methods, combining eager and lazy replication, have also been proposed.

Eager replication maintains strong consistency (in the sense defined in 11.6.1), but it delays data access, and does not scale. In practice, its use is limited to sites located on a local area network, under moderate load. Lazy replication scales well, and may be used with unreliable communication networks. Lazy replication only ensures weak consistency, in the sense that a read may return an out of date value; in addition, an update operation runs the risk of being aborted in order to preserve consistency.

In both eager and lazy replication, a read operation may usually be performed at any site holding a replica. Regarding updates, a distinction may be made between two approaches, which correspond to the two patterns identified in 11.3.1:

- *Update primary* (*or single-master*). For each object, one replica is designated as the primary copy, or master. All updates are made on the master, which is responsible for propagating them to the other replicas.
- *Update anywhere* (*or multi-master*). An update operation may originate at any replica. Potential conflicts must be resolved by an agreement protocol between the involved replicas.

This distinction is orthogonal to that between eager and lazy replication; thus both eager and lazy replication may be implemented using either "update primary" or "update anywhere".

In this section, we present an overview of data replication techniques, illustrated by examples from both databases and distributed systems. Section 11.7.1 deals with strongly consistent replication. Section 11.7.2 examines weakly consistent replication. Section 11.7.3 presents a brief conclusion on data replication.

11.7.1 Strongly Consistent (Eager) Replication

Eager replication preserves a strong form of consistency, usually one-copy serializability, as defined in 11.6.1. Another consistency criterion (snapshot isolation), weaker than 1SR, is discussed later in this section.

[Wiesmann et al. 2000a] propose a classification of eager replication techniques for databases using transactions to access data. In addition to the distinction between primary copy and update everywhere, they identify two other criteria: (i) linear or constant interaction, according to whether the updates are propagated operation by operation, or in a grouped action at the end of a transaction; and (ii) voting or non voting, according to whether the different replicas execute or not a coordination phase at the end of a transaction. They conclude that linear interaction suffers from excessive overhead (due to the large number of update propagation messages), and that constant interaction should be preferred. Voting is needed if the order in which operations are performed on the replicas is non-deterministic.

We now examine the main techniques used to ensure eager replication.

Eager Update Anywhere

The basic protocol for eager replication is *read one-write all* (ROWA). An update operation on an object is performed on all the nodes holding a replica of the object; a read operation is only done on the local copy (or on the closest node holding a replica).

The drawback of this technique is that a failure of a node blocks update operations on the objects that have a replica on that node. Two solutions have been proposed to avoid this problem.

- Update only the available replicas, i.e., those located on non-failed, accessible nodes (*read one-write all available*, or ROWAA [Bernstein et al. 1987]). When a failed node recovers, it needs to update the replicas that it holds before being able to process requests.

- Use a quorum-based protocol [Gifford 1979, Thomas 1979], which requires that a minimum number of sites (a read or write *quorum*) be available for both read and write operations. The basic protocol requires that $r + w > N$, where r is the read quorum, w is the write quorum, and N the number of replicas. This guarantees that a read quorum and a write quorum always intersect; therefore, if the most recently updated copy is selected for a read, it is guaranteed to be up to date. The additional condition $w > N/2$ only allows an update if a majority of sites are available; this prevents divergence of replicas in case of a network partition. The protocol may be refined by assigning different weights to the replicas (N is now the sum of the weights).

If transactions are supported, a 2PC protocol (9.4) is needed if transactions contain writes, and 1SR is ensured by a concurrency control algorithm.

[Jiménez-Peris et al. 2003] compare ROWAA and quorum-based protocols. They conclude that ROWAA is the best solution in most practical situations, specially in terms of availability. Quorums may be of interest in the case of extreme loads.

For both methods, scalability is limited (a few tens of nodes), specially if the update rate is high. For database systems using locks to ensure concurrency control, a strong limiting factor for scalability if the probability of deadlocks, which may in some cases grow as N^3, N being the number of replicas [Gray et al. 1996].

Several approaches, often used in combination, have been proposed to overcome the limitations of ROWAA.

- Using group communication protocols to propagate update to replicas, with order guarantees, in order to reduce the probability of conflicts, and to allow increased concurrency.

- Replacing the 1SR condition by a weaker one, *snapshot isolation* (SI), see 9.2.3. Within a transaction, the first write creates a new version of the updated object[17]. Subsequent reads and writes access this version, while reads not preceded by a write return the last committed version of the object being read. Concurrent writes on the same object by two transactions are not allowed (one of the transactions must abort). Thus reads are neither blocking nor being blocked, and are therefore decoupled from writes, which increases concurrency.

- Taking advantage of cluster architectures.

We illustrate these approaches with a brief review of three examples.

1) Database State Machine. The database state machine [Pedone et al. 2003] is the transposition of the replicated state machine model (11.3.1) to databases. It uses atomic multicast to order updates. The entire database is replicated on several sites. A transaction is processed at a single site (e.g., the nearest). The identity of the objects that are read and written by the transaction (*readset* and *writeset*, respectively) is collected, as well as the values of the updates. When the transaction issues a commit command, it is immediately

[17]This is similar to the copy-on-write technique used in shared virtual memory systems.

committed if it is read only. If the transaction contains writes, the writesets, readsets and updates are atomically multicast to all sites holding a replica of the database. When a site receives this information for a transaction T, it certifies T against all transactions that conflict with T (as known to the the site). All sites execute the same certification protocol (based on concurrency control rules) and get the transaction data in the same order. Therefore, all sites will make the same decision for T (commit or abort). If the transaction commits, its updates are performed at all sites (this is called the deferred update technique). The certification test described in [Pedone et al. 2003] guarantees 1SR. The database state machine approach avoids distributed locking, which improves scalability.

The database state machine is an abstract model, which has only been evaluated by simulation. It may be considered as a basis for the development of actual protocols (for instance, the Postgres-R(SI) protocol described below is based on the same principles, with a relaxed consistency condition).

2) Postgres-R(SI): Group Communication and Snapshot Isolation. Postgres-R(SI) [Wu and Kemme 2005] is a system implementing SI, and based on group communication. A transaction can be submitted at any replica. It executes on that replica, and collects all its write operations in a writeset. When the transaction commits, it multicasts its writesets to all replicas, using uniform atomic (totally ordered) multicast. All replicas execute the operations described in the writesets they receive, in the order the writesets were delivered, which is the same for all replicas. This protocol uses less messages than ROWAA, and allows for increased concurrency, since a transaction may commit locally, without running a 2PC protocol. We do not describe the concurrency control algorithm.

Site failures are detected by the group communication protocol. The group continues working with the available sites. When a site restarts after a failure, it must run a recovery protocol to update its state, by applying the writesets that it missed (these are recovered from a log on a correct site). During this recovery process, the writesets generated by current transactions are buffered for further execution.

3) C-JDBC: Cluster-based Replicated Database. Clusters of workstations have been developed as an alternative to high-end servers. In addition to being easily extensible at a moderate cost, they allow a potentially high degree of parallelism, together with high performance communication. One proposal to exploit these features for data replication at the middleware level is Clustered JDBC, or C-JDBC [Cecchet et al. 2004]. C-JDBC is a front-end to one or several totally or partially replicated databases hosted on a cluster and accessible through the Java Database Connectivity (JDBC) API. It presents a single database view (a virtual database) to the client applications.

C-JDBC is a software implementation of the concept of RAIDb (Redundant Array of Inexpensive Databases), which is a counterpart of the existing RAIDs (Redundant Array of Inexpensive Disks). Like RAIDs, RAIDbs come in different levels, corresponding to various degrees of replication and error checking.

C-JDBC is composed of a JDBC driver, which exports the interface visible to the client applications, and a controller, which handles load balancing and fault tolerance, and acts as a proxy between the C-JDBC driver and the database back-ends.

Operations are synchronous, i.e., the controller waits until it has a response from all back-ends involved in an operation before it sends a response to a client. To reduce latency for the client, C-JDBC also implements early response, i.e., the controller returns the response as soon as a preset number of back-ends (e.g., one, or a majority) has executed an operation. In that case, the communication protocol ensures that the operations are executed in the same order at all involved back-ends.

C-JDBC does not use a 2PC protocol. If a node fails during the execution of an operation, it is disabled. After repair, a log-based recovery protocol restores an up to date state.

The controller is a single point of failure in C-JDBC. To improve availability, the controller may be replicated[18]. To ensure mutual consistency, the replicated controllers use totally ordered group communication to synchronize write requests and transaction demarcation commands.

A system inspired by C-JDBC is Ganymed [Plattner and Alonso 2004], which implements snapshot isolation in order to reduce the probability of conflicts. Ganymed uses a primary copy approach, and the controller ensures that updates are done in the same order at the backup replicas.

In conclusion, update anywhere for eager replication may be summarized as follows. ROWAA is a simple protocol for non-transactional operations. If transactions are used, the traditional technique based on ROWAA is hampered by the deadlock risks of distributed locking and by the cost of distributed commitment. Alternative approaches for transactions involve the use of atomic multicast to propagate updates, and a weakening of the one-copy serializability condition.

Eager Primary Copy

In the primary copy replication technique, one particular site holding a replica of an object is designated as the primary for that object (note that different subsets of data may have different primary sites); the other sites holding replicas are backups. A transaction which updates an object is executed at the primary for that object. A the end of the transaction, the updates are propagated to the backups in a single operation, which groups the changes in FIFO order. If the technique is non-voting, the primary commits the transaction. If it uses voting (in practice, applying a 2PC protocol), the primary and the backups must wait till the conclusion of the vote. Some remarks are in order.

- If there are several primaries, the situation is close to that of an update anywhere technique[19] described above. In the non-voting scheme, the communication protocol must guarantee total order for the updates at the backup sites. In a voting scheme, no order guarantee is needed, but the transaction runs the risk of being aborted; a weak order guarantee (such as FIFO) reduces the risk of abort.

[18]This is called "horizontal" replication, in contrast with "vertical" replication, which consists in building a tree of controllers to support a large number of back-ends.

[19]For that reason, some primary copy systems disallow transactions that update objects with different primary sites.

- In a non-voting protocol, a read operation is only guaranteed to deliver an up to date result if it is performed at the primary (or if the primary does not commit before all backups have been updated). Otherwise, the situation is the same as in lazy replication (11.7.2).

- If a primary fails, the recovery process is different for voting and non-voting techniques. With the voting technique, a backup is always ready to become primary (this is called hot standby). With the non-voting technique, the backup may have pending updates, and needs to install them before becoming primary (this is called cold standby).

In conclusion, primary copy for eager replication is essentially used in two modes: (i) backup copies are used for reads, in which case a voting protocol must be used; or (ii) backups are used for recovery only, in which case all operations are done on the primary.

An early example of this latter use is Tandem's Remote Data Facility [Lyon 1990], in which there is a single backup, and updates are immediately propagated, thus ensuring hot standby.

11.7.2 Loosely Consistent (Lazy) Replication

Lazy replication trades consistency for efficiency; fault tolerance is less of a priority than in eager replication. After recovery from a failure, some updates may have been lost. Lazy replication may be implemented using the primary copy or the update anywhere techniques. In both cases, the main consistency criterion is eventual convergence (as defined in 11.6.1).

Lazy Primary Copy

With the lazy primary copy (or single master) technique, all updates are done on a designated site, the primary[20]. All operations commit immediately. Updates are propagated asynchronously to the backup sites, using either a push or a pull method. Read operations may be done either on the primary or on the backups; in the latter case, an out of date value may be returned, which is acceptable in many applications. Some systems allow a given degree of freshness to be specified for reads, e.g., in terms of maximum age.

Since all updates are done on a single site, potential conflicts between concurrent updates are easily detected and may be avoided by delaying or aborting some operations.

Because of its simplicity, lazy primary copy is the most commonly used technique in commercial database systems. Its main drawback is that the primary site is a single point of failure. If the primary fails, a backup is selected as the new primary, but some updates may have been lost.

One technique that attempts to avoid the loss of updates is to keep a log of updates in stable (failure-resistant) storage. This technique is used by the Slony-I system [Slony-I], which supports several back-ups, possibly organized as a cascade. This technique attempts to approximate hot standby, while keeping the benefits of primary copy update.

[20]In some systems, different objects may have different primary sites.

Lazy Update Anywhere

Lazy update anywhere (or multi-master) techniques are intended to be more efficient than those based on primary copy, by increasing concurrency. The counterpart is that conflicts may occur between concurrent updates at different sites; such conflicts must be detected and resolved. For these systems, fault tolerance is not the primary concern.

[Saito and Shapiro 2005] is an extensive survey of lazy (optimistic) update anywhere replication approaches, mostly for non-database application. They distinguish between state-transfer and operation-transfer systems. In the former, the replicas of an object are updated by copying their contents. In the latter, the operations are propagated to the replicas and executed there.

For state-transfer systems, Thomas's write rule [Thomas 1979] ensures uniformity. Recall (9.2.3) that this rule is based on timestamps; a replica of an object is updated when it detects (e.g., by periodic inspection) a peer holding a more recent copy of the object. Conflicts between concurrent updates are resolved by the "last writer wins" policy.

For operation transfer systems, the situation is analogous to that described in 11.7.1: a sufficient condition for consistency is that the updates be applied at each replica in the same order. This order may be imposed by the sender, through atomic broadcast, or determined by the receivers, using time-stamp vectors. As above, one can use the semantics of the application to loosen the constraint on the order of updates (for example, non-interfering updates may be applied in any order).

11.7.3 Conclusion on Data Replication

Several issues need to be considered for data replication: implemented by a middleware layer or integrated in the application, eager vs lazy update, what consistency guarantees?

Eager replication, which gives strong consistency guarantees, is gaining favor as new designs allow improved performance, specially on clusters. Lazy replication is mandatory in systems that are highly dynamic or have network reliability or connectivity problems, such as MANETs (mobile ad-hoc networks). Such systems also rely on probabilistic techniques (gossip protocols), such as described in 11.4.4.

In practice, essentially for performance reasons, many commercial database systems use primary-based, lazy update replication, integrated within the database kernel. Availability relies on hot standby solutions.

Solutions based on middleware-based systems are investigated. Current efforts are based on group protocols, and tend to favor the eager, update anywhere approach, with snapshot isolation. There is still a gap between current research and actual practice (see [Cecchet et al. 2008]).

11.8 Partial Availability

This section still unavailable.

11.9 Case Studies

This section still unavailable.

11.10 Historical Note

Fault tolerance has always been a primary concern in computing systems. In the early days, attention was mainly focused on unreliable hardware. Error detecting and correcting codes were used to deal with unreliable memory and communication links. Backup techniques (saving drum or disk contents on magnetic tapes) were used for data preservation, with the risk of data loss in the intervals between backups. Later on, disk mirroring techniques improved the situation, but their use was limited by their high relative cost. To deal with processor failures, Triple Modular Redundancy was introduced for critical applications. Software reliability only started receiving attention by the end of the 1960s.

The 1970s are a period of fast progress for computing systems availability. The first *International Symposium on Fault Tolerant Computing* (FTCS) takes place in 1971; it is still mostly devoted to hardware aspects, but software based techniques are beginning to develop. An early attempt at a purely software approach to fault tolerance is the concept of recovery block [Randell 1975], based on redundant programming of critical routines. The design of the SIFT aircraft control system [Wensley et al. 1976] is a large scale effort, mainly using software methods, to build a reliable critical application.

The Tandem highly available systems (1976) mark a breakthrough. While previous fault tolerance techniques were essentially application specific, the Tandem NonStop system [Bartlett 1981] introduces a generic approach combining hardware and software replication (most notably, process pairs). However, process checkpointing is a source of overhead. The Stratus systems (1980) eliminates overhead by a purely hardware approach, at the expense of quadruple redundancy (2 duplexed pairs of processors, the processors of each pair working in lockstep and performing a continuous consistency check).

One of the first available server systems based on the primary-backup approach (with a single backup) is described in [Alsberg and Day 1976]. The replicated state machine paradigm is introduced in [Lamport 1978a], and further refined in [Schneider 1990]. The atomic commitment problem, and its first solution (2PC), are presented in [Gray 1978]. One of the first impossibility results (widely known as “the Generals’ paradox” from [Gray 1978]), related to unreliable communication, is published in [Akkoyunlu et al. 1975]. An influential paper on data availability is [Lampson and Sturgis 1979], which introduces the notion of stable storage. Algorithms for database replication are studied, and the concepts of quorum and majority voting are introduced [Gifford 1979, Thomas 1979].

The 1980’s and early 90’s are marked by a sustained effort to ground the design and construction of fault tolerant systems on a sound fundamental base. Computing systems become distributed; the first *Symposium on Reliable Distributed Systems* (SRDS) takes place in 1981. As a consequence, communication aspects take an increasing importance. Group communication is identified as an essential ingredient of fault tolerant distributed computing, and the central role of agreement protocols (consensus and atomic commitment) is recognized:

- Paxos [Lamport 1998], an efficient consensus algorithm, is published in 1989, but its significance will only be fully recognized much later.
- Unreliable failure detectors are introduced in [Chandra and Toueg 1991]; this paper also gives a consensus-based implementation of atomic broadcast.
- The atomic commitment problem is further explored; a non-blocking solution is proposed in [Skeen 1981]; further advances are described in [Babaoğlu and Toueg 1993]. Advances in fault-tolerant database systems are described in [Bernstein et al. 1987].

The notion of virtual synchrony, which associates group membership and communication, appears in [Birman and Joseph 1987].

Byzantine agreement (a problem that was already identified during the design of the SIFT system) is explored, and its impossibility for $3f$ processes with f faults is proven [Lamport et al. 1982]. A number of other impossibility results, including FLP [Fischer et al. 1985] are discovered; a review of these results is in [Lynch 1989].

The first mention of gossip-based communication appears in [Demers et al. 1987], in a replicated database context, but the idea will only be put to wider practical use a decade later.

The main advance in hardware-based availability in this period is the invention of the RAIDs [Patterson et al. 1988]. As hardware becomes more reliable, the increasing role of software and administration faults is recognized [Gray 1986].

[Cristian 1991] describes the state of the art in fault-tolerant distributed systems at the end of this period.

The mid and late 1990's see the development of distributed systems: cluster computing, distributed transactions, database replication, fault-tolerant middleware. Group communication is now well understood ([Hadzilacos and Toueg 1993] present a survey of this area, which clarifies the concepts and terminology), and used in practical systems. The interplay between theory and practice in the areas of group communication and replication, and its evolution over time, is analyzed in [Schiper 2003, Schiper 2006b].

The 2000's are dominated by two main trends: the rise of Internet computing and services, and the increasing use of mobile devices and ad hoc networking.

An analysis of Internet services failures [Oppenheimer et al. 2003] confirms the trends identified in [Gray 1986]: it shows the decreasing importance of hardware faults, and the dominating role of management related failure causes, such as wrong configuration. A consequence of large scale is the fact that all elements of a system cannot be expect to be fully operational at any point in time. Partial availability is bound to be the rule. The importance of fast failure detection and recovery is emphasized, as illustrated, for example, by project ROC [ROC 2005].

As the size and dynamism of computing systems increases, there is a need for new communication paradigms. Probabilistic methods, such as gossip based broadcast, appear as a promising path in large scale systems communication (see [Kermarrec and van Steen 2007a]).

Concerning the interplay between theory and practice, the treatment of Byzantine failures receives an increasing attention. Byzantine failures are shown to be tractable in practice with an acceptable cost, as illustrated by the experiments described in 11.6.4

Chapter 12

Resource Management and Quality of Service

In a general sense, Quality of Service (QoS) is defined as a set of quality requirements (i.e., desirable properties) of an application, which are not explicitly formulated in its functional interfaces (1.4.1). In that sense, QoS includes fault tolerance, security, performance, and a set of "ilities" such as availability, maintainability, etc. In a more restricted meaning (considered in this chapter), QoS characterizes the ability of an application to satisfy performance-related constraints. This specific meaning of QoS is specially relevant in areas such as multimedia processing, real-time control, or interactive services for end users.

Performance control is achieved through resource management, the main theme of this chapter. We examine the main abstractions and patterns for managing resources, including the use of feedback control methods.

12.1 Introducing Resource Management

The function of a computing system is to provide *services* to its users. Each service is specified by a contract between a service provider and a service requester. This contract defines both the functional interface of the service and some extra-functional aspects, collectively known as Quality of Service (QoS), which include performance, availability, security, and need to be accurately specified for each application or class of applications. The part of the contract that defines QoS is called a *Service Level Agreement* (SLA). The technical expression of an SLA usually consists of a set of *Service Level Objectives* (SLO), each of which defines a precise objective for one of the specific aspects covered by the SLA. For instance, for an SLA on the performance of a web server, an SLO can specify a maximum response time to be achieved for 95% of the requests submitted by a certain class of users.

12.1.1 Motivation and Main Definitions

In order to perform its function, a computing system uses various resources such as processors, memory, communication channels, etc. Managing these resources is an important function, and there are several strong reasons for performing it accurately:

- Maintaining quality of service. Various indicators related to QoS in user applications, specially performance factors, are directly influenced by resource allocation decisions.
- Resource accounting. The users of a shared facility should be charged according to their actual resource consumption. Therefore any consumed resource must be traced back to a user activity.
- Service differentiation and resource pricing. The resource management system may allow users to pay for improved service. The system should guarantee this differentiated form of service, and the pricing scheme should adequately reflect the added value thus acquired.
- Detecting and countering Denial of Service (DoS). A DoS attack aims at preventing useful work from being done, by undue massive acquisition of resources such as CPU time, memory, or network bandwidth.
- Tracking and eliminating performance bugs. Without accurate monitoring of resource usage, a runaway activity might invisibly consume large amounts of resources, leading to performance degradation in user applications; or a regular activity may reserve unnecessary resources, thus hampering the progress of other activities.

In the traditional view of a computing system, resource allocation, i.e., the sharing of a common set of resources between applications contending for their use, was a task performed by the operating system, and user applications had little control over this process. This situation has changed due to the following causes:

- The increasing number of applications subject to strong constraints in time and space, e.g., embedded systems and applications managing multimedia data.
- The growing variability of the environment and operating conditions of many applications, e.g., those involving mobile communications.
- The trend towards more open systems, and the advent of open middleware.

Thus an increasing part of resource management is being delegated to the upper levels, i.e., to the middleware layers and to the applications themselves. In this chapter, we examine some aspects of resource allocation in these upper levels. We start by recalling a few basic definitions (see also 10.1.1).

The term *resource* applies to any identifiable entity (physical or virtual) that is used by a system for service provision. The entity that actually implements service provision, using resources, is called a *resource principal*. Examples of physical resources are processors, memory, disk storage, routers, network links, sensors. Examples of virtual resources are virtual memory, network bandwidth, files and other data (note that virtual resources are abstractions built on top of physical resources). Examples of resource principals are processes in an operating system, groups of processes dedicated to a common task (possibly across several machines), various forms of "agents" (computational entities that may move or spread across the nodes of a network). One may define a hierarchy of principals: for example, a virtual machine provides (virtual) resources to the applications it supports, while requesting (physical or even virtual) resources from a hypervisor.

A general principle that governs resource management is the separation between policies and mechanisms, an instance of separation of concerns (1.4.2). A *policy* defines overall goals for resource allocation, and the algorithms or rules of usage best suited to reach these goals. Such algorithms are implemented using *mechanisms*, which are defined at a low level, with direct access to the individual resources. These mechanisms must be neutral with respect to policies; i.e., several policies may be implemented using the same set of mechanisms, and the design of a mechanism should not preclude its use by different policies. An example of a general policy is to ensure that a given resource (e.g., CPU time) is allocated to a set of processes so that each process gets a share of that resource proportional to a predefined ratio. This policy may be implemented by various mechanisms, e.g., round robin with priorities or lottery scheduling (12.3.2). Conversely, these mechanisms may be used to implement different policies, e.g., including time-varying constraints.

12.1.2 Goals and Policies

The role of resource management is to allocate resources to the service providers (principals), subject to the requirements and constraints of both service providers and resource providers. The objective of a service provider is to respect its SLA, the contract that binds it to service requesters (clients). The objective of a resource provider is to maximize the utilization rate of its resources, and possibly the revenue it draws from their provision. The relationships between clients, service providers, and resource providers are illustrated on Figure 12.1 (as noted before, a resource provider may itself request resources from a higher level provider).

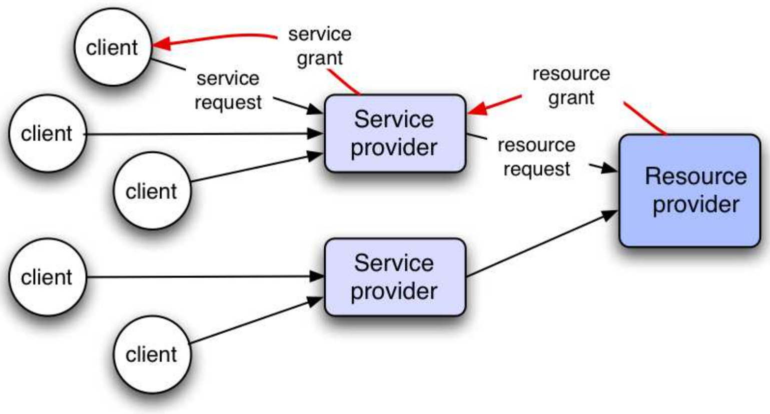


Figure 12.1. Service providers and resource providers

The requirements are the following, as seen by a service provider.

- The service provider should offer QoS guarantees to its clients, as specified by an SLA. The SLA may take various forms: strict guarantee, probabilistic (the agreed levels will be reached with a specified probability, or for a specified fraction of the demands), best effort (no guarantee on the results). The SLA is a contract that goes both ways, i.e., the guarantees are only granted if the clients respects some conditions on their requests. These conditions may apply to each client individually or to a population of clients as a whole.

- The service provider allocates resources for the satisfaction of clients' requests. It should ensure equitable treatment (*fair share*), in the sense that each client needing a resource should be guaranteed a share of that resource proportional to its "right" (or "priority"), as defined by a global policy. The share should be guaranteed in average over a specified period of time. If all clients have the same right, then each should be guaranteed an equal share. Other situations may occur:

  - *Service differentiation* is a means of specifying different classes of clients with different rights, which may be acquired by purchase or by negotiation.
  - The rights may be time-dependent, e.g., the right of a client may be increased to allow it to meet a deadline.

  More generally, the guarantee may be in terms of a minimal rate of progress, which prevents *starvation*, a situation in which a client's requests are indefinitely delayed.

- If several classes of clients are defined, the service provider should guarantee *service isolation*: the allocation of resources for a class should not be influenced by that of other classes. This means that a misbehaving client (one that does not respect its side of the SLA) may only affect other clients of its class, not clients from other classes.

The main requirement of a resource provider is to optimize the usage of the resources, according to various criteria, e.g., ensuring maximal utilization rate over time (possibly with a different weight for each resource), or minimizing energy consumption, etc. This implies that no resource should be idle while there exist unsatisfied requests for that resource (unless this situation is imposed by a higher priority policy, e.g. energy economy). This requirement is subject to the constraint of resource availability: the supply of most resources is bounded. There may be additional constraints on resource usage, e.g., due to administrative reasons, or to locality (e.g., usage and cost conditions are different for a local and a remote resource).

In all cases, a *metric* should be defined to assess the satisfaction of the requirements, both for QoS and for resource utilization factors. Metrics are examined in 12.3.1.

The above requirements may be contradictory. For instance, ensuring guarantees through worst case reservation may entail sub-optimal resource utilization. Such conflicts may only be resolved according to a higher level policy, e.g., through priority setting or through negotiation. Another approach is to pool resources, in order to amortize their cost among several applications, provided that peak loads do not occur at the same time for all of them. Also note that the fairness requirement should be met both at the global level (sharing resources among service providers) and for each service provider with respect to its own service requesters.

Resource management policies may be classified using various criteria (based on prediction and/or observation, using open loop or closed loop) and may face various operating conditions: if resources are globally sufficient to meet the global demand, combinatorial optimization is the relevant tool; if resources are insufficient (the common case), control methods are more appropriate. Examples throughout this chapter illustrate these situations.

Resource allocation is subject to a number of known risks, which any resource management policy should take into account.

- *Violation of fairness*, examples of which are the above-mentioned starvation, and priority inversion, a reversal of prescribed priorities due to unwanted interference between synchronization and priority setting (see 12.3.2).
- *Congestion*, a situation in which the available resources are insufficient to meet the demand. This may be due to an inadequate policy, in which resources are overcommitted, or to a peak in the load. As a result, the system's time is essentially spent in overhead, and no useful work can be done, a situation known as *thrashing*. Thrashing is usually avoided or delayed by an admission control policy (see 12.3.1).
- *Deadlock*, a situation of circular wait, in which a set of processes are blocked, each of them waiting for a resource that is held up by another member of the set. Deadlock may be prevented by avoiding circular dependencies (e.g., through ordered allocation), or detected and resolved, usually at some cost in progress rate.

In the context of this book, we are specifically interested in resource management for middleware systems; among these, Internet services are the subject of an intense activity, due to their economic importance. We conclude this section with a review of the main aspects of resource management for this class of systems.

12.1.3 Resource Management for Internet Services

An increasing number of services are available over the Internet, and are subject to high demand. Internet services include electronic commerce, e-mail, news diffusion, stock trading, and many other applications. As these services provide a growing number of functions to their users, their scale and complexity have also increased. Many services may accept requests from millions of clients. Processing a request submitted by a client typically involves several steps, such as analyzing the request, looking up one or several databases to find relevant information, doing some processing on the results of the queries, dynamically generating a web page to answer the request, and sending this page to the client. This cycle may be shortened, e.g., if a result is available in a cache. To accommodate this interaction pattern, a common form of organization of Internet services is a multi-tier architecture, in which each tier is in charge of a specific phase of request processing.

To answer the demand in computational power and storage space imposed by large scale applications, clusters of commodity, low cost machines have proved an economic alternative to mainframes and high-performance multiprocessors. In addition to flexibility, clusters allow high availability by replicating critical components. Thus each tier of a cluster-based application is deployed on a set of nodes (Figure 12.2). How the application components running on the servers of the different tiers are connected together depends on the architecture of the application; examples may be found in the rest of this chapter. Nodes may be reallocated between different tiers, according to the resource allocation policy.

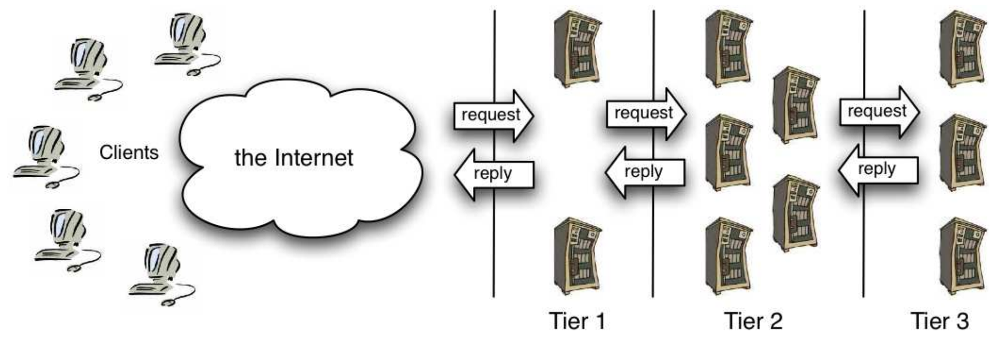


Figure 12.2. A cluster-based multi-tier application

A service provider may deploy its own cluster to support its applications. An alternative solution, in increasing use, is for the provider to host the application on a general purpose platform (or data center) owned by a computing facility provider, and shared with other service providers. The drawback of this solution is that the service provider has less control on the fine-grain tuning of the infrastructure on which its application is running. However, there are several benefits to using shared platforms.

- The service provider is freed from the material tasks of maintaining the infrastructure.
- The fixed cost of ownership (i.e., the part of the cost that is not proportional to the amount of resources) is shared between the users of the common facility.
- Mutualizing a large pool of resources between several applications allows reacting to load peaks by reallocating resources, provided the peaks are not correlated for the different applications.
- Resource sharing improves global availability, because of redundancy in hosts and network connections.

The load imposed on a service is defined by the characteristics of the requests and by their distribution in time. A request is characterized by the resources it consumes at each stage of its processing. Trace analysis of real loads has allowed typical request profiles to be determined for various classes of services (e.g., static content web server, electronic stores, auctions, etc.), which in turn allow building realistic load generators (see examples in [TPC 2006], [Amza et al. 2002]). Predicting the time distribution of the load is much more difficult, since the demand on Internet services is subject to huge, unexpected, variations. In the so-called *flash crowd* effect, the load on a service may experience a 100-fold or 1000-fold increase in a very short time, causing the service to degrade or to crash. Dealing with such overload situations is one of the main challenges of resource allocation.

We conclude this section by a summary of the main issues of resource allocation for Internet services.

Determining the amount of resources of a computing infrastructure is known as the *capacity planning* problem (see e.g., [Menascé and Almeida 2001]). Its solution relies on an estimation of the expected load. However, due to the possibility of wide variations in the load, dimensioning the installation for the maximum expected load leads to over-provisioning. Given that peak loads are exceptional events, a better approach is to define SLAs for a "normal" load (determined after observation, e.g., as the maximum load over 95% of the time). Relating resources to load is done by using a model, such as those described in 12.5. Overload situations are dealt with by admission control, as discussed below.

The problem of *resource provisioning* is that of allocating the resources of an installation in the face of competing demands. These demands may originate from different applications in the case of a shared facility, or from different stages of an application, or from different clients. In all cases, provisioning may be static or dynamic. Static provisioning, also called reservation, is based on an a priori estimate of the needs. Dynamic provisioning varies with time and is adapted to the current demand through a control algorithm.

Admission control (sometimes called *session policing*[1]) is a response to an overload situation, which consists in turning away a fraction of the requests to ensure that the remaining requests meet their SLA. Admission control needs to take into account some measure of the "value" of the requests, which may have different forms: class of service (if service differentiation is supported), estimated resource consumption of the request, etc.

A complementary way of dealing with overload situations is *service degradation*, which consists in providing a lower quality of service to some requests, thus consuming less resources. The measure of quality is application-dependent; examples include lowering image resolution, stripping down a web page by eliminating images, reducing requirements on the freshness or consistency of data, etc.).

In the rest of this chapter, we examine some approaches to solving these problems. We first examine the main abstractions relevant to resource management. We then consider resource management from the point of view of control and we present the main mechanisms and policies used for resource management at the middleware or application level, using both feed-forward and feedback forms of control. This presentation is illustrated by several case studies.

12.2 Abstractions for Resource Management

Resource management involves three main classes of entities: the resources to be allocated, the resource principals (resource consumers), and the resource managers. The first task in developing a resource management framework is to define suitable abstractions for these entities and for their mutual relationships, and to design mechanisms for their implementation.

[1]This term comes from network technology: *traffic policing* means controlling the maximum rate of traffic sent or received on a network interface, in order to preserve network QoS.

12.2.1 Resources

The physical resources of a system are the identifiable items that may be used for the execution of a task, at a certain level of visibility. Usually this level is that of executable programs, for which the resources are CPUs, memory, disk space; communication related resources, such as disk or network I/O, are quantified in terms of available bandwidth.

An operating system creates an abstract view of these resources through virtualization, in order to hide low-level allocation mechanisms from its users. Thus CPU time is abstracted by threads, physical memory by virtual memory, and disk storage by files. These abstractions may be considered as resources when allocated to users through processes; but, from the point of view of the operating system kernel, they are principals for the corresponding physical resources.

The whole set of physical resources that makes up a machine may itself be abstracted in the form of a virtual machine, which provides the same interface as the physical machine to its users, and runs its own operating system. This is again a two-level allocation mechanism: at the lower level, a hypervisor multiplexes the physical resources between the virtual machines; on each virtual machine, the operating system's allocator shares the virtual resources between the users. The interplay between the two levels of resource management is examined in Section 12.2.2. The notion of a virtual machine has been extended to multiprocessors [Govil et al. 2000] and to clusters (as discussed in more detail in Section 12.2.2).

A resource is defined by a number of properties, which influence the way it may be used.

- Exclusive or shared use. Most resources are used in exclusive mode, i.e., they may only be allocated to one principal at a time. Such is the case of a CPU, or a block of private memory. In some infrequent cases, a resource may be shared, i.e., simultaneously used by several principals. This is the case of a shared, read-only memory page or disk file. Even in that case, a maximal degree of sharing may be specified to avoid performance degradation.

- Stateful or stateless. A resource may have a state related to the principal that currently uses it. This state needs to be set up when the resource is allocated; it should be cleaned up, and possibly stored for later reuse, when the resource is reallocated. This operation may be simple and cheap (e.g., in the case of a CPU, in which the state consists of a set of registers), or fairly costly, in the case of a whole computer for which a new operating system may need to be installed.

- Individual or pooled. A resource may exist as a single instance, or it may be part of a pool of identical resources (such that any resource of the pool may be allocated to satisfy a request). Typical pooled resources are memory pages, CPUs in a multiprocessor, or machines in a homogeneous cluster. Note that, if a pooled resource is stateful, there may be a preference to reallocate an instance previously used by the same principal, in order to spare state cleanup and restoration (this is called *affinity scheduling* in the case of CPUs).

  For virtual resources, pooling is often used as an optimization device. Contrasting with physical resources, virtual resources may be created and deleted; these operations have a cost. In order to reduce the overhead of creation and deletion, a pool of resources is initially created. A request is satisfied by taking a resource from the pool; when released, the resource is returned to the pool. Various policies may be used to adapt the size of the pool to the mean demand (if the pool is too small, it will frequently be exhausted; if too large, resources will be wasted). Such pools are frequently used for threads, for network connections, or for components in container-based middleware.

In addition to the above characteristics, a resource may have a number of attributes (e.g., for a printer, its location, its throughput, its color characteristics, etc.) and may allow access to some current load factors (e.g., number of pending requests, etc.). All the properties of a resource are typically grouped in a data structure, the resource descriptor, and are used for resource discovery, the process of looking up for a resource satisfying some constraints (e.g., the closest color printer, or the one with the shortest job queue, etc.). Techniques used for service discovery (3.2.3) are relevant here.

In a large scale system, providing an accurate, up to date, view of the current state of all resources may in itself be a challenging task. Such a global view is seldom needed, however, since most distributed resource management systems aim at satisfying a request with a resource available in a local environment.

12.2.2 Resource Principals

In current operating systems, resources such as virtual memory or CPU time are allocated to processes. A process, in turn, is created on behalf of a user. Thus the resources consumed by a process are charged to the user for which it executes.

This scheme is not adequate for fair and accurate resource management, for several reasons: the time spent in the system kernel, e.g., for servicing interrupts, is not taken into account; a single system process may do work for a number of different users, e.g., for network related activities; conversely, a single user task may be split across several processes. Therefore there is a need for defining a separate notion of a resource principal.

This is yet another application of the principle of separation of concerns (1.4.2). A resource principal is defined as a unit of independent activity devoted to the execution of a well-identified service; all the resources that is uses should also be well-identified. There is no reason why resource principals should coincide with entities defined using different criteria such as unit of protection or unit of CPU scheduling.

Note that resource principals are essentially mechanisms; they may be used to implement any globally defined policy for resource sharing among the independent activities that they represent.

Several attempts have been done towards defining resource principals and relating them to other entities. We briefly review some of these efforts. This subject is still an area of active research.

Resource Containers

On a single machine, *resource containers* have been introduced in [Banga et al. 1999], specifically in the context of networked servers, such as Web or database servers. A resource container is an abstract entity that contains all the system resources used by an application to achieve a particular independent activity (in the case of Web servers, an activity is the servicing of a particular HTTP connection).

Consider the case of CPU time. A resource container is bound to a number of threads, and this binding is dynamic. Thus a given thread may do work for several containers and its binding changes accordingly over time. Each container receives CPU time according to some policy (e.g., fixed share, etc.), and distributes it among the threads that are currently bound to it. Thus a thread that is multiplexed among several containers receives resources, at different times, from all these containers[2].

Other resources are managed in a similar manner. Thus a container may be dynamically bound to a number of sockets and file descriptors, to implement the management of network and disk bandwidth, respectively.

Containers may be organized in a hierarchy to control resource consumption at a finer grain: a container may divide its resources among child containers. Resource allocation within a container, managed at a certain level, is independent of resource allocation between the containers at that level. Containers defined at the top level get their resources from the system, according to a global resource management policy.

In the experiments described in [Banga et al. 1999], resource containers have been used to implement prioritized handling of clients, to control the amount of resources used by CGI processing, and to protect a server against SYN-flooding, a DoS attack that attempts to monopolize the use of the server's network bandwidth.

Cluster Reserves and Related Work

Cluster reserves [Aron et al. 2000] are an extension of resource containers to cluster-based servers, running on a set of commodity workstations connected by a network. Typically, a request for a service is sent to a node designated as the front-end, which in turn assigns the execution of the request to one or several nodes, according to a load sharing policy (12.3.2). Such a policy defines a set of service classes (units of independent activity), together with rules for sharing the cluster's resources among the service classes. An implementation of a global policy should ensure performance isolation, i.e., it should guarantee that these rules are actually enforced (for instance, an application should not "steal" resources from another one, leading to a violation of the global allocation policy). Assigning a request to a service class may be done on various criteria, such as request content, client identity, etc.

Service classes are implemented by cluster reserves. A *cluster reserve* is a cluster-wide resource principal that aggregates resource containers hosted on the nodes of the cluster. A resource globally allocated to a cluster reserve may be dynamically split among a set of resource containers on different nodes.

The problem of partitioning the resources among the individual containers is mapped to a constrained optimization problem. The goal is to compute a resource allocation on each node, while satisfying several constraints: the total allocation of each resource for

[2]If the binding changes frequently, rescheduling the thread at each change may prove costly; thus the scheduling of a shared thread is based on a combined allocation of the set of containers to which it is bound, and this set is periodically recomputed; thus the rescheduling period may be controlled, at the expense of fine grain accuracy.

each reserve should minimally deviate from that specified by the global allocation policy; the total allocation of any resource on each node should not exceed the available amount; no resource should be wasted (i.e., allocated to a container that has no use for it); minimal progress should be guaranteed for each service class. A resource manager node runs a solver program that recomputes the optimal resource allocation (periodically or on demand) and notifies it to the local manager of each node.

Experiments reported in [Aron et al. 2000] show that cluster reserves can achieve better resource usage than a policy that reserves a fixed set of nodes for each service class, and that they provide good performance isolation between a set of service classes.

An extension of the principle of cluster reserves has been developed in the Sharc system [Urgaonkar and Shenoy 2004]. An application running on a cluster is decomposed in individual components, or *capsules*, each running on a single node. Each capsule imposes its own requirements in terms of performance and resource needs (the resources controlled by Sharc are CPU and network bandwidth, while cluster reserves only control CPU). Several application may concurrently share the cluster. A two-level resource manager (see 12.2.3) allocates the resources among the capsules. When a new application is started, the manager determines the placement of its capsules on the nodes, according to available resources. During execution, resources may be traded between capsules on a node (within a single application), depending on their current needs; thus a capsule that has spare resources may temporarily lend them to another capsule. Experience shows that this system effectively shares resources on a moderate size cluster.

Virtual Clusters

Clusters on Demand (COD) [Chase et al. 2003, Moore et al. 2002] is another attempt towards global resource management on clusters and grids. The hardware platform is a (typically large) collection of machines connected by a network. The approach taken here is to consider this platform as a utility, to be shared by different user communities with different needs, possibly running different software. Thus each such community receives a *virtual cluster* (in short, *vcluster*), i.e., an autonomous, isolated partition composed of a collection of nodes. The users of a *vcluster* have total control on it (subject to authorization), and may install a new software environment, down to the operating system.

The nodes that compose a *vcluster* are dynamically allocated. A node is usually a physical machine, but it may also be a virtual machine hosted by a physical node. A *vcluster* may expand or shrink, by acquiring or releasing nodes. This resizing may be at the initiative of the *vcluster*, to react to variations in load, or at the initiative of the system manager, according to the global management policy. Thus a negotiation protocol takes place between these two levels to resolve conflicts (e.g., if several *vclusters* need to expand at the same time).

A node is a stateful resource. When a node is reallocated, some of its state needs to be saved, and it may possibly be entirely reinstalled. A bootstrap mechanism allows an entirely new operating system to be installed at node reallocation.

Vclusters, like other forms of resource principals, are mechanisms. They may be used to implement various policies, both for resource allocation within a *vcluster* (reacting to load variations) and for global cluster management.

Another system based on the same principle is Oceano [Appleby et al. 2001].

12.2.3 Resource Managers

A *resource manager* is the entity responsible for managing resources. Depending on the system structure, it may take the form of a server process or of a passive object. In the first case, the requests are sent as messages; in the second case, they take the forms of procedure or method calls. The call may be implicit, i.e., triggered by an automatic mechanism; such is the case for operating systems resource managers (e.g., a memory manager is called by a page fault hardware trap). A resource manager may also be distributed, i.e., it is composed of several managers that collaborate using a group protocol (peer to peer or master-slave), while providing a single API to its users.

The role of a manager is to allocate resources at a certain level, by multiplexing resources available at a lower level. Thus, a complex system usually relies on a hierarchy of managers. The lowest level managers allocate physical resources such as CPU or memory, and deliver corresponding resources in a virtual form. These virtual resources are in turn allocated by higher level managers. For example, a low-level scheduler allocates CPU to kernel-level threads; these in turn are used as resources by user-level threads. A resource manager is usually associated with a management *domain* (10.1.2), which groups a set of resources subject to a common policy.

These hierarchical schemes may be exploited in various ways, according to the overall organization of resource management. In an organization using an elaborate model for resource principals, a global management level allocates resources between these principals. Then a local manager is associated with each principal (e.g., a virtual cluster) to allocate resources within that principal, which achieves isolation. In an organization based on multi-node containers (as described in Section 12.2.2) the manager hierarchy may correspond to a physical organization: a manager for a resource in a multi-node container relies on sub-managers for that resource on each node. The already mentioned Sharc system [Urgaonkar and Shenoy 2004], for instance, uses this two-level manager scheme.

12.3 Resource Management as a Control Process

Like other aspects of system administration, resource management may be viewed as a control process. The principles of control were introduced in 10.2.1. Recall that control may take two forms: open loop (or feed-forward), and closed loop (or feedback). Both approaches are used for resource management. However, open loop control relies on a prediction of the load and needs an accurate model of the controlled system, two requirements that are difficult to meet in practice. Therefore, most open-loop policies either make strong assumptions on the resource needs of an application, or apply to local decisions for which a policy may be defined a priori.

In 12.3.1, we examine the main issues of resource management algorithms. In 12.3.2, we discuss the main basic open loop policies used to manage resources. In 12.3.3, we present market-based approaches, another form of open loop management relying on a decentralized, implicit, form of global control. Resource management using feedback control is the subject of 12.4.

12.3.1 Resource Management Algorithms

Assume that the managed system is in a state in which its current resource principals meet a prescribed QoS specification using the resources already acquired (call it a “healthy” state). A control algorithm for resource management attempts to keep the system in a healthy state, using the three means of action outlined in 12.1.3: dynamic resource provisioning, admission control, and service degradation. These may be controlled through feed-forward, feedback, or through a mixed approach. In addition, the problem of bringing the system in an initial healthy state should also be solved.

Several common questions arise in the design of a control algorithm. We examine them in turn.

1) System States and Metrics

The first question is how to define a healthy state. In other words, what metric is used to assess the state of the system?

Recall (12.1) that QoS is specified by Service Level Objectives (SLOs), which are the technical expression of an SLA. SLOs involve high-level performance factors, such as global response time or global throughput. While these factors are related to client satisfaction, they cannot be directly used to characterize the state of a system, for which resource allocation indicators are more relevant. These indicators are more easily measured, and can be used for capacity planning and for controlling the system during operation.

The problem of SLA decomposition for performance QoS [Chen et al. 2007] is to derive low-level resource occupation factors from Service Level Objectives. Note that equivalent versions of this problem exist for other aspects of QoS, such as availability and security. These are discussed in Chapters 11 and 13, respectively. Here we only consider the performance aspects of QoS.

To illustrate this issue, consider a 3-tier implementation of an Internet service, for which SLOs are expressed as a maximum mean response time R and a minimum throughput T. For a given system infrastructure, the problem is to map these requirements onto threshold values for resource occupation at the different tiers:

$$(R, T) \mapsto (\eta_{http-cpu}, \eta_{http-mem}, \eta_{app-cpu}, \eta_{app-mem}, \eta_{db-cpu}, \eta_{db-mem})$$

where η_{*-cpu} and η_{*-mem} are the occupation rates of CPU and memory, respectively, for the three tiers: HTTP, Application, and Database.

Two main approaches have been proposed to solve the SLA decomposition problem.

- Using a model of the system to derive low-level resource occupation thresholds from high-level SLOs.
- Using statistical analysis to infer relevant system-related metrics from user-perceived performance factors.

The model-based approach relies on the ability to build a complete and accurate model of the system. This is a difficult task, due to the complexity of real systems, and to the widely varying load conditions. However, progress is being made; the most promising approach seems to be based on queueing network models. [Doyle et al. 2003] uses a simple queueing model to represent a static content Web service. [Chen et al. 2007] use a more elaborate queueing network to describe a multi-tier service with dynamic web content. This model is similar to that of [Urgaonkar et al. 2007], also described in Section 12.5.5.

The statistical analysis approach is fairly independent of specific domain knowledge, and may thus apply to a wide range of systems and operating environments. An example of this approach is [Cohen et al. 2004]. The objective is to correlate system-level metrics and threshold values with high-level performance factors such as expressed in SLOs. To that end, they use Tree-Augmented Naive Bayesian Networks, or TANSs [Friedman et al. 1997], a statistical tool for classification and data correlation. Experiments with a 3-tier e-commerce system have shown that a small number of system-level metrics (3 to 8) can predict SLO violations accurately, and that combinations of such metrics are significantly more predictive than individual metrics (a similar conclusion was derived from the experiments described in 12.5.4). The method is useful for prediction, but its practical use for closed loop control has not been validated.

2) Predictive vs Reactive Algorithms

Are decisions based on prediction or on reaction to real-time measurement through sensors? In the predictive approach, the algorithm tries to assess whether the decision will keep the system in a healthy state. This prediction may be based on estimated upper limits of resource consumption (using a model, as described above), or on the prior observation of a typical class of workload. Both approaches to prediction are useful for estimating mean values and medium-term evolution, but does not help in the case of load peaks. Thus a promising path seems to design algorithms that combine prediction with reaction and thus implement a mixed feed-forward-feedback control scheme.

3) Decision Instants

What are the decision instants? The decisions may be made periodically (with a predefined period), or may be linked to significant events in the system, such as the arrival or the termination of a request, or depending on measurements (e.g., if some load factor exceeds a preset threshold). These approaches are usually combined.

4) Heuristics and Strategies

While the design of a resource management algorithm depends on the specific features of the controlled system and of its load, a few heuristic principles apply to all situations.

- *Allocate resources in proportion of the needs.* As discussed above, the needs may be estimated by various methods. Techniques for proportional allocation are discussed in 12.3.2.

- In the absence of other information, attempt to *equally balance resource occupation.* An example illustrating this principle (load balancing algorithms) is presented in 12.3.2.

- *Shed load to avoid thrashing.* Experience shows that the best way of dealing with a peak load is to keep only a fraction of the load that can be serviced within the SLA, and to reject the rest. This is the main objective of admission control (see 12.5 for detailed examples).

Recall (12.1.3) that three forms of resource management algorithms may be used, in isolation or combined. The base for their decisions is outlined below.

In the case of resource provisioning, the decision is to allocate a resource to a principal (acting on behalf of a request or set of requests), either from an available resource pool, or by taking it from another principal. The decision is guided by the "proportional allocation" principle, possibly complemented by an estimate of the effect of the action on the measured QoS.

In the case of admission control, a request goes through an admission filter. The filtering decision (admit or reject) is based on an estimate of whether admitting the request would keep or not the system in a healthy state. There may be a single filter at the receiving end, or multiple filters, e.g., at the entry of each tier in a multi-tiered system. A rejected request may be kept by the system to be resubmitted latter, or to be granted when admission criteria are met again, e.g., after resources have been released. Alternatively, in an interactive system such as a web server, the request may be discarded and a rejection message may be sent to the requester, with possibly an estimate of a future acceptance time.

In the case of service degradation, the decision is to lower the service for some requests, with the goal of maintaining the system in a healthy state with respect to other requests. There are two aspects to the decision: how to select the "victim" requests, and by what amount to degrade the service.

In all cases, the decision involves an estimate of its impact on the state of the system. As explained in the discussion on system states, this estimate involves correlating resource utilization thresholds with SLOs, through the use of a model and/or statistical analysis.

12.3.2 Basic Open Loop Policies

Open-loop policies for resource management make decisions based on the knowledge of the resource needs, which may either be given a priori, or predicted by one of the techniques discussed above. Some of these policies are a base on which more elaborate resource management systems using feedback have been developed.

In this section, we first briefly present static reservation. We then go on to a discussion of some usual techniques for implementing proportional scheduling: allocating resources in proportion of specified needs. We finally illustrate balanced resource allocation with a brief overview of load balancing.

1) Reservation

One way to ensure that a service provider respects its SLA is to reserve for it all the resources that it may need, assuming the resource needs (or at least an upper bound of the needs for each kind of resource) are known in advance. Thus the resources of the system are statically divided between the different classes of services. Each provider also may use reservation to allocate its resource pool among the service requesters. If a service provider closes its activity, its resources become available, and may be reallocated to a new provider, if they satisfy its needs.

This method assumes that the total available amount of each resource is large enough to satisfy the aggregated demand. In addition, each reservation should be made by considering the worst case situation. Thus, if there is a large variation in resource demand, or if the worst case demand cannot be accurately estimated, a significant amount of resources may be wasted. This limits the applicability of static reservation; however, this drawback may be reduced by temporary trading unused resources, such as proposed in Sharc (12.2.2 and [Urgaonkar and Shenoy 2004]).

The same principle is used for capacity planning, i.e., estimating the global amount of resources of a platform, given its expected load and the SLAs. Capacity planning thus relies on a mapping of SLAs to resource occupation, as discussed above. If worst case estimates are used, overestimating the amount of resources leads to underutilization. To counter this risk, [Urgaonkar et al. 2002] propose a controlled overbooking strategy.

A common use of reservation is to guarantee QoS in network communications, by statically reserving network bandwidth and buffer space.

2) Proportional Scheduling

A frequent situation is one in which the policy to be implemented for sharing a resource is one of *proportional-share*, i.e., each requester gets a share of the resource proportional to a predefined ratio. This ratio may be fixed, or it may vary over time. This may be used at the global level, to allocate resources among various service classes, or within a service provider, to share resources among the requesters of that service.

A more precise statement of the requirements is as follows. Let there be n contenders for a (non-shared) resource, and define a set of ratios $r_1, \ldots r_n$, such that $0 \leq r_i \leq 1$ and $\sum_{i=1}^{n} r_i = 1$. The goal is that, at any time, each contender c_i should be granted a fraction r_i of the resource. If the resource is, for example, CPU time, this means that c_i gets a virtual processor whose speed is r_i times that of the actual CPU. If the resource is memory, c_i gets a fraction r_i of the available memory (possibly rounded to the nearest page limit if the memory is paged). This holds even if the r_i or n vary over time.

Priority-based methods. The problem of proportional scheduling has initially been attacked by assigning priorities to the contenders, in proportion to their assigned ratios. This is the classical method used for CPU scheduling in operating systems. In the simplest case, all processes have equal priority, and are ordered in a single run queue, scheduled by round robin with a fixed quantum q. This approximates the "shared processor" model (the limit case when q goes to 0), in which each process is granted a virtual processor of equal speed (the speed of the physical processor divided by the number of processes). If priorities are introduced, the scheduler orders the run queue by decreasing priority, with FIFO ordering within each priority.

A stronger bias in favor of high priority processes may be achieved by several means.

- Preemption. A new process preempts the processor if its priority is higher than that of the active process.
- Multiple queues. A different queue is used for each priority, with lower quantum values for high priority. The processor is granted to the first process of the non-empty queue with the highest priority, and preemption is applied.

While widely used in commercial systems, priority-based methods suffer from several drawbacks.

- The relative proportion of priorities has no simple relationship with the prescribed rates; thus priorities must be set by trial and error to approximate these rates.
- The system is subject to starvation. This may be prevented by having the priority of a process increase with its waiting time (to be restored to its initial value after it has been served), but this further complexifies the behavior of the system.
- For real-time systems, in which processes must meet deadlines, there is a risk of priority inversion. Consider a process *P0* of priority 0 (the highest), which is blocked by a mutex held by a process *P7* of priority 7. *P7* in turn is delayed by processes of intermediate priorities (4, 5, ...) and thus cannot release the mutex. As a consequence, *P0* may miss a deadline. Ad hoc techniques have been proposed to avoid priority inversion (e.g., increasing the priority of a process while it holds a mutex), but the competition between processes is not always apparent, as the mutex may be hidden under several layers of abstraction in an operating system routine.

Therefore alternative methods have been proposed for achieving proportional rate resource allocation.

Lottery scheduling. A more accurate and efficient solution is provided by *lottery scheduling* [Waldspurger and Weihl 1994], a mechanism that has initially been proposed for proportional-share CPU scheduling. Each contender c_i gets a number of tickets proportional to the ratio r_i; each ticket carries a different value. Each time an allocation should be made (e.g., at periodic instants, or at each new request, etc.), a lottery (i.e., a random drawing in the set of tickets, with equal chances) is held, and the resource is allocated to the holder of the winning ticket.

Lottery scheduling has been extended and adapted to other resources such as memory or network bandwidth, and to the allocation of multiple resources [Sullivan et al. 1999].

It can be shown that the allocation is probabilistically fair, in the sense that the expected allocation to clients is indeed proportional to the number of tickets they hold, i.e., to the assigned ratios. While a discrepancy may exist between the actual proportion of the resource allocated to a client and the expected one, the accuracy improves like $\sqrt{N}$, where N is the number of drawings. The overhead cost of a random drawing is very small.

Several additional mechanisms may be implemented, such as temporary transfer of tickets between clients, e.g., when a client cannot use its share of tickets because it waits for a message or another resource. This method may be used to prevent priority inversion.

3) Load balancing

In the context of cluster computing, *load balancing* is a technique used to distribute a set of tasks among the nodes of a cluster. A typical situation is that of a cluster-based web server which receives requests from clients. A common architecture consists of a front-end node (the switch) and a set of homogeneous back-end (or server) nodes on which the actual work is performed. Client requests are directed to the switch, which distributes them among the server nodes (Figure 12.3(a)). This scheme is also used between the successive tiers of a multi-tier system, in which each tier runs on a set of nodes (Figure 12.3(b)).

From the point of view of a client, the front-end node and the servers may be considered together as a virtual server, which globally provides the specified service. The clients are unaware of the distribution of requests among the servers.

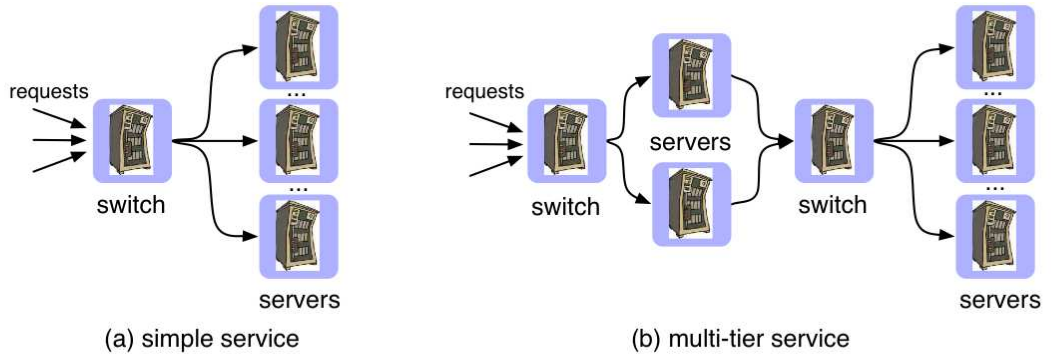


Figure 12.3. Load balancing

The objectives of a load balancing policy are the following.

- Keeping the servers active; no server should be idle while unsatisfied requests are pending. A heuristic to achieve this goal is to distribute the load "evenly" among the servers (in a sense to be illustrated by the examples that follow).
- Increasing the overall performance, e.g., reducing the mean response time of the requests, or maximizing the throughput.
- As an accessory goal, exploiting the existence of multiple servers to increase availability; however, for this goal to be achieved, the switch itself should be replicated to preserve its function in case of failure.

These objectives are subject to a number of constraints, among which the following.

- If the requests are not independent (e.g., if they activate subtasks of a parallelized global task), the precedence constraints between the subtasks should be respected.
- If client-specific state is maintained by the server, the requests from a given client should be run on the same server (this property is called persistence[3]).

[3]This term has a different meaning from that used in Chapter 8.

The simplest load balancing policy is Round-Robin: successive requests are sent to servers in a sequential, circular order (S_1, S_2, ..., S_n, S_1, ...). This policy is static, in the sense that it depends neither on the characteristics of the requests nor on the current load of the servers. Round-Robin gives acceptable results for a "well-behaved" load, in which arrival and service times are regularly distributed (e.g., following an exponential law). However, as pointed out in 12.1.3, real loads usually exhibit a less predictable behavior, with high variability in both arrival rates and service times. For such irregular loads, Round-Robin may cause the load on the servers to be heavily unbalanced. For this reason, dynamic policies have been investigated. Such policies base their decisions on the current load of the back-end servers and/or on the content of the requests.

A first improvement on basic Round-Robin is to take the current server load into account, by assigning each server a weight proportional to its current load (server load estimates are periodically updated). Thus lightly loaded nodes are privileged for accepting new requests. Server load is estimated by CPU or disk utilization, by the number of open connections, or by a combination of these factors. This policy, called Dynamic Weighted Round-Robin, is simple to implement, since it does not involve request analysis. The same principle is applied for balancing the load between heterogeneous nodes with different performance characteristics: the nodes with higher performance are privileged.

In addition to server load, more elaborate policies also take request contents into account. One of the first attempts in this direction was the Locality Aware Request Distribution (LARD) proposed in [Pai et al. 1998]. The main objective is to improve cache hit rates in the servers, thus reducing mean response time. To do so, the name space that defines the data to be fetched from the database (e.g., the URLs) is partitioned, for instance through a hash-coding function, and each partition (or target) is assigned to a particular back-end server, thus improving cache locality. LARD combines load-aware balancing and high locality, by giving priority to lightly loaded servers for the first assignment of a target. In addition, server activity is monitored in order to detect substantial load imbalance, and to correct it by reassigning targets between servers.

More recent work, ADAPTLOAD [Zhang et al. 2005], attempts to improve on LARD by dynamically adjusting the parameters of the load distribution policy, in order to react to peaks in the load. To do so, ADAPTLOAD uses knowledge of the history of request distribution.

A survey of load balancing policies for clustered web servers may be found in [Cardellini et al. 2002].

12.3.3 Market-based Resource Management

The presentation of the goals and policies of resource allocation (12.1.2) suggests an analogy with an economic system in which customers and suppliers are trading goods and services. A number of projects have tried to exploit this analogy (see [Sutherland 1968] for an early attempt), but none of these proposals has actually achieved wide application.

In this section, we summarize the main problems of market-based resource management, and we illustrate them with a case study.

1) Issues in Market-based Resource Management

With the emergence of large distributed systems shared by a community of users (clusters, grids, PlanetLab [Bavier et al. 2004]), the idea of using market mechanisms for resource allocation has been revived in several recent proposals. Drawing on the analogy with an economic system, the goal is to assign resources to requests, while satisfying a set of constraints. These include limitations on both the amount of resources available and on the payment means of the requesters, as well as meeting the goals of both providers and requesters. Since these goals may be contradictory (as pointed out in 12.1.2), each party will attempt to maximize its own utility function, based on a valuation of resources, and a global compromise should be sought.

The following main issues need to be considered.

- How to assign a value to a request? Note that this value may be a function of time, since a request may prove of no value (or may even cause a penalty) if satisfied too late.
- How to assign a value to a resource? This value may again vary, depending on availability and on demand (e.g., through a form of bidding).
- Does there exist an equilibrium point between the requirements of resource providers and requesters?
- How is value measured? Economic analogies suggest bartering (trading a resource for another one), or using a form of currency, real or virtual.
- What mechanisms are used for contracting? Again, analogies suggest the notions of brokering (using intermediate agents), using leases (granting a resource for a fixed period of time), etc.
- How to prevent and detect misbehavior (cheating, stealing, overspending)?

These issues are still being explored, and there is no universally accepted solution. The following example gives an idea of some orientations of current research.

2) Case Study: Currency-based Resource Management

We present this case study in two steps. We first describe Sharp (Secure Highly Available Resource Peering) [Fu et al. 2003], a framework for distributed resource management in an Internet scale computing infrastructure. We then show the use of this framework to acquire resources by means of virtual currency.

Sharp defines abstractions and mechanisms for coordinated resource allocation on a system composed of a collection of sites, where a *site* is a local resource management domain, which may range from a single machine to a cluster. This framework is designed for building resource management systems that are flexible (allowing for various policies and trade-offs), robust (tolerant to partial failures), and secure (resisting to attacks).

Sharp relies on a hierarchy of resource managers, based on two roles: authority and broker. An *authority* is the entity that has actual control over a set of resources and maintains their status. A *broker*[4] is an intermediate entity that has a delegation from an authority and receives requests for the resources managed by that authority; an authority may grant delegation to several brokers.

A *claim* is an assertion stating that some principal has control on some set of resources over some time interval. A claim may be soft, i.e., it is only a promise that may possibly not be fulfilled; or it may be hard, i.e., the resources are actually granted. A soft claim is concretely represented by a *ticket*, while a hard claim is represented by a *lease*. Both tickets and leases are delivered by certified authorities, and are cryptographically protected to avoid forgery.

A typical situation is that represented on Figure 12.4 (ignore the left part, for the time being): the authority is a resource provider (e.g., a platform provider), and resource principals are hosted services. To get resources, a hosted service uses a two-phase process: it first gets a ticket from a local resource manager (a broker acting for the resource provider); it then uses this ticket to get a lease from that resource provider. It then may use the claimed resources until the end of the specified period.

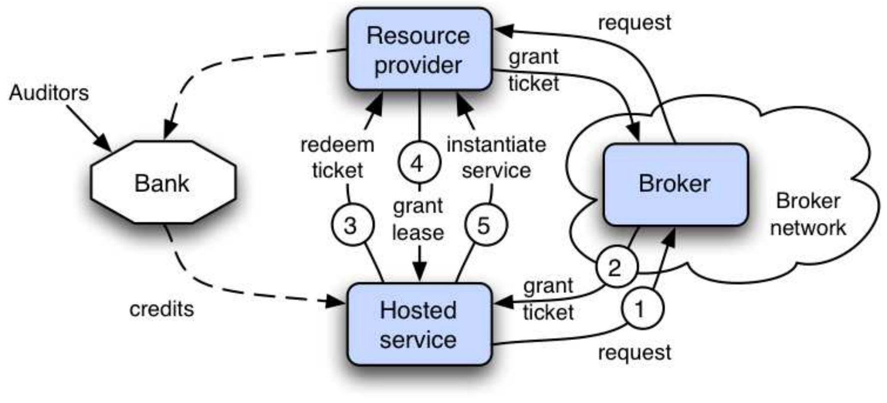


Figure 12.4. Resource allocation in SHARP (from [Fu et al. 2003, Irwin et al. 2005])

After the hosted service has successfully redeemed his ticket with the resource provider, it is granted the set of requested resources, and may then run (serve its own clients) using these resources until the end of the lease period.

The distinction between tickets and leases allows flexibility for resource allocation by delaying the resource allocation decision to the last possible time. Tickets may also be transferred from one resource principal to another one; for instance, a principal may create children and delegate part of its tickets to them; the children may do the same, thus creating a resource delegation tree.

Over-subscription (delivering more tickets than could simultaneously be redeemed) is possible, allowing better satisfaction and improving resource availability, at the risk of occasionally refusing or delaying the granting of a lease. Over-subscription may occur at all levels, i.e., a principal may delegate more tickets than it holds. When processing a tree of nested claims, a simple algorithm examines the claims in bottom up order and locates conflicts due to over-subscription at the lowest common ancestor, thus ensuring

[4]called *agent* in [Fu et al. 2003].

accountability for conflicts.

Tickets and leases are signed by the authority that delivered them. This makes them unforgeable, non-repudiable, and independently verifiable by third parties (e.g., for an audit). The finite duration of leases improves the system's resilience since any resource shall be freed in bounded time, even if its manager fails.

Experiments have shown that the proposed framework improves resource utilization. However, to achieve that goal, the various parameters (degree of over-subscription, duration of the claim, structure of the delegation trees) need to be carefully selected, using experience. The implementation of claims by tickets and leases provides flexibility and allows various trade-offs to be explored, while signatures improve security and traceability.

We now describe a virtual currency model built on top of Sharp, and used in a project called Cereus [Irwin et al. 2005]. While resources may be traded through bartering (identifying mutual coincidence of needs between parties), the introduction of currency allows external regulation of resource allocation policies within a community.

Cereus currency is self-recharging, i.e., spent amounts are restored to each consumer's budget after some time. Each consumer is allocated a number of *credits* in virtual currency (this initial allocation might itself be purchased with actual money). If a consumer has a budget of c credits, it may use them to acquire resources through contracts or auctions. Credits are automatically recharged after a fixed amount of time (say r) from the moment they are spent. A consumer may not spend or commit more than c over any interval of time of length r, and it may not accumulate more than c credits at any time. This prevents hoarding, which might allow malicious actions such as starving other users. Note that this system behaves like lottery scheduling (fair share proportional allocation, see 12.3.2) if the recharge time is set to a very small value. For larger recharge time, the users have more freedom to schedule their requests over time.

In a system in which resources are acquired through auctions, users spend their credits by bidding for resources. A sum committed to a bid at time t is recharged at time $t + r$, which encourages early bidding and discourages canceled bids. Cereus uses an auction protocol in which a broker posts a call price based on recent history, and returns resources in proportion to the bid if the demand exceeds the amount available.

In order to regulate the use of currency, Cereus coordinates all currency transactions through a trusted banking service. Currency is implemented using the claim mechanism provided by Sharp: c credits are represented by a claim for c units of a special type, virtual currency. This claim, called a credit note, is an instance of a Sharp ticket. Credit notes are issued to consumers by trusted banks (left part of Figure 12.4). To spend currency, a consumer delegates control of its credits to a supplier, or to a broker that conducts an auction. In return, the consumer (hosted service) gets resource tickets that it may redeem for leases on the acquired resources (middle part of Figure 12.4). The credits are spent by the brokers to get resources from the suppliers, and are ultimately returned to the banks to be reissued to consumers.

An auditing system allows malicious behavior (overspending, attempting to recharge credits before recharge time) to be detected, since all the needed information is contained in the credit notes.

Other aspects of the economic approach to resource management have been investigated.

The TYCOON project [Feldman et al. 2005] proposes a resource allocation mechanism for shared clusters, based on a continuous reevaluation of users' preferences through an ongoing auction. They explore existence conditions for an equilibrium (in the sense that resources may be allocated in a way that maximizes each user's utility function) and propose strategies for reaching equilibrium with small overhead. TYCOON is close to CEREUS, but places emphasis on agility (fast reaction to change), while CEREUS guarantees medium-term stability through leasing.

The work of [Balazinska et al. 2004] proposes a load management mechanism for loosely coupled distributed systems, through contracts between the participants. Their price-setting mechanism produces "acceptable" allocations, in the sense that *no* participant is overloaded if spare resources are available, and that overload, if it occurs, is shared by *all* participants. This achieves a form of fairness.

12.4 Feedback-controlled Resource Management

An overview of the control aspects of systems management has been presented in 10.2.1. Here we concentrate on the aspects related to performance and resource management, using feedback control. As noted in 10.2.1, the main advantage of using feedback vs. feed-forward control is that it does need a detailed model of the controlled system, and is well adapted to an unpredictable behavior of the load.

In order to apply feedback control to resource management, one needs to specify the model used to represent the system, the variables, both observed and controlled, and the sensors and actuators that allow the controller to interact with the variables, and the policy implemented by the controller. We examine these aspects in turn. See [Hellerstein 2004], [Diao et al. 2005] for more details, and the book [Hellerstein et al. 2004] for an in-depth study of the subject.

12.4.1 Models

There are four main approaches to modeling the resource management and performance aspects of a computing system.

The first approach is empirical, and has been often used, due to the complexity of the actual systems. For instance, in an early effort to design a control algorithm to prevent thrashing in paged virtual memory systems [Brawn and Gustavson 1968], the evolution of the system was represented by a transition graph between three states (normal, underload, overload), characterized by value ranges of resource occupation rates. The controller attempted to prevent the system from getting into the overload state by acting on the degree of multiprogramming (the number of jobs admitted to share memory). Despite its simplicity, this approach proved remarkably efficient. Other empirical models are based on observation and represent the behavior of the system using curve-fitting methods (see an example in [Uttamchandani et al. 2005], using a piecewise linear approximation to model the behavior of a large scale storage utility).

The second approach is to consider the controlled system as a black box, i.e., a device whose internal organization is unknown, and which only communicates with the outside world through a set of sensors and actuators. One also needs to assume that the behavior of the system follows a known law. A common assumption is to consider that the system is linear and time-invariant (see 12.5.4, note 7 for a definition of these terms). Then the numerical values of the model's parameters are determined by performing appropriate experiments (e.g., submitting the system to a periodic input and measuring the response on the output channels), a technique known as system identification. An example of the use of this technique may be found in 12.5.4.

The third approach is based on queueing theory. It considers the system as a queueing network, i.e., a set of interconnected queues, each of which is associated with a specific resource or set of resources. A request follows a path in this network, depending on the resources it needs for its completion. In order to fully specify the model, one needs to define the service law and the inter-arrival law for each queue, and the transition probabilities from one queue to another one. Both analytical and computational solutions for queueing problems have been developed (a book on the applications of queueing theory to computing system is [Lazowska et al. 1984], also available on line[5]).

Models based on queueing systems are commonly used to build workload generators, in order to study the impact of various load parameters on the performance of an application. These models fall into two main classes depending on how job (or request) arrivals are organized. In a closed model, new arrivals are conditioned by job completions, since a job goes back to a fixed size pool upon completion, to be resubmitted later. In an open model, new jobs arrive independently of job completions and completed jobs disappear from the system. Partly open models combine the characteristics of open and closed models, as only part of the job population is recycled upon completion. Open and closed models exhibit vastly different behaviors, as stressed in [Schroeder et al. 2006]. Examples of the use of queueing models for QoS control are presented in this chapter: a closed model in 12.5.5, and an open model in 12.5.6.

The last approach is to represent the behavior of the system by an analytical model. This approach is only applicable when the organization of the controlled system is sufficiently simple and well known. This is often the case for single-resource problems, such as CPU scheduling, as illustrated by the example of 12.5.6

12.4.2 Sensors and Actuators

In order to apply feedback control to resource management, one needs to specify the variables, both observed and controlled, the sensors and actuators that allow the controller to interact with the variables, and the policy implemented by the controller. We first examine the common aspects related to variables.

Since the objective of a resource management policy is to improve QoS, using QoS-related factors as observed variables seems a natural choice. Such variables depend on the relevant aspect of QoS, such as response time (mean, variance, maximum value), throughput, jitter, etc. However, these variables are not always easy to observe (e.g., how to observe the mean response time for a large, widely distributed user population?). Therefore, variables related to resource utilization, which are more readily accessible, are often used. Resource utilization metrics include CPU load, memory occupation, power consumption, I/O channel rate, percentage of servers used in a pool, etc.

[5]http://www.cs.washington.edu/homes/lazowska/qsp/

The problem of relating these variables to the measure of QoS is discussed in 12.3.1.

Actuators for resource management may take a variety of forms:

- Allocate or release a unit of allocation, e.g., an area of storage, a time slice on a CPU, a whole server in a server farm, a share of bandwidth on a communication channel, etc.

- Change the state of a resource, e.g., turn off a server, or modify the frequency of a CPU, in order to reduce energy consumption.

- Add or remove a resource to or from a pool. The resource may be physical (e.g., a server or a memory board), or virtual (e.g., a virtual machine or a virtual cluster).

- Allow or deny a principal the right to compete for resource allocation (in a certain resource pool). One common example is adjusting the multiprogramming level, i.e., the number of jobs or tasks allowed to run concurrently on a node.

We briefly mention specific actuators associated with two resource management techniques: admission control and system reconfiguration.

Admission control relies on regulating a flow of incoming requests. If the decision (admit or reject) depends on the contents of the request, each request must be considered individually. Otherwise, for "anonymous" admission control, one may rely on a method only based on the incoming flow rate, such as the token bucket, a flow-throttling device used for network control. Each token in the bucket gives a right to admit a unit of flow (e.g., a fixed number of bytes); the token is consumed when the unit is admitted. Tokens may be added to the bucket, up to a maximum capacity; if the capacity is exceeded, the bucket "overflows" and no token is added. Thus the two controlling parameters are the token issue rate and the capacity of the bucket (Figure 12.5). Token buckets are used, for example, to regulate admission control in the Seda system (12.5.2).

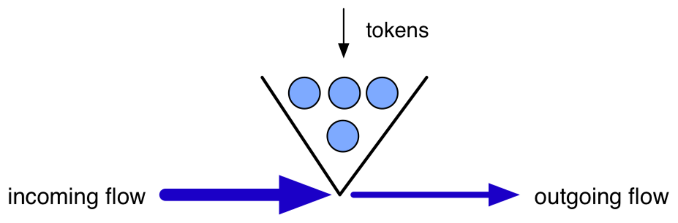


Figure 12.5. The token bucket: an actuator for anonymous admission control

Reallocating resources may involve system reconfiguration. For example, if additional resource units are added to a pool, connections between these units and the rest of the system need to be set up. Techniques for dynamic system reconfiguration are examined in 10.5.

12.4.3 Control Algorithms

In many instances, specially when no model of the system is available, resource management algorithms using feedback are not based on control theory, but use a rule-based approach. For instance, a simplistic admission control algorithm may use thresholds: if the measured QoS indicator (e.g., mean response time) is less than a lower limit, admit; if it exceeds a higher limit, reject. A number of experiments (see [Hellerstein et al. 2004]) have investigated more elaborate algorithms, based on control theory.

The control of computing systems is a new area, which presents several challenges. The controlled systems are usually non-linear, and they are submitted to a highly variable workload. In addition, sensors and effectors are themselves fairly complex systems, and their own dynamics should be integrated into the model of the controlled system.

While many common control algorithms are Single Input-Single Output (SISO), experience (see 12.5.4) has shown that Multiple Input-Multiple Output (MIMO) models more adequately reflect the behavior of complex computing systems. Control algorithms use a proportional (P) or Proportional Integral (PI) control law, assuming that the controlled system is linear, or possibly piecewise linear. Control laws based on the Derivative (D) are seldom used, because they tend to overreact to steep load variations, which are common in Internet services.

Case studies of resource management algorithms based on control theory are presented in 12.5.4 and 12.5.6.

12.4.4 Problems and Challenges

The application of feedback control methods to resource management and QoS in computer systems is still in its infancy. A number of problems still need to be solved.

- Accurate models of real, large scale computing systems are difficult to build, and the performance factors of these systems, as perceived by the clients, are difficult to measure. As a consequence, the correlation between QoS factors and controlled parameters (as set by actuators) is not easy to establish.
- The load on real system has complex, time dependent characteristics, and is difficult to model, even if realistic load generators are now available.
- Most of the methods and tools for solving control problems have been developed for linear systems. However, real systems such as Internet services exhibit highly non-linear behavior, because of such phenomena as saturation and thrashing.

More generally, the communities of computer science and control systems still have different cultures, and progress is needed in the mutual understanding of their respective concepts and methods. Joint projects, examples of which are mentioned in this chapter, should contribute to this goal.

12.5 Case Studies in Resource Management

In order to illustrate the concepts and techniques presented in the previous sections, we examine six case studies, which are representative of the variety of situations and approaches.

Their main features are described in Table 12.1.

sect.	appl.	infrastr.	model	sensors	actuators	diff.	reference
12.5.1	IS[1]	cluster	empiric	resp. time	AC[4], RP[5]	yes	[Blanquer et al. 2005]
12.5.2	IS[1]	cluster	empiric	resp.time	AC[4], RP[5]	yes	[Welsh and Culler 2003]
12.5.3	IS[1]	server	empiric	CPU util.	AC[4]	no	[Cherkasova and Phaal 2002]
12.5.4	WS[2]	server	black-box + ident.	CPU util. mem. util.	MaxClient KeepAlive	no	[Diao et al. 2002a]
12.5.5	IS[1]	cluster	queues	resp. time	AC[4], RP[5]	yes	[Urgaonkar et al. 2007]
12.5.6	RT[3]	server	analytic	CPU util. miss ratio	RP[5] (CPU scheduling)	yes	[Lu et al. 2002]

[1]Internet service
[2]Web server
[3]Real time
[4]Admission control
[5]Resource provisioning

Table 12.1. Case studies in resource management

The case studies are classified according to the following criteria: nature of the application (most applications are Internet services, with the special case of a simple web server); supporting infrastructure (single machine or cluster); model (empirical, analytic, "black-box" with parameter identification); observed and controlled variables (through sensors and actuators, respectively); support for differentiated service.

There are two main approaches to implement a resource management policy.

- Considering the managed system as a black box and installing the management system outside this black box. Examples of this approach are developed in 12.5.1 and 12.5.4.

- Embedding the management system within the managed system; this may be done at various levels, e.g., operating system, middleware, or application. Examples of this approach are developed in 12.5.5 and 12.5.6.

The first approach has the advantage of being minimally invasive, and to be independent of the internal structure of the managed system, which may possibly be unknown, as is the case with legacy systems. The second approach allows a finer grain control of the management policy, but involves access to the internals of the managed systems. The resource management programs also need to evolve together with the application, which entails additional costs.

A mixed approach is to consider the system as an assembly of black boxes, and to install the management system at the boundaries thus defined, provided that the architecture of the managed system is indeed explicit. An example is presented in 12.5.2.

12.5.1 Admission Control using a Black-Box Approach

As an example of the black-box approach, we present Quorum [Blanquer et al. 2005], a management system for cluster-based Internet services. Quorum aims at ensuring QoS guarantees for differentiated classes of service. For each class of service, the SLA is composed of two Service Level Objectives (SLO): minimum average throughput, and maximum response time for 95% of the requests. The SLA may only be guaranteed if it is feasible for the expected load and the cluster capacity, i.e., if 1) the clients of each class announce the expected computation requirements for the requests of the class; 2) the incoming request rate for each class is kept below its guaranteed throughput; and 3) the resources of the cluster are adequately dimensioned. In line with the black-box approach, SLA guarantees are defined at the boundary of the cluster.

Quorum is implemented in a front-end node sitting between the clients and the cluster on which the application is running; this node acts as a load balancer (12.3.2), and is connected to the nodes that support the first tier of the application.

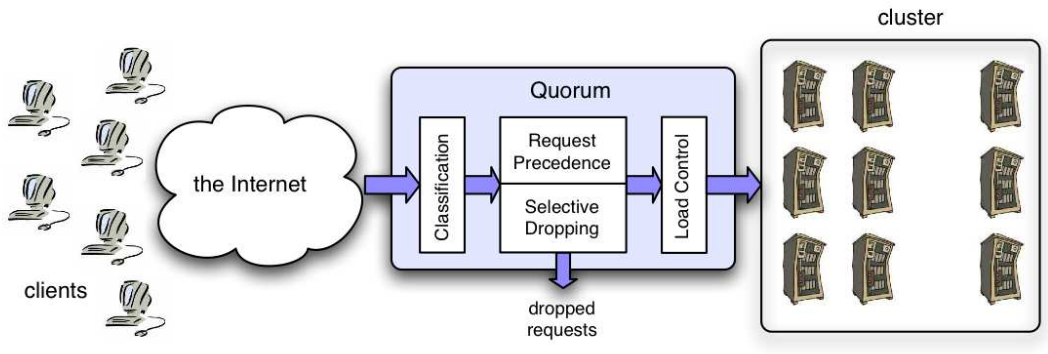


Figure 12.6. The architecture of Quorum (from [Blanquer et al. 2005])

Quorum is composed of four modules, which run on the front-end node (Figure 12.6). Their function is as follows (more details are provided further on).

- The *Classification* module receives all client requests, and determines the service class of each request.
- The *Request Precedence* module determines which proportion of the requests within each class will be transmitted to the cluster.
- The *Selective Dropping* module performs admission control, by discarding the requests for which the specified maximum response time cannot be guaranteed.
- The *Load Control* module releases the surviving requests into the cluster, and determines the rate of this request flow in order to keep the load of the cluster within acceptable bounds.

Thus Quorum controls both the requests to be forwarded to the cluster and the flow rate of these requests.

Load Control regulates the input rate of the requests through a sliding window mechanism similar to that of the TCP transport protocol. The window size determines the number of outstanding requests at any time; it is recomputed periodically (a typical period is 500 ms, a compromise between fast reaction time and significant observation time). The observed parameter is the most restrictive response time for the set of service classes. The current algorithm increments or decrements linearly the window size to keep the response time within bounds.

Request Precedence virtually partitions the cluster resources among the service classes, thus ensuring isolation between classes. The goal is to ensure that the fraction of the global cluster capacity allocated to each class allows the throughput guarantee for this class to be met. The resource share allocated to each class is computed from the guaranteed throughput for this class and the expected computation requirements of the requests. Should the requests for a class exceed these requirements (thus violating the client's part of the contract), the throughput for that class would be reduced accordingly, thus preserving the other classes' guarantees.

In summary, Request Precedence ensures the throughput SLO for each class, while Load Control ensures the response time SLO. Selective Dropping discards the requests for which the response time SLO cannot be achieved under the resource partition needed to guarantee throughput.

Experience with a medium-sized cluster (68 CPUs) and a load replayed from traces of commercial applications shows that Quorum achieves its QoS objectives with a moderate performance overhead (3%), and behaves well under wide fluctuations of incoming traffic, or in the presence of misbehaving classes (i.e., exceeding their announced computation requirements).

Other systems have experimented with the black-box approach for 3-tier applications.

Gatekeeper [Elnikety et al. 2004] does not provide QoS guarantees, but improves the overall performance of the system by delaying thrashing, using admission control. In the absence of a client-side contract, Gatekeeper needs a preliminary setup phase in order to determine the capacity of the system.

Yaksha [Kamra et al. 2004] is based on a queueing model of the managed system, and uses a PI (Proportional Integral) feedback control loop to pilot admission control. It does not provide service differentiation. The SLA only specifies a maximum response time requirement; throughput is subject to best effort. The system is shown to resist to overload, and to rapidly adjust to changes in the workload.

12.5.2 Staged Admission Control

Seda (Staged Event-Driven Architecture [Welsh et al. 2001], [Welsh and Culler 2003]) is an architecture for building highly concurrent Internet services. High concurrency aims at responding to massive request loads. The traditional approach to designing concurrent systems, by servicing each request by a distinct process or thread, suffers from a high overhead under heavy load, both in switching time and in memory footprint. Seda explores an alternative approach, in which an application is built as a network of event-driven stages connected by event queues (Figure 12.7(a)). Within each stage, execution is carried out by a small-sized thread pool. This decomposition has the advantages of modularity, and the composition of stages through event queues provide mutual performance isolation between stages.

The structure of a stage is shown on Figure 12.7(b). A stage is organized around an application-specific event handler, which schedules reactions to incoming events, by activating the threads in the pool. Each invocation of the event handler treats a batch of events, whose size is subject to control, as explained later.

The event handler acts as a finite state machine. In addition to changing the local state, its execution generates zero or more events, which are dispatched to event queues of

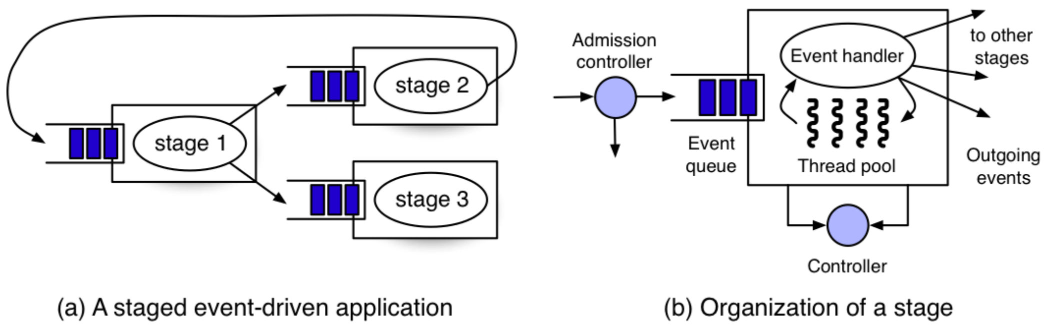


Figure 12.7. The architecture of SEDA (from [Welsh and Culler 2003])

other stages. To ensure performance, the event handling programs should be non-blocking; thus a non-blocking I/O library is used. Note that this structure is similar to that of Click [Kohler et al. 2000], the modular packet router described in 4.5.2.

Two forms of control are used within each stage, in order to maintain performance in the face of overload: admission control is applied to each event queue, and resource allocation within a stage is also subject to control.

Doing admission control on a per stage basis allows focusing on overloaded stages. The admission control policy usually consists in limiting the rate at which a stage accepts incoming events to keep the observed performance at that stage within specified bounds. If an event is rejected by this mechanism, it is the responsibility of the stage that emitted the event to react to the rejection (e.g., by sending the event to another stage, a form of load balancing). In the experiments described, performance metrics is defined by the 90th-percentile response time, smoothed to prevent over-reaction in case of sudden spikes. The policy also implements service differentiation, by defining a separate instance of the admission controller for each customer class. Thus a larger fraction of the lower class requests are rejected, ensuring service guarantees to the higher classes.

Within a stage, resources may be controlled by one or more specialized controllers. One instance is the thread pool controller, which adjusts the number of threads according to the current load of the stage, estimated by the length of the event queue. Another instance is the batching controller, which adjusts the size of the batch of events processed by each invocation of the event handler. The goal of this controller is to maintain a trade-off between the benefits of a large batching factor (e.g., cache locality and task aggregation, which increase throughput), and the overhead it places on response time. Thus the controller attempts to keep the batch size at the smallest value allowing high throughput.

Experiments on a SEDA-based mail server [Welsh and Culler 2003] have shown that the admission control controller is effective in reacting to load spikes, and keeps the response time close to the target. However, the rejection rate may be as high as 80% for massive load spikes. It has also been shown that per-stage admission control rejects less requests than single point admission control.

Another policy for responding to overload is to degrade the QoS of the request being served, in order to reduce the rejection rate at the expense of lower quality (see 12.1.3). Service degradation gives best results when coupled with admission control.

In its current state, SEDA does not attempt to coordinate the admission control decisions made at the different stages of an application. Therefore, a request may be dropped at a late stage, when it has already consumed a significant amount of resources. The work presented in the next section addresses this problem.

12.5.3 Session-Based Admission Control

Session-Based Admission Control [Cherkasova and Phaal 2002] is a method used to improve the performance of commercial web sites by preventing overload. This is essentially an improvement of a basic predictive admission control method based on the individual processing of independent requests, in which the admission algorithm keeps the server close to its peak capacity, by detecting and rejecting requests that would lead to overload.

This method is not suited for a commercial web site, in which the load has the form of sessions. A *session* is a sequence of requests separated by think intervals, each request depending on the previous ones (e.g., browsing the site, making inquiries, moving items into the cart, etc.). If the requests are admitted individually, request rejection may occur anywhere in a session. Thus some sessions may be interrupted and aborted before completion if retries are unsuccessful; for such a session, all the work done till rejection will have been wasted. As a consequence, although the server operates near its full capacity, the fraction of its time devoted to useful work may be quite low for a high server load (and thus a high rejection rate).

Therefore a more appropriate method is to consider sessions (rather than individual requests) as units of admission, and to make the admission decision as early as possible for a session. The admission criterion is based on server utilization, assuming that the server is CPU-bound (the criterion could be extended to include other resources such as I/O or network bandwidth).

A predefined threshold U_{max} specifies the critical server utilization level (a typical value is 95%). The load of the server is predicted periodically (typically every few seconds), based on observations over the last time interval:

$$U^{i+1}_{predicted} = f(i+1), \text{ where } f \text{ is defined by: } \quad \begin{cases} f(1) = U_{max} \\ f(i+1) = (1-k)f(i) + kU^{i}_{measured} \end{cases}$$

Thus the predicted load for interval T_{i+1} is estimated as a weighted mean of the previous estimated load and the load actually measured over the last interval T_i. The coefficient k (the weight of measurement versus extrapolation) is set between 0 and 1. The admission control algorithm is as follows.

- if $U^{i+1}_{predicted} > U_{max}$, then any new session arriving during T_{i+1} will be rejected (a rejection message is send to the client). The server will only process the already accepted sessions.
- if $U^{i+1}_{predicted} \leq U_{max}$, then the server will accept new sessions again.

The weighting factor k allows the algorithm to be tuned by emphasizing responsiveness (k close to 1) or stability (k close to 0). A responsive algorithm (strong emphasis on current observation) tends to be restrictive since it starts rejecting requests at the first sign of overload; the risk is server underutilization. A stable algorithm (strong emphasis on past history) is more permissive; the risk is a higher rate of abortion of already started sessions if the load increases in the future (a session in progress may be aborted in an overloaded server, due to timeouts on delayed replies and to limitation of the number of retries).

This algorithm has been extensively studied by simulation using SpecWeb, a standardized benchmark for measuring basic web server performance. Since sessions (rather than individual requests) are defined as the resource principals, an important parameter of the load is the average length (number of requests) of a session. Thus the simulation was done with three session populations, of respective average length 5, 15 and 50. We present a few typical results (refer to the original paper for a detailed analysis).

With a fully responsive admission control algorithm ($k = 1$), a sampling interval of 1 second, and a threshold $U_{max} = 95\%$

1. The throughput for completed sessions is improved for long sessions (e.g., from 80% to 90% for length 50 at 250% server load[6]); it is fairly stable for medium length sessions, and worse for short sessions. This may be explained by the fact that more short sessions are rejected at high load, with a corresponding rise of the rejection overhead.

2. The percentage of aborted sessions falls to zero for long and medium sessions, at all server loads. It tends to rise for short sessions at high loads (10% at 250% load, 55% at 300% load).

3. The most important effect is the increase in useful server utilization, i.e., the fraction of processor time spent in processing sessions that run to completion (i.e., are not aborted before they terminate). This factor improves from 15% to 70%, 85% and 88% for short, medium and long sessions, respectively, for a 200% server load. This is correlated with result 2 (low rate of aborted sessions).

Lowering the weighting factor k (going from responsive to stable) has the expected effect of increasing the useful throughput at moderate load, but of decreasing it at higher load, where more sessions are aborted due to inaccurate prediction (the threshold load value is about 170% for 15-request sessions). This suggests using an adaptive policy in which k would be dynamically adjusted by observing the percentage of refused connections and aborted requests. If this ratio rises, k is increased to make the admission policy more restrictive; if it falls to zero, k is gradually decreased.

Another dynamic strategy consists in trying to predict the number of sessions that the server will be able to process in the next interval, based on the observed load and on workload characterization. The server rejects any new session above this quota.

Both dynamic strategies (adapting the responsiveness factor k and predicting the acceptable load) have been tried with different load patterns drawn from experience ("usual day" and "busy day"). Both strategies were found to reduce the number of aborted sessions and thus to improve the useful throughput, the predictive strategy giving the best results.

[6]the server load is the ratio between the aggregated load generated by the requests and the server processing capacity.

12.5.4 Feedback Control of a Web Server

The work reported in [Diao et al. 2002a], [Diao et al. 2002b] is an early attempt to apply feedback control to the management of an Internet service (a single Apache web server delivering static content).

The controlled variables are CPU and memory utilization, denoted by CPU and MEM, respectively, which are easily accessible low-level variables, assuming that the settings of these variables are correlated to QoS levels such as expressed in an SLA. The controlling variables are those commonly used by system administrators, namely *MaxClients* (MC), the maximum number of clients that can connect to the server, and *KeepAlive Timeout* (KA), which determines how long an idle connection is maintained in the HTTP 1.1 protocol. This parameter allows an inactive client (or a client whose think time has exceeded a preset limit) to be disconnected from the server, thus leaving room to serve other clients.

The Apache web server is modeled by a black box, with two inputs (MC and KA) and two outputs (CPU and MEM). The behavior of this black box is assumed to be linear and time-invariant[7]. In a rough first approximation, CPU is strongly correlated with KA, and MEM is strongly correlated with MC. This suggests modeling the evolution of the server by the two Single Input, Single Output (SISO) equations:

$$\begin{aligned} CPU(k+1) &= a_{CPU}CPU(k) + b_{CPU}KA(k) \\ MEM(k+1) &= a_{MEM}MEM(k) + b_{MEM}MC(k) \end{aligned}$$

where time is discretized ($X(k)$ denotes the value of X at the k-th time step), and the as and bs denotes constant coefficients, to be determined by identification (least squares regression with discrete sine wave inputs).

A more general model, which does not assume a priori correlations, is the Multiple Input, Multiple Output (MIMO) model:

$$\mathbf{y}(k+1) = \mathbf{A}\mathbf{y}(k) + \mathbf{B}\mathbf{u}(k)$$

in which

$$\mathbf{y}(k) = \begin{bmatrix} CPU(k) \\ MEM(k) \end{bmatrix}, \mathbf{u}(k) = \begin{bmatrix} KA(k) \\ MC(k) \end{bmatrix},$$

and $\mathbf{A}$ and $\mathbf{B}$ are 2×2 constant matrices, again determined by identification.

The first step is to test the accuracy of the models. Experience with a synthetic workload generator, comparing actual values with those predicted by the model, gives the following results.

- While the SISO model of MEM is accurate, the SISO model of CPU provides a poor fit to actual data, specially with multiple step inputs.
- Overall, the MIMO model gives a more accurate fit than the dual SISO model.
- The accuracy of the models degrades near the ends of the operating region, showing the limits of the linearity assumption.

[7]A system with input $\mathbf{u}$ and output $\mathbf{y}$ is *linear* if $\mathbf{y}(a\mathbf{u}) = a\mathbf{y}(\mathbf{u})$ and $\mathbf{y}(\mathbf{u}_1 + \mathbf{u}_2) = \mathbf{y}(\mathbf{u}_1) + \mathbf{y}(\mathbf{u}_2)$. It is *time-invariant* if its behavior is insensitive to a change of the origin of time.

The second step is to use the model for feedback control. The controller is Proportional Integral (PI), with gain coefficients determined by classical controller design methods (pole placement for SISO, and cost function minimization for MIMO). Experience shows that the SISO model performs well in spite of its inaccuracy, due to the limited degree of coupling between KA and MC. However, the MIMO model does better overall, specially for heavy workload.

The contributions of this work are to show that control theory can indeed be applied to Internet services, and that good results can be obtained with a simple design. In that sense, this work is a proof of concept experiment. However, being one of the first attempts at modeling an Internet service, it suffers from some limitations.

- The model is linear, while experience shows that actual systems exhibit a non-linear behavior due to saturation and thrashing.
- The model uses low-level system parameters as controlled variables, instead of user-perceived QoS factors. In that sense, it does not solve the SLA decomposition problem, as presented in 12.3.1.
- The model considers a single server, while clusters are more common.
- The workload generator does not create wide amplitude load peaks.

More recent models, such as that presented in the next section, aim at overcoming these limitations.

Another early attempt at applying feedback control to web server performance is [Abdelzaher et al. 2002]. They use a SISO algorithm, in which the measured variable is system utilization and the controlled variable is a single index that determines the fraction of clients to be served at each service level (assuming the service to be available at various levels of degradation). This index is the input of an admission control actuator.

12.5.5 Resource Provisioning for a Multi-tier Service

The work reported in [Urgaonkar et al. 2007] differs from that presented in the previous sections, in that it is based on an analytical model of the managed system. This model may be used to predict response times, and to drive resource allocation, using both dynamic resource provisioning and admission control.

The class of systems under study is that of multi-tiered systems (12.1.3). A multi-tiered system is represented as a network of queues: queue Q_i represents the i-th tier ($i = 1, 2, \ldots, M$). In the first, basic, version of the model, there is a single server per tier. A request goes from tier 1 to tier M, receiving service at each tier, as shown on Figure 12.8, which represents a 3-tier service.

The model represents a session as a set of requests (cf 12.5.3). Sessions are represented by a special queueing system, Q_0, which consists of N "servers", which act as request generators. Each server in Q_0 represents a session: it emits successive requests, separated by a think time. When a session terminates, a new one is started at the same server; thus Q_0 represents a steady load of N sessions.

The model takes the following aspects into account:

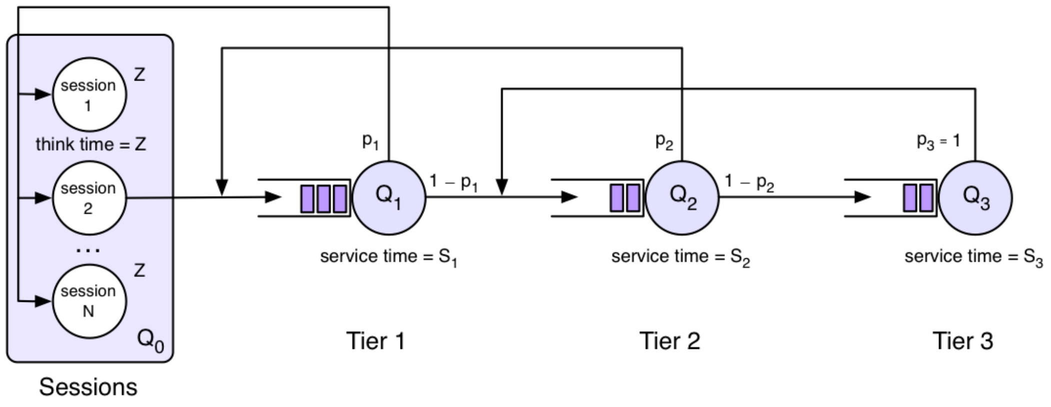


Figure 12.8. Queueing network for a 3-tier service (adapted from [Urgaonkar et al. 2007])

- A request may visit a tier (say number k) several times (e.g., if it makes several queries in a database).
- A request may go no further than a certain tier (e.g., because the answer was found in a cache in that tier, and no further processing is needed).

This is represented in the model by a transition from tier k to tier $k-1$. The two above cases are represented using different transition probabilities. A request terminates when it goes back from Q_1 to Q_0.

The parameters of the basic version of the model are the service times S_i at each tier i, the transition probabilities between queues (p_i is the transition probability from Q_i to Q_{i-1}), and the think time of the user, represented by the service time of the servers in Q_0. Numerical values of these parameters, for a specific application, are estimated by monitoring the execution of this application under a given workload. Once these parameters are known, the system may be resolved, using the Mean-Value Analysis (MVA) algorithm [Reiser and Lavenberg 1980] for closed queueing networks, to compute the mean response time.

Several enhancement have been added to the basic version.

- Modeling multiple servers. Internet services running on clusters use replicated tiers for efficiency and availability. This is represented by multiple queues at each tier: the single queue Q_i is replaced by r_i queues $Q_{i,1}, \ldots, Q_{i,r_i}$, where r_i is the degree of replication at tier i. As described in 12.3.2, a load balancer forwards requests from one tier to the next, and dispatches them across replicas. A parameter of the model captures the possible imbalance due to specific constraints such as stateful servers or cache affinity.
- Handling concurrency limits. There is a limit to the number of activities that a node may concurrently handle. This is represented by a drop probability (some requests are dropped due to concurrency limits).
- Handling multiple session classes. Given the parameters of each class (think time, service times, etc.), the model allows the average response time to be computed on

a per-class basis.

The model has some limitations: a) Each tier is modeled by a single queue, while requests actually compete for several different resources: CPU, memory, network, disk. b) The model assumes that a request uses the resources of one tier at a time. While this is true for many Internet services, some applications such as streaming servers use pipeline processing, not currently represented by the model.

Experiments with synthetic loads have shown that the model predicts the response time of applications within the 95% confidence interval. In addition, the model has been used to assist two aspects of resource management: dynamic resource provisioning (for predictable loads) and admission control (for unexpected peaks).

1. *Dynamic resource provisioning.* Assume that the workload of the application may be predicted for the short term, in terms of number of sessions and characteristics of the sessions. Then resource provisioning works as follows: define an initial server assignment for each tier (e.g., one server per tier). Use the model to determine the average response time. If it is worse than the target, examine the effect of adding one more server to each tier that may be replicated. Repeat until the response time is below the target. This assumes that the installation has sufficient capacity (total number of servers to meet the target). The complexity of the computation is of the order of $M.N$ (M number of tiers, N number of sessions), which is acceptable if resources are provisioned for a fairly long period (hours).

2. *Admission control.* Admission control is intended to react to unexpected workload peaks. In the model under study, it is performed by a front-end node, which turns out excess sessions to guarantee the SLA (here the response time). The criterion for dropping requests is to maximize a utility function, which is application dependent. For example, the utility function may be a global revenue generated by the admitted sessions; assigning a revenue to each session class allows service differentiation. In the experiments described, the system admits the sessions in non-increasing order of generated revenue, determining the expected response time according to the model, and drops a session if its admission would violate the SLA for at least one session class.

The interest of this work is that it provides a complete analytical model for a complex system: a multi-tiered application with replicated tiers, in the presence of a complex load. The system is able to represent such features as load unbalancing and caching effects, and allows for differentiated service. Its main application is capacity planning and dynamic resource provisioning for loads whose characteristics may be predicted with fair accuracy.

Similar queueing models have been used for correlating SLOs with system-level thresholds [Chen et al. 2007], and for identifying the bottleneck tier in a multi-tier system, in order to apply a control algorithm for service differentiation [Diao et al. 2006].

12.5.6 Real-Time Scheduling

[Lu et al. 2002] have explored the use of feedback control theory for the systematic design of algorithms for real-time scheduling. This is a single-resource problem: a number of tasks, both periodic and aperiodic, compete for the CPU. Each task must meet a deadline; for periodic tasks, the deadline is set for each period, and for non-periodic tasks, the deadline is relative to the arrival time. The arrival times (for aperiodic tasks) and CPU utilization rates are unknown, but estimates are provided for each task.

The controlled variables, i.e., the performance metrics controlled by the scheduler, are sampled at regular intervals (with a period W), and defined over the kth sampling window $[(k-1)W, kW]$. These variables include:

- The deadline miss ratio $M(k)$, the number of deadline misses divided by the number of completed and aborted tasks in the window.
- The CPU utilization $U(k)$, the percentage of CPU busy time in the window.

Reference values M_S and U_S are set for these variables (e.g. $M_S = 0$ and $U_S = 90\%$). Three variants of the control system have been studied: FC-U, which only controls $M(k)$, FC-M, which only controls $U(k)$, and FC-UM, which combines FC-U and FC-M as described later.

There is a single manipulated variable, on which the scheduler may act to influence the system's behavior: $B(k)$, the total estimated CPU utilization of all tasks in the system. The actuator changes the value of $B(k)$ according to the control algorithm. Then an optimization algorithm determines the precise allocation of CPU to tasks over the $(k+1)$th window so that their total CPU utilization is $B(k+1)$. In addition to being called at each sampling instant, the actuator is also invoked upon the arrival of each task, to correct prediction errors on arrival times..

The optimization algorithm relies on the notions of QoS level and value. Each task T_i is assigned a certain number (at least two) of QoS levels, which determine possible conditions of execution, defined by a few attributes associated with each level j, among which a relative deadline $D_i[j]$, an estimated CPU utilization $B_i[j]$ and a value $V_i[j]$. Level 0 corresponds to the rejection of the task; both $B_i[0]$ and $V_i[0]$ are set[8] to 0. The value $V_i[j]$ expresses the contribution of the task (as estimated by the application designer) if it meets its deadline $D_i[j]$ at level j; if it misses the deadline, the contribution is $V_i[0]$ (equal or less than 0). Both $B_i[j]$ and $V_i[j]$ increase with j.

The optimization algorithm assigns a (non-zero) QoS level to each task to maximize its value density (the ratio $V_i[j]/B_i[j]$, defined as 0 for level 0), and then schedules the tasks in the order of decreasing density until the total prescribed CPU utilization $B(k)$ is reached.

Intuitively, a system with two levels of QoS corresponds to an admission control algorithm: a task is either rejected (level 0) or scheduled at its single QoS level (level 1); the value density criterion gives priority to short, highly valued tasks. Increasing the number of levels allows for a finer control over the relative estimated value of a task's completion: instead of deciding to either run or reject a task, the algorithm may chose to run it at an intermediate level of QoS for which the ratio between contribution and consumed resources qualifies the task for execution.

The controller algorithm uses a simple proportional control function (Figure 12.9), i.e., the control function is $D(k) = K.E(k)$, where $E(k)$ (the error), is the difference between

[8]$V_i[0]$ may also be set to a negative value, which corresponds to a penalty.

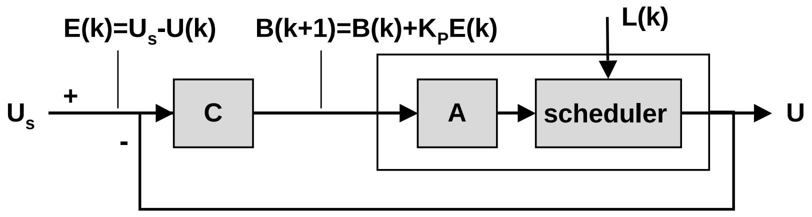


Figure 12.9. Feedback control of CPU scheduling

the reference value and the measured output, i.e., either $M_S - M(k)$ or $U_S - U(k)$ depending on the controlled variable. Then the command to the actuator is $B(k+1) = B(k) + D(k)$.

Combining FC-U and FC-M is done as follows: the two separate control loops are set up, then the two command signals $D(k)$ are compared, and the lower signal is selected.

The disturbance is the actual load, which introduces an uncertainty factor since the estimates of arrival times and CPU consumption may be incorrect.

A complete analysis of this system has been done, using the control theory methodology. The system is non-linear, due to saturation. This can be modeled by two different regimes separated by a threshold; each of the regimes can be linearized. Then the analysis determines stability conditions and allows the parameters to be tuned for various load characteristics.

The experimental results obtained with various synthetic loads show that the system is stable in the face of overload, that it has low overshoot in transient states, and that it achieves an overall better utilization of CPU than standard open-loop control algorithms such as EDF. In addition, the system has all the benefits of the availability of a model, i.e., performance prediction and help in parameter selection.

12.6 Conclusion

We summarize a few conclusions on resource management to achieve QoS requirements.

- Resource management has long been guided by heuristics. The situation is changing, due to a better understanding of SLA decomposition (the relationship between user-perceived QoS and internal resource occupation parameters). This in turn results from the elaboration of more accurate models of the managed systems, and from the development of better statistical correlation methods.

- The application of control theory to resource management is still in an early stage. This may be explained by the inherent complexity of the systems under study and by their highly non-linear and time-dependent character. Control-based techniques should also benefit from the above-mentioned progress in SLA decomposition. Most likely, successful methods will combine predictive (model-based) and reactive (feedback-controlled) algorithms.

- Admission control appears to be the dominant strategy for dealing with unexpected load peaks. It may be made minimally invasive, using a black box approach, and its actuators are easy to implement.

However, this apparent simplicity should not offset the difficulty of the design of a good admission control algorithm. As many experiments have shown, a good knowledge of the load can be usefully exploited, again mixing predictive and reactive approaches.

- A number of aspects need further investigation, e.g., allocation problems for multiple resources, interaction between resource management for multistage requests (such as multiple tiers), further elaboration of the contract satisfaction criteria (probabilistic, guaranteed).

12.7 Historical Note

Resource allocation was one the main areas of operating systems research in the mid-1960s. The pitfalls of resource management, such as deadlock and starvation, were identified at that time, and solutions were proposed. The influence of resource allocation on user-perceived performance was also investigated, using both queueing models and empirical approaches. Capacity planning models were used for dimensioning mainframes. The problems of resource overcommitment, leading to thrashing, were understood in the late 1960s, and the principle of their solution (threshold-based control of the multiprogramming degree) is still valid today.

The notion of Quality of Service was not present as such for operating systems (although similar concepts were underlying the area of performance management). QoS was introduced in networking, first in the form of support for differentiated service, then for application-related performance. QoS emerged as an important area of study with the advent of the first multimedia applications (e.g., streaming audio and video), for which the “best effort” philosophy of the Internet Protocol was inappropriate. Techniques such as channel reservation, traffic policing, scheduling algorithms, and congestion avoidance were developed. [Wang 2001] presents the main concepts and techniques of Internet QoS.

In the 1980s, heuristic-based feedback control methods were successfully applied to network operation (e.g., for adaptive routing in the Internet, for flow control in TCP). The extension of QoS to middleware systems was initiated in the 1990s (see e.g., [Blair and Stefani 1997]).

The rapid rise of Internet services in the 2000s, and the growing user demand for QoS guarantees, have stimulated efforts in this area. As a part of the autonomic computing movement (see 10.2), several projects have attempted to apply control theory to the management of computing systems, specially for performance guarantees. [Hellerstein et al. 2004] gives a detailed account of these early efforts. In addition to feedback control, current research topics are the development of accurate models for complex Internet services, and the use of statistical methods to characterize both the service request load and the behavior of the managed systems.

Chapter 13

Security

This chapter is still unavailable.

References

[Abbott and Peterson 1993] Abbott, M. B. and Peterson, L. L. (1993). A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19.

[Abd-El-Malek et al. 2005] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. (2005). Fault-Scalable Byzantine Fault-Tolerant Services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 59–74. *Operating Systems Review*, 39(5), December 2005.

[Abdelzaher et al. 2002] Abdelzaher, T. F., Shin, K. G., and Bhatti, N. (2002). Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96.

[Aberer et al. 2005] Aberer, K., Alima, L. O., Ghodsi, A., Girdzijauskas, S., Hauswirth, M., and Haridi, S. (2005). The essence of P2P: A reference architecture for overlay networks. In *Proceedings of 5th IEEE International Conference on Peer-to-Peer Computing*, pages 11–20, Konstanz, Germany.

[Adjie-Winoto et al. 1999] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., and Lilley, J. (1999). The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 186–201, Kiawah Island Resort, near Charleston, South Carolna.

[Aguilera et al. 2000] Aguilera, M. K., Chen, W., and Toueg, S. (2000). Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125.

[Aguilera et al. 2003] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. (2003). Performance Debugging for Distributed systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 74–89. ACM Press.

[Aguilera et al. 1999] Aguilera, M. K., Strom, R. E., Sturman, D. C., Astley, M., and Chandra, T. D. (1999). Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing (PODC)*, pages 53–61.

[Akkoyunlu et al. 1975] Akkoyunlu, E. A., Ekanadham, K., and Huber, R. V. (1975). Some constraints and tradeoffs in the design of network communications. *SIGOPS Operating Systems Review*, 9(5):67–74.

[Albitz and Liu 2001] Albitz, P. and Liu, C. (2001). *DNS and BIND*. O'Reilly. 4th ed., 622 pp.

[Aldrich 2005] Aldrich, J. (2005). Open Modules: Modular Reasoning about Advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming, Glasgow, UK*, number 3586 in Lecture Notes in Computer Science, pages 144–168. Springer-Verlag.

[Aldrich et al. 2002] Aldrich, J., Chambers, C., and Notkin, D. (2002). ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 187–197, Orlando, FL, USA.

[Alexander 1964] Alexander, C. (1964). *Notes on the Synthesis of Form*. Harvard University Press.

[Alexander et al. 1977] Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press. 1216 pp.

[Allen 1997] Allen, R. (1997). *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science. Issued as CMU Technical Report CMU-CS-97-144.

[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.

[Alonso et al. 2004] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004). *Web Services*. Springer. 354 pp.

[Alsberg and Day 1976] Alsberg, P. A. and Day, J. D. (1976). A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE'76)*, pages 627–644, San Francisco, California, USA.

[Alvisi et al. 2007] Alvisi, L., Doumen, J., Guerraoui, R., Koldehofe, B., Li, H., van Renesse, R., and Tredan, G. (2007). How robust are gossip-based communication protocols? *ACM SIGOPS Operating Systems Review, Special Issue on Gossip-Based Networking*, pages 14–18.

[Amza et al. 2002] Amza, C., Cecchet, E., Chanda, A., Cox, A., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., and Zwaenepoel, W. (2002). Specification and Implementation of Dynamic Web Site Benchmarks. In *WWC-5: IEEE 5th Annual Workshop on Workload Characterization*, Austin, TX, USA.

[Andersen et al. 2001] Andersen, D. G., Balakrishnan, H., Kaashoek, M. F., and Morris, R. (2001). Resilient Overlay Networks. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, Banff, Canada.

[Anderson 2003] Anderson, P. (2003). The Complete Guide to LCFG (Local ConFiGuration system). http://www.lcfg.org/doc/guide.pdf.

[Andrade et al. 1996] Andrade, J. M., Carges, M., Dwyer, T., and Felts, S. D. (1996). *The Tuxedo System: Software for Constructing and Managing Distributed Business Applications*. Addison-Wesley. 444 pp.

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

[Appia] Appia. Appia comunication framework. http://appia.di.fc.ul.pt.

[Appleby et al. 2001] Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. (2001). Oceano - SLA based management of a computing utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*.

[Arntsen et al. 2008] Arntsen, A.-B., Mortensen, M., Karlsen, R., Andersen, A., and Munch-Ellinsen, A. (2008). Flexible transaction processing in the Argos middleware. In *Proceedings of the 2008 EDBT Workshop on Software Engineering for Tailor-made Data Management, Nantes, France*, pages 12–17.

[Aron et al. 2000] Aron, M., Druschel, P., and Zwaenepoel, W. (2000). Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 90–101, Santa Clara, California.

[ASM] ASM. A Java bytecode manipulation framework. http://www.objectweb.org/asm.

[Aspnes 2003] Aspnes, J. (2003). Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175.

[Avižienis et al. 2004] Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[Babaoğlu and Marzullo 1993] Babaoğlu, Ö. and Marzullo, K. (1993). Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In Mullender, S., editor, *Distributed Systems*, pages 55–96. Addison-Wesley.

[Babaoğlu and Toueg 1993] Babaoğlu, Ö. and Toueg, S. (1993). Non-Blocking Atomic Commitment. In Mullender, S., editor, *Distributed Systems*, pages 147–168. Addison-Wesley.

[Baduel et al. 2006] Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., and Quilici, R. (2006). *Grid Computing: Software Environments and Tools*, chapter "Programming, Deploying, Composing for the Grid". Springer-Verlag.

[Balazinska et al. 2004] Balazinska, M., Balakrishnan, H., and Stonebraker, M. (2004). Contract-Based Load Management in Federated Distributed Systems. In *First USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 197–210, San Francisco, CA, USA.

[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.

[Ban 1998] Ban, B. (1998). Design and Implementation of a Reliable Group Communication Toolkit for Java. Technical Report, Dept. of Computer Science, Cornell University, September 1998. http://www.jgroups.org/.

[Banavar et al. 1999] Banavar, G., Chandra, T. D., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C. (1999). An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 262–272, Austin, Texas.

[Banga et al. 1999] Banga, G., Druschel, P., and Mogul, J. C. (1999). Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, Louisiana.

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

[Bartlett 1981] Bartlett, J. F. (1981). A NonStop Kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 22–29, New York, NY, USA. ACM.

[Bartlett and Spainhower 2004] Bartlett, W. and Spainhower, L. (2004). Commercial Fault tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96.

[Bavier et al. 2004] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. (2004). Operating System Support for Planetary-Scale Network Services. In *First USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 253–266, San Francisco, CA, USA.

[BCEL] BCEL. Byte Code Engineering Library. http://jakarta.apache.org/bcel.

[Berenson et al. 1995] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ANSI SQL isolation levels. In *SIGMOD'95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, New York, NY, USA. ACM.

[Bernardo et al. 2002] Bernardo, M., Ciancarini, P., and Donatiello, L. (2002). Architecting Families of Software Systems with Process Algebras. *ACM Transactions on Software Engineering and Methodology*, 11(4):386–426.

[Bernstein and Goodman 1981] Bernstein, P. A. and Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221.

[Bernstein et al. 1987] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. 370 pp.

[Besancenot et al. 1997] Besancenot, J., Cart, M., Ferrié, J., Guerraoui, R., Pucheral, Ph., and Traverson, B. (1997). *Les systèmes transactionnels : concepts, normes et produits*. Hermès. 416 pp.

[Bettstetter and Renner 2000] Bettstetter, C. and Renner, C. (2000). A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol. In *Proceedings of the Sixth EUNICE Open European Summer School*, Twente, Netherlands.

[Beugnard et al. 1999] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making Components Contract Aware. *IEEE Computer*, 32(7):38–45.

[Bidinger and Stefani 2003] Bidinger, Ph. and Stefani, J.-B. (2003). The Kell calculus: operational semantics and type system. In *Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 03)*, Paris, France.

[Bieber and Carpenter 2002] Bieber, G. and Carpenter, J. (2002). Introduction to Service-Oriented Programming. http://www.openwings.org.

[Birman 2007] Birman, K. (2007). How robust are gossip-based communication protocols? *ACM SIGOPS Operating Systems Review, Special Issue on Gossip-Based Networking*, pages 8–13.

[Birman and Joseph 1987] Birman, K. P. and Joseph, T. A. (1987). Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP'87)*, pages 123–138.

[Birrell et al. 1982] Birrell, A. D., Levin, R., Schroeder, M. D., and Needham, R. M. (1982). Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274.

[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.

[Blahut 2003] Blahut, R. (2003). *Algebraic Codes for Data Transmission*. Cambridge University Press. 482 pp.

[Blair and Stefani 1997] Blair, G. and Stefani, J.-B. (1997). *Open Distributed Processing and Multimedia*. Addison-Wesley. 452 pp.

[Blanquer et al. 2005] Blanquer, J. M., Batchelli, A., Schauser, K., and Wolski, R. (2005). Quorum: Flexible Quality of Service for Internet Services. In *Second USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, pages 159–174, Boston, Mass., USA.

[Boender et al. 2006] Boender, J., Di Cosmo, R., Durak, B., Leroy, X., Mancinelli, F., Morgado, M., Pinheiro, D., Treinen, R., Trezentos, P., and Vouillon, J. (2006). News from the EDOS project: improving the maintenance of free software distributions. In Berger, O., editor, *Proceedings of the International Workshop on Free Software (IWFS'06)*, pages 199–207, Porto Allegre, Brazil.

[Bouchenak et al. 2006] Bouchenak, S., De Palma, N., Hagimont, D., and Taton, C. (2006). Autonomic management of clustered applications. In *IEEE International Conference on Cluster Computing*, Barcelona, Spain.

[Bracha and Toueg 1985] Bracha, G. and Toueg, S. (1985). Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840.

[Brawn and Gustavson 1968] Brawn, B. and Gustavson, F. (1968). Program behavior in a paging environment. In *Proceedings of the AFIPS Fall Joint Computer Conference (FJCC'68)*, pages 1019–1032.

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

[Brownbridge et al. 1982] Brownbridge, D. R., Marshall, L. F., and Randell, B. (1982). The Newcastle Connection — or UNIXes of the World Unite! *Software–Practice and Experience*, 12(12):1147–1162.

[Bruneton 2001] Bruneton, É. (2001). *Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties*. PhD thesis, Institut National Polytechnique de Grenoble.

[Bruneton et al. 2006] Bruneton, É., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The Fractal Component Model and its Support in Java. *Software–Practice and Experience*, 36(11-12):1257–1284. special issue on "Experiences with Auto-adaptive and Reconfigurable Systems".

[Bruneton et al. 2002] Bruneton, É., Coupaye, T., and Stefani, J.-B. (2002). Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga (Spain).

[Burgess 1995] Burgess, M. (1995). Cfengine: a site configuration engine. *USENIX Computing Systems*, 8(3).

[Buschmann et al. 1995] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1995). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons. 467 pp.

[Busi and Zavattaro 2001] Busi, N. and Zavattaro, G. (2001). Publish/subscribe vs. shared dataspace coordination infrastructures. In *In Proc. of IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'01)*, pages 328–333. IEEE Press.

[CACM 1993] CACM (1993). *Communications of the ACM*, special issue on concurrent object-oriented programming. 36(9).

[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.

[Campbell and Habermann 1974] Campbell, R. H. and Habermann, A. N. (1974). The specification of process synchronization by path expressions. In Gelenbe, E. and Kaiser, C., editors, *Operating Systems, an International Symposium*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer Verlag.

[Candea et al. 2004] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. (2004). Microreboot – A Technique for Cheap Recovery. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 31–44, San Francisco, CA, USA.

[Cardellini et al. 2002] Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. S. (2002). The state of the art in locally distributed Web-server systems. *ACM Computing Surveys*, 34(2):263–311.

[Carol 2005] Carol (2005). Common Architecture for RMI ObjectWeb Layer. The ObjectWeb Consortium. http://carol.objectweb.org.

[Carriero and Gelernter 1989] Carriero, N. and Gelernter, D. (1989). Linda in context. *Communications of the ACM*, 32(4):444–458.

[Carzaniga et al. 2001] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2001). Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383.

[Castro and Liskov 1999] Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 173–186, New Orleans, Louisiana.

[Castro and Liskov 2002] Castro, M. and Liskov, B. (2002). Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461. Based on work published in *Usenix OSDI'99* (173–186) and *OSDI'00* (273–288).

[CCM] CCM. The CORBA Component Model. Object Management Group Specification. http://www.omg.org/technology/documents/formal/components.htm.

[Cecchet et al. 2008] Cecchet, E., Ailamaki, A., and Candea, G. (2008). Middleware-based Database Replication: The Gaps between Theory and Practice. In *SIGMOD'08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, June 9–12 2008, Vancouver, Canada.

[Cecchet et al. 2004] Cecchet, E., Marguerite, J., and Zwaenepoel, W. (2004). C-JDBC: Flexible Database Clustering Middleware. In *Proc. USENIX Annual Technical Conference, Freenix Track*, Boston, MA, USA.

[Chakrabarti et al. 2002] Chakrabarti, A., de Alfaro, L., Henzinger, T. A., Jurdzinski, M., and Mang, F. Y. (2002). Interface Compatibility Checking for Software Modules. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 428–441. Springer-Verlag.

[Chan and Chuang 2003] Chan, A. T. S. and Chuang, S.-N. (2003). MobiPADs: a reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29(12):1072–1085.

[Chandra et al. 2007] Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *Proceedings of the Twenty-sixth ACM Symposium on Principles of Distributed Computing (PODC'07)*, pages 398–407, New York, NY, USA. ACM.

[Chandra et al. 1996a] Chandra, T. D., Hadzilacos, V., and Toueg, S. (1996a). The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722.

[Chandra et al. 1996b] Chandra, T. D., Hadzilacos, V., Toug, S., and Charron-Bost, B. (1996b). On the Impossibility of Group Membership. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC).*

[Chandra and Toueg 1991] Chandra, T. D. and Toueg, S. (1991). Unreliable failure detectors for reliable distributed systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 325–340.

[Chandra and Toueg 1996] Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267. A preliminary version appeared in [Chandra and Toueg 1991].

[Chandy and Lamport 1985] Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75.

[Chandy et al. 1998] Chandy, K. M., Rifkin, A., and Schooler, E. (1998). Using Announce-Listen with Global Events to Develop Distributed Control Systems. *Concurrency: Practice and Experience*, 10(11-13):1021–1028.

[Chase et al. 2003] Chase, J. S., Irwin, D. E., Grit, L. E., Moore, J. D., and Sprenkle, S. E. (2003). Dynamic virtual clusters in a grid site manager. In *Proceedings 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, Seattle, Washington.

[Chen et al. 2002] Chen, W., Toueg, S., and Aguilera, M. K. (2002). On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580.

[Chen et al. 2007] Chen, Y., Iyer, S., Liu, X., Milojicic, D., and Sahai, A. (2007). SLA Decomposition: Translating Service Level Objectives to System Level Thresholds. Technical Report HPL-2007-17, Hewlett-Packard Laboratories, Palo Alto.

[Cheng et al. 2004] Cheng, S.-W., Huang, A.-C., Garlan, D., Schmerl, B., and Steenkiste, P. (2004). An Architecture for Coordinating Multiple Self-Management Systems. In *The 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4)*.

[Cheriton and Mann 1989] Cheriton, D. R. and Mann, T. P. (1989). Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183.

[Cherkasova and Phaal 2002] Cherkasova, L. and Phaal, P. (2002). Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers*, 51(6):669–685.

[Chockler et al. 2001] Chockler, G. V., Keidar, I., and Vitenberg, R. (2001). Group Communication Specifications: a Comprehensive Study. *ACM Computing Surveys*, 33(4):427–469.

[Ciancarini 1996] Ciancarini, P. (1996). Coordination Models and Languages as Software Integrators. *ACM Computing Surveys*, 28(2):300–302.

[Clark 1988] Clark, D. D. (1988). The design philosophy of the DARPA Internet protocols. In *Proc. SIGCOMM'88, Computer Communication Review, vol.18(4)*, pages 106–114, Stanford, CA. ACM.

[Clarke et al. 2001] Clarke, M., Blair, G. S., Coulson, G., and Parlavantzas, N. (2001). An Efficient Component Model for the Construction of Adaptive Middleware. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, LNCS 2218*, pages 160–178. Springer-Verlag.

[Cohen et al. 2004] Cohen, I., , Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. (2004). Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 31–44, San Francisco, CA, USA.

[Comer and Peterson 1989] Comer, D. and Peterson, L. L. (1989). Understanding Naming in Distributed Systems. *Distributed Computing*, 3(2):51–60.

[Condie et al. 2005] Condie, T., Hellerstein, J. M., Maniatis, P., Rhea, S., and Roscoe, T. (2005). Finally, a use for componentized transport protocols. In *Proceedings of the Fourth Workshop on Hot Topics in Networks (HotNets IV)*, College Park, MD, USA.

[Coulouris et al. 2005] Coulouris, G., Dollimore, J., and Kindberg, T. (2005). *Distributed Systems - Concepts and Design*. Addison-Wesley, 4th edition. 928 pp.

[Coulson et al. 2008] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. (2008). A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1):1–42.

[Coulson et al. 2002] Coulson, G., Blair, G. S., Clarke, M., and Parlavantzas, N. (2002). The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126.

[Courtois et al. 1971] Courtois, P. J., Heymans, F., and Parnas, D. L. (1971). Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668.

[Cowling et al. 2006] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006). HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 177–190, Seattle, WA, USA.

[Cristian 1991] Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78.

[Cugola et al. 2001] Cugola, G., Nitto, E. D., and Fuggetta, A. (2001). The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850.

[Culler et al. 2005] Culler, D., Dutta, P., Ee, C. T., Fonseca, R., Hui, J., Levis, P., Polastre, J., Shenker, S., Stoica, I., Tolle, G., and Zhao, J. (2005). Towards a Sensor Network Architecture: Lowering the Waistline. In *Proceedings of the Tenth International Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, USA.

[Czerwinski et al. 1999] Czerwinski, S. E., Zhao, B. Y., Hodes, T. D., Joseph, A. D., and Katz, R. H. (1999). An Architecture for a Secure Service Discovery Service. In *Proceedings of Mobile Computing and Networking (MobiCom '99)*, pages 24–35, Seattle, WA.

[Dahl et al. 1968] Dahl, O., Myhrhaug, B., and Nygaard, K. (1968). Simula 67 - Common Base Language. Norwegian Computing Center, Forskningveien 1B, Oslo 3, Norway.

[Dahl et al. 1970] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1970). The SIMULA 67 common base language. Technical Report S-22, Norwegian Computing Center, Oslo, Norway.

[DASADA] DASADA. Dynamic Assembly for System Adaptability, Dependability, and Assurance. http://www.schafercorp-ballston.com/dasada/index2.html.

[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.

[Dashofy et al. 2002] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. (2002). An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 266–276, Orlando, FL, USA.

[Dashofy et al. 2005] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. (2005). A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology*, 14(2):199–245.

[Dearle et al. 2004] Dearle, A., Kirby, G. N. C., McCarthy, A., and Diaz y Carballo, J. C. (2004). A Flexible and Secure Deployment Framework for Distributed Applications. In Emmerich, W. and Wolf, A. L., editors, *Component Deployment: Second International Working Conference, CD 2004, Edinburgh, UK, May 20-21, 2004*, volume 3083 of *Lecture Notes in Computer Science*, pages 219–233. Springer.

[Défago et al. 2004] Défago, X., Schiper, A., and Urbán, P. (2004). Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(4):372–421.

[Demers et al. 1987] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing (PODC '87)*, pages 1–12, New York, NY, USA. ACM.

[Dennis and Van Horn 1966] Dennis, J. B. and Van Horn, E. C. (1966). Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155.

[DeRemer and Kron 1976] DeRemer, F. and Kron, H. H. (1976). Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86.

[Di Cosmo et al. 2006] Di Cosmo, R., Durak, B., Leroy, X., Mancinelli, F., and Vouillon, J. (2006). Maintaining large software distributions: new challenges from the FOSS era. In *Proceedings of the FRCSS 2006 Workshop*, volume 12 of *EASST Newsletter*, pages 7–20.

[Diao et al. 2002a] Diao, Y., Gandhi, N., Hellerstein, J. L., Parekh, S., and Tilbury, D. M. (2002a). MIMO Control of an Apache Web Server: Modeling and Controller Design. In *Proceeedings of the American Control Conference*, volume 6, pages 4922–4927.

[Diao et al. 2002b] Diao, Y., Gandhi, N., Hellerstein, J. L., Parekh, S., and Tilbury, D. M. (2002b). Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In *IEEE/IFIP Network Operations and Management Symposium (NOMS'02)*, pages 219–234, Florence, Italy.

[Diao et al. 2005] Diao, Y., Hellerstein, J. L., Parekh, S., Griffith, R., Kaiser, G., and Phung, D. (2005). Self-Managing Systems: A Control Theory Foundation. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 441–448, Greenbelt, MD, USA.

[Diao et al. 2006] Diao, Y., Hellerstein, J. L., Parekh, S., Shaikh, H., and Surendra, M. (2006). Controlling Quality of Service in Multi-Tier Web Applications. In *26st International Conference on Distributed Computing Systems (ICDCS'06)*, pages 25–32, Lisboa, Portugal.

[Dijkstra 1968] Dijkstra, E. W. (1968). The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346.

[Dolstra 2005] Dolstra, E. (2005). Efficient upgrading in a purely functional component deployment model. In *Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005)*, volume 3489 of *Lecture Notes in Computer Science*, pages 219–234, St. Louis, Missouri, USA. Springer.

[Dolstra et al. 2004a] Dolstra, E., de Jonge, M., and Visser, E. (2004a). Nix: A safe and policy-free system for software deployment. In Damon, L., editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA. USENIX.

[Dolstra et al. 2004b] Dolstra, E., Visser, E., and de Jonge, M. (2004b). Imposing a memory management discipline on software deployment. In Estublier, J. and Rosenblum, D., editors, *26th International Conference on Software Engineering (ICSE'04)*, pages 583–592, Edinburgh, Scotland. IEEE Computer Society.

[Doyle et al. 2003] Doyle, R. P., Chase, J. S., Asad, O. M., Jin, W., and Vahdat, A. M. (2003). Model-based Resource Provisioning in Web Service Utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS-03)*, Seattle, WA, USA.

[Dumant et al. 1998] Dumant, B., Horn, F., Tran, F. D., and Stefani, J.-B. (1998). Jonathan: an Open Distributed Processing Environment in Java. In Davies, R., Raymond, K., and Seitz, J., editors, *Proceedings of Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 175–190, The Lake District, UK. Springer.

[Dwork et al. 1988] Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323.

[Edwards 2006] Edwards, W. K. (2006). Discovery systems in ubiquitous computing. *IEEE Pervasive Computing*, 5(2):70–77.

[Ee et al. 2006] Ee, C. T., Fonseca, R., Kim, S., Moon, D., Tavakoli, A., Culler, D., Shenker, S., and Stoica, I. (2006). A modular network layer for sensornets. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 249–262, Seattle, WA, USA.

[EJB] EJB. Enterprise JavaBeans Technology. Sun Microsystems. http://java.sun.com/products/ejb/.

[Elmagarmid 1992] Elmagarmid, A. K., editor (1992). *Database transaction models for advanced applications*. Morgan Kaufmann. 611 p.

[Elnikety et al. 2004] Elnikety, S., Nahum, E., Tracey, J., and Zwaenepoel, W. (2004). A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th International Conference on World Wide Web (WWW'2004)*, pages 276–286, New York, NY, USA.

[Elnozahy et al. 2002] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408.

[Eppinger et al. 1991] Eppinger, J. L., Mummert, L. B., and Spector, A. Z. (1991). *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann.

[Erven et al. 2007] Erven, H., Hicker, G., Huemer, C., and Zapletal, M. (2007). The Web Services-BusinessActivity-Initiator (WS-BA-I) Protocol: an Extension to the Web Services-BusinessActivity Specification. In *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS 2007)*, pages 216–224. IEEE Computer Society.

[Eswaran et al. 1976] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633.

[Eugster et al. 2003] Eugster, P. Th., Felber, P., Guerraoui, R., and Kermarrec, A.-M. (2003). The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131.

[Eugster et al. 2000] Eugster, P. Th., Guerraoui, R., and Sventek, J. (2000). Type-Based Publish/Subscribe. Tech. Report DSC ID 2000-029, École Polytechnique Fédérale de Lausanne, Laboratoire de Programmation Distribuée.

[Excalibur] Excalibur. The Apache Excalibur Project. http://excalibur.apache.org.

[Fabry 1974] Fabry, R. S. (1974). Capability-based addressing. *Communications of the ACM*, 17(7):403–412.

[Fassino et al. 2002] Fassino, J.-Ph., Stefani, J.-B., Lawall, J., and Muller, G. (2002). THINK: A software framework for component-based operating system kernels. In *Proceedings of Usenix Annual Technical Conference*, Monterey (USA).

[Fekete et al. 2005] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. (2005). Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528.

[Feldman et al. 2005] Feldman, M., Lai, K., and Zhang, L. (2005). A price-anticipating resource allocation mechanism for distributed shared clusters. In *Proceedings of the 6th ACM Conference on Electronic Commerce (EC'05)*, pages 127–136, New York, NY, USA. ACM Press.

[Felix 2007] Felix (2007). The Apache Felix Project. http://felix.apache.org/.

[Fischer et al. 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

[Fleury and Reverbel 2003] Fleury, M. and Reverbel, F. (2003). The JBoss extensible server. In *Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03)*, volume 2672 of *Lecture Notes in Computer Science*, pages 344–373. Springer-Verlag.

[Flissi and Merle 2004] Flissi, A. and Merle, Ph. (2004). Vers un environnement multi-personnalités pour la configuration et le déploiement d'applications à base de composants logiciels. In *DECOR'2004, 1ère Conférence Francophone sur le Déploiement et la (Re)Configuration de Logiciels*, pages 3–14, Grenoble.

[Flissi and Merle 2005] Flissi, A. and Merle, Ph. (2005). Une démarche dirigée par les modèles pour construire les machines de déploiement des intergiciels à composants. *L'Objet*, 11(1-2):79–94. Hermès.

[Ford et al. 1997] Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O. (1997). The Flux OSKit: a substrate for kernel and language research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 38–51, New York, NY, USA. ACM.

[Fowler 2004] Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection Pattern. http://www.martinfowler.com/articles/injection.html.

[Fowler 1986] Fowler, R. J. (1986). The complexity of using forwarding address for decentralized object finding. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 108–120.

[Fox et al. 1998] Fox, A., Gribble, S. D., Chawathe, Y., and Brewer, E. A. (1998). Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications*, pages 10–19.

[Fractal] Fractal. The Fractal Project. http://fractal.objectweb.org/.

[Franklin 2004] Franklin, M. J. (2004). Concurrency control and recovery. In Tucker, A. J., editor, *Computer Science Handbook*, chapter 56. Chapman and Hall/CRC. 2nd ed., 2752 pp.

[Fraser 1971] Fraser, A. G. (1971). On the meaning of names in programming systems. *Communications of the ACM*, 14(6):409–416.

[Friedman et al. 1997] Friedman, N., Geiger, D., and Goldszmidt, M. (1997). Bayesian network classifiers. *Machine Learning*, 29(13):131–163.

[Fu et al. 2003] Fu, Y., Chase, J., Chun, B., Schwab, S., and Vahdat, A. (2003). SHARP: an architecture for secure resource peering. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 133–148. ACM Press.

[Gabber et al. 1999] Gabber, E., Small, C., Bruno, J., Brustoloni, J., and Silberschatz, A. (1999). The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 267–281, Berkeley, CA, USA. USENIX Association.

[Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 416 pp.

[Ganek and Corbi 2003] Ganek, A. G. and Corbi, T. A. (2003). The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–17. Special issue on autonomic computing.

[García-Molina and Salem 1987] García-Molina, H. and Salem, K. (1987). Sagas. In Dayal, U. and Traiger, I. L., editors, *SIGMOD'87: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259. ACM Press.

[Garlan et al. 2000] Garlan, D., Monroe, R. T., and Wile, D. (2000). Acme: Architectural Description of Component-Based Systems. In Leavens, G. T. and Sitamaran, M., editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press.

[Garlan et al. 2001] Garlan, D., Schmerl, B., and Chang, J. (2001). Using Gauges for Architecture-Based Monitoring and Adaptation. In *Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia.

[Gelernter 1985] Gelernter, D. (1985). Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112.

[Gelernter and Carriero 1992] Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107.

[Georgiadis et al. 2002] Georgiadis, I., Magee, J., and Kramer, J. (2002). Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA. ACM Press.

[Gifford 1979] Gifford, D. K. (1979). Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP'79)*, pages 150–162.

[Goldberg and Robson 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

[Goldsack et al. 2003] Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., and Toft, P. (2003). SmartFrog: Configuration and Automatic Ignition of Distributed Applications. HP OVUA 2003 - HP OpenView University Association.

[GoTM] GoTM. The GoTM Project. http://gotm.objectweb.org/.

[Govil et al. 2000] Govil, K., Teodosiu, D., Huang, Y., and Rosenblum, M. (2000). Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262.

[Grace et al. 2006] Grace, P., Coulson, G., Blair, G. S., and Porter, B. (2006). A distributed architecture meta-model for self-managed middleware. In *Proceedings of the 5th Workshop on Adaptive and Reflective Middleware (ARM '06)*, Melbourne, Australia.

[Gray 1978] Gray, J. (1978). Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK. Springer-Verlag.

[Gray 1981] Gray, J. (1981). The transaction concepts: Virtues and limitations. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*, pages 144–154. Cannes, France, September 9–11 (invited paper).

[Gray 1986] Gray, J. (1986). Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12.

[Gray et al. 1996] Gray, J., Helland, P., O'Neil, P., and Shasha, D. (1996). The dangers of replication and a solution. In *SIGMOD'96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182.

[Gray and Lamport 2006] Gray, J. and Lamport, L. (2006). Consensus on transaction commit. *ACM Transactions on Database Systems*, 31:133–160. ACM Press.

[Gray et al. 1981] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I. (1981). The recovery manager of the System-R database manager. *ACM Computing Surveys*, 13(2):223–242.

[Gray and Reuter 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann. 1070 pp.

[Grimes 1997] Grimes, R. (1997). *Professional DCOM Programming*. Wrox Press. 592 pp.

[Gruber et al. 2005] Gruber, O., Hargrave, B. J., McAffer, J., Rapicault, P., and Watson, T. (2005). The Eclipse 3.0 platform: Adopting OSGi technology. *IBM Systems Journal*, 44(2):289–300.

[Gryphon 2003] Gryphon (2003). The Gryphon System. http://www.research.ibm.com/gryphon/.

[Guerraoui 1995] Guerraoui, R. (1995). Revistiting the relationship between non-blocking atomic commitment and consensus. In *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 87–100, London, UK. Springer-Verlag.

[Guerraoui et al. 2008] Guerraoui, R., Quéma, V., and Vukolić, M. (2008). The next 700 BFT protocols. Technical Report LPD-2008-08, École Polytechnique Fédérale de Lausanne, School of Computer and Communication Sciences, Lausanne, Switzerland.

[Guerraoui and Schiper 1995] Guerraoui, R. and Schiper, A. (1995). The decentralized non-blocking atomic commitment protocol. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Systems*, pages 2–9.

[Guerraoui and Schiper 1997] Guerraoui, R. and Schiper, A. (1997). Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74.

[Guerraoui and Schiper 2001] Guerraoui, R. and Schiper, A. (2001). The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41.

[Guttman 1999] Guttman, E. (1999). Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, 3(4):71–80.

[Hadzilacos and Toueg 1993] Hadzilacos, V. and Toueg, S. (1993). Fault-Tolerant Broadcasts and Related Problems. In Mullender, S., editor, *Distributed Systems*, pages 97–168. Addison-Wesley.

[Haerder and Reuter 1983] Haerder, T. and Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317.

[Hagimont and De Palma 2002] Hagimont, D. and De Palma, N. (2002). Removing Indirection Objects for Non-functional Properties. In *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*.

[Hall 2004] Hall, R. S. (2004). A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks. In Emmerich, W. and Wolf, A. L., editors, *Component Deployment: Second International Working Conference, CD 2004, Edinburgh, UK, May 20-21, 2004*, volume 3083 of *Lecture Notes in Computer Science*, pages 81–96. Springer.

[Hall et al. 1999] Hall, R. S., Heimbigner, D., and Wolf, A. L. (1999). A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 174–183, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Hamilton et al. 1993] Hamilton, G., Powell, M. L., and Mitchell, J. G. (1993). Subcontract: A flexible base for distributed programming. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, volume 27 of *Operating Systems Review*, pages 69–79, Asheville, NC (USA).

[Hayton et al. 1998] Hayton, R., Herbert, A., and Donaldson, D. (1998). Flexinet: a flexible, component oriented middleware system. In *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal.

[Heineman and Councill 2001] Heineman, G. T. and Councill, W. T., editors (2001). *Component-Based Software Engineering*. Addison-Wesley.

[Hellerstein 2004] Hellerstein, J. L. (2004). Challenges in Control Engineering of Computer Systems. In *Proceedings of the American Control Conference*, pages 1970–1979, Boston, Mass, USA.

[Hellerstein et al. 2004] Hellerstein, J. L., Diao, Y., Tilbury, D. M., and Parekh, S. (2004). *Feedback Control of Computing Systems*. John Wiley and Sons. 429 pp.

[Herlihy and Wing 1990] Herlihy, M. P. and Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492.

[Hewitt 1977] Hewitt, C. (1977). Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, 8(3):323–364.

[Hoare 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585.

[Hutchinson and Peterson 1991] Hutchinson, N. C. and Peterson, L. L. (1991). The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76.

[IBM 2003] IBM (2003). An architectural blueprint for autonomic computing. IBM White Papers on Autonomic Computing. http://www-3.ibm.com/autonomic/pdfs/ACwpFinal.pdf.

[Ichbiah et al. 1986] Ichbiah, J. D., Barnes, J. G., Firth, R. J., and Woodger, M. (1986). Rationale for the Design of the Ada® Programming Language. U.S. Government, Ada Joint Program Office.
http://archive.adaic.com/standards/83rat/html/.

[Irwin et al. 2005] Irwin, D., Chase, J., Grit, L., and Yumerefendi, A. (2005). Self-recharging virtual currency. In *P2PECON '05: Proceedings of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems*, pages 93–98, New York, NY, USA. ACM Press.

[J2EE] J2EE. The Java2 Platform, Enterprise Edition. Sun Microsystems.
http://java.sun.com/j2ee.

[Jackson et al. 2000] Jackson, D., Schechter, I., and Shlyahter, H. (2000). Alcoa: the ALLOY constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 730–733, New York, NY, USA. ACM Press.

[Jacob et al. 2004] Jacob, B., Lanyon-Hogg, R., Nadgir, D. K., and Yassin, A. F. (2004). A practical guide to the IBM autonomic toolkit. Technical report, IBM International Technical Support Organization. http://www.redbooks.ibm.com/redbooks/SG246635.

[Jajodia and Kerschberg 1997] Jajodia, S. and Kerschberg, L., editors (1997). *Advanced Transaction Models and Architectures*. Kluwer. 400 p.

[Jannotti et al. 2000] Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., and O'Toole, Jr., J. W. (2000). Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 197–212, San Diego, CA, USA.

[JEE] JEE. Java Platform, Enterprise Edition. Sun Microsystems. http://java.sun.com/javaee.

[Jiménez-Peris et al. 2003] Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G., and Kemme, B. (2003). Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294.

[Johnson 1997] Johnson, R. E. (1997). Frameworks=(Components+Patterns): How frameworks compare to other object-oriented reuse techniques. *Communications of the ACM*, 40(10):39–42.

[Jonathan 2002] Jonathan (2002). The Jonathan Tutorial, by Sacha Krakowiak. http://jonathan.objectweb.org/doc/tutorial/index.html.

[Jones 1993] Jones, M. B. (1993). Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, Asheville, NC (USA).

[Jul et al. 1988] Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133.

[Kamra et al. 2004] Kamra, A., Misra, V., and Nahum, E. (2004). Yaksha: A Self-Tuning Controller for Managing the Performance of 3-Tiered Web Sites. In *International Workshop on Quality of Service (IWQoS)*.

[Kandula 2007] Kandula (2007). The Apacha Kandula project. http://ws.apache.org/kandula/.

[Kephart 2005] Kephart, J. O. (2005). Research Challenges of Autonomic Computing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 15–22, New York, NY, USA. ACM Press.

[Kephart and Chess 2003] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, 36(1):41–50.

[Kermarrec and van Steen 2007a] Kermarrec, A.-M. and van Steen, M., editors (2007a). *Gossip-Based Computer Networking*, volume 41(5), Special Topic of *ACM Operating Systems Review*.

[Kermarrec and van Steen 2007b] Kermarrec, A.-M. and van Steen, M. (2007b). Gossiping in distributed systems. *ACM SIGOPS Operating Systems Review, Special Issue on Gossip-Based Networking*, pages 2–7.

[Kiczales 1996] Kiczales, G. (1996). Aspect-Oriented Programming. *ACM Computing Surveys*, 28(4):154.

[Kiczales et al. 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press. 345 pp.

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355, Budapest, Hungary. Springer-Verlag.

[Knopflerfish 2007] Knopflerfish (2007). OSGi Knopflerfish. http://www.knopflerfish.org.

[Knuth 1992] Knuth, D. E. (1992). *Literate Programming.* Center for the Study of Language and Information, Stanford University - Lecture Notes, No 27. 368 pp.

[Kohler et al. 2000] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297.

[Kon et al. 2002] Kon, F., Costa, F., Blair, G., and Campbell, R. (2002). The case for reflective middleware. *Communications of the ACM*, 45(6):33–38.

[Kotla et al. 2007] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2007). Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, pages 45–58. *Operating Systems Review*, 41(6), December 2007.

[Kramer and Magee 1990] Kramer, J. and Magee, J. (1990). The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306.

[Kramer 1998] Kramer, R. (1998). iContract - The Java Design by Contract Tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS) Conference*, pages 295–307.

[Kurose and Ross 2004] Kurose, J. F. and Ross, K. W. (2004). *Computer Networking: A Top-Down Approach Featuring the Internet, 2nd ed.* Addison-Wesley.

[Kuz et al. 2007] Kuz, I., Liu, Y., Gorton, I., and Heiser, G. (2007). CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5):687–699.

[Lagaisse and Joosen 2006] Lagaisse, B. and Joosen, W. (2006). True and Transparent Distributed Composition of Aspect-Components. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'06)*, volume 4290 of *Lecture Notes in Computer Science*, pages 42–61, Melbourne, Australia. Springer-Verlag.

[Lamport 1978a] Lamport, L. (1978a). The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 2:95–114.

[Lamport 1978b] Lamport, L. (1978b). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–56.

[Lamport 1979] Lamport, L. (1979). How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 9(C-28):690–691.

[Lamport 1998] Lamport, L. (1998). The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169. First appeared as DEC-SRC Research Report 49, 1989.

[Lamport et al. 1982] Lamport, L., Shostak, R. E., and Pease, M. C. (1982). The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

[Lampson 2001] Lampson, B. (2001). The ABCDs of Paxos. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*.

[Lampson 1986] Lampson, B. W. (1986). Designing a Global Name Service. *Proceedings of the 5th. ACM Symposium on the Principles of Distributed Computing*, pages 1–10.

[Lampson and Sturgis 1979] Lampson, B. W. and Sturgis, H. E. (1979). Crash recovery in a distributed data storage system. Technical report, Xerox PARC (unpublished), 25 pp. Partially reproduced in: *Distributed Systems – Architecture and Implementation*, ed. Lampson, Paul, and Siegert, Lecture Notes in Computer Science 105, Springer, 1981, pp. 246–265 and pp. 357–370.

[Laprie 1985] Laprie, J.-C. (1985). Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Proceedings of the 15th IEEE International Symposium on Faul-Tolerant Computing (FTCS-15)*, pages 2–11.

[Larrea et al. 2004] Larrea, M., Fernández, A., and Arévalo, S. (2004). On the Implementation of Unreliable Failure Detectors in Partially Synchronous Systems. *IEEE Transactions on Computers*, 53(7):815–828.

[Larrea et al. 2007] Larrea, M., Lafuente, A., Soraluze, I., and nas, R. C. (2007). Designing Efficient Algorithms for the Eventually Perfect Failure Detector Class. *Journal of Software*, 2(4):1–11.

[Larus and Rajwar 2007] Larus, J. R. and Rajwar, R. (2007). *Transactional Memory.* Morgan & Claypool. 211 pp.

[Lau et al. 2006] Lau, K.-K., Ornaghi, M., and Wang, Z. (2006). A Software Component Model and Its Preliminary Formalisation. In de Boer, F. S., Bonsangue, M. M., Graf, S., and de Roever, W.-P., editors, *Proceedings of Fourth International Symposium on Formal Methods for Components and Objects*, number 4111 in Lecture Notes in Computer Science, pages 1–21. Springer-Verlag.

[Lau and Wang 2007] Lau, K.-K. and Wang, Z. (2007). Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724.

[Lauer and Needham 1979] Lauer, H. C. and Needham, R. M. (1979). On the Duality of Operating Systems Structures. In *Proc. Second International Symposium on Operating Systems,* IRIA, Oct. 1978, volume 13 of *ACM Operating Systems Review*, pages 3–19.

[Laumay et al. 2001] Laumay, Ph., Bruneton, É., De Palma, N., and Krakowiak, S. (2001). Preserving Causality in a Scalable Message-Oriented Middleware. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg.

[Layaida and Hagimont 2005] Layaida, O. and Hagimont, D. (2005). Designing Self-adaptive Multimedia Applications through Hierarchical Reconfiguration. In Kutvonen, L. and Alonistioti, N., editors, *Proc. DAIS 2005*, volume 3543 of *Lecture Notes in Computer Science*, pages 95–107. Springer.

[Lazowska et al. 1984] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. (1984). *Quantitative System Performance : Computer system analysis using queueing network models.* Prentice Hall. 417 pp., available on-line at http://www.cs.washington.edu/homes/lazowska/qsp/.

[LDAP 2006] LDAP (2006). Lightweight Directory Access Protocol: Technical Specification Road Map. RFC 4510, The Internet Society. http://tools.ietf.org/html/rfc4510.

[Lea 1999] Lea, D. (1999). *Concurrent Programming in Java.* The Java Series. Addison-Wesley, 2nd edition. 412 pp.

[Lea et al. 1993] Lea, R., Jacquemot, C., and Pillevesse, E. (1993). COOL: System Support for Distributed Object-oriented Programming. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):37–47.

[Leavens and Sitaraman 2000] Leavens, G. and Sitaraman, M., editors (2000). *Foundations of Component Based Systems.* Cambridge University Press.

[Leclercq et al. 2007] Leclercq, M., Özcan, A. E., Quéma, V., and Stefani, J.-B. (2007). Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 209–219, Minneapolis, MN, USA.

[Leclercq et al. 2005] Leclercq, M., Quéma, V., and Stefani, J.-B. (2005). DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9).

[Leiner et al. 2003] Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., Postel, J., Roberts, L. G., and Wolff, S. (2003). A Brief History of the Internet. The Internet Society. http://www.isoc.org/internet/history/brief.shtml.

[Lendenmann 1996] Lendenmann, R. (1996). *Understanding OSF DCE 1.1 for AIX and OS/2*. Prentice Hall. 312 pp.

[Leveson and Turner 1993] Leveson, N. and Turner, C. S. (1993). An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41.

[Levin et al. 1975] Levin, R., Cohen, E. S., Corwin, W. M., Pollack, F. J., and Wulf, W. A. (1975). Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 132–140.

[Lindholm and Yellin 1996] Lindholm, T. and Yellin, F. (1996). *The Java Virtual Machine Specification*. Addison-Wesley. 475 pp.

[Lions et al. 1996] Lions, J.-L. et al. (1996). Ariane 5 Flight 501 Failure Report by the Inquiry Board. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[LISA] LISA. Large Installation Systems Administration Conferences (Usenix and Sage). http://www.usenix.org/events/byname/lisa.html.

[Liskov 1988] Liskov, B. (1988). Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312.

[Loo et al. 2005] Loo, B., Condie, T., Hellerstein, J., Maniatis, P., Roscoe, T., and Stoica, I. (2005). Implementing declarative overlays. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP'05)*, pages 75–90, Brighton, UK.

[Lu et al. 2002] Lu, C., Stankovic, J. A., Tao, G., and Son, S. H. (2002). Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. *Real Time Systems Journal*, 23(1/2):85–126.

[Luckham and Vera 1995] Luckham, D. C. and Vera, J. (1995). An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734.

[Lynch 1989] Lynch, N. (1989). A hundred impossibility proofs for distributed computing. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–28, New York, NY. ACM Press.

[Lyon 1990] Lyon, J. (1990). Tandem's remote data facility. In *Proceedings of IEEE Spring CompCon*, pages 562–567.

[Maes 1987] Maes, P. (1987). Concepts and Experiments in Computational Reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 147–155, Orlando, Florida, USA.

[Magee et al. 1995] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying Distributed Software Architectures. In Schafer, W. and Botella, P., editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain. Springer-Verlag, Berlin.

[Magee et al. 1989] Magee, J., Kramer, J., and Sloman, M. (1989). Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675.

[Martin-Flatin et al. 1999] Martin-Flatin, J.-Ph., Znaty, S., and Hubaux, J.-P. (1999). A Survey of Distributed Enterprise Network and Systems Management Paradigms. *Journal of Network and Systems Management*, 7(1):9–26.

[Marzullo and Wiebe 1987] Marzullo, K. and Wiebe, D. (1987). Jasmine: a software system modelling facility. In *Proceedings of the second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE-2)*, pages 121–130, New York, NY, USA. ACM.

[Massie et al. 2004] Massie, M. L., Chun, B. N., and Culler, D. E. (2004). The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840.

[McIlroy 1968] McIlroy, M. (1968). Mass produced software components. In Naur, P. and Randell, B., editors, *Software Engineering: A Report On a Conference Sponsored by the NATO Science Committee*, pages 138–155, Garmisch, Germany.

[McQuillan 1978] McQuillan, J. M. (1978). Enhanced message addressing capabilities for computer networks. *Proceeedings of the IEEE*, 66(11):1517–1526.

[Medvidovic et al. 2007] Medvidovic, N., Dashofy, E. M., and Taylor, R. N. (2007). Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49:12–31.

[Medvidovic and Taylor 2000] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93.

[Mehta et al. 2000] Mehta, N. R., Medvidovic, N., and Phadke, S. (2000). Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 178–187. ACM Press.

[Menascé and Almeida 2001] Menascé, D. A. and Almeida, V. A. F. (2001). *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall. 608 pp.

[Meyer 1992] Meyer, B. (1992). Applying Design by Contract. *IEEE Computer*, 25(10):40–52.

[Middleware 1998] Middleware (1998). IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing. September 15-18 1998, The Lake District, England.

[Miranda et al. 2001] Miranda, H., Pinto, A., and Rodrigues, L. (2001). Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Proc. 21st International conference on Distributed Computing Systems (ICDCS'01)*, pages 707–710, Phoenix, Arizona, USA. IEEE Computer Society.

[Mitchell et al. 1994] Mitchell, J. G., Gibbons, J., Hamilton, G., Kessler, P. B., Khalidi, Y. Y. A., Kougiouris, P., Madany, P., Nelson, M. N., Powell, M. L., and Radia, S. R. (1994). An overview of the Spring system. In *Proceedings of COMPCON*, pages 122–131.

[Mitchell et al. 1978] Mitchell, J. G., Maybury, W., and Sweet, R. (1978). Mesa Language Manual. Technical Report CSL-78-1, Xerox Research Center, Palo Alto, CA, USA.

[Mockapetris and Dunlap 1988] Mockapetris, P. and Dunlap, K. J. (1988). Development of the Domain Name System. In *SIGCOMM '88: Symposium Proceedings on Communications Architectures and Protocols*, pages 123–133, New York, NY, USA. ACM Press.

[Mohan et al. 1992] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992). Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162.

[Mohan et al. 1986] Mohan, C., Lindsay, B., and Obermarck, R. (1986). Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396.

[Monson-Haefel 2002] Monson-Haefel, R. (2002). *Enterprise JavaBeans*. O'Reilly & Associates, Inc., 3rd edition. 550 pp.

[Moore et al. 2002] Moore, J., Irwin, D., Grit, L., Sprenkle, S., , and Chase, J. (2002). Managing mixed-use clusters with cluster-on-demand. Technical report, Department of Computer Science, Duke University. http://issg.cs.duke.edu/cod-arch.pdf.

[Morrison et al. 2004] Morrison, R., Kirby, G. N. C., Balasubramaniam, D., Mickan, K., Oquendo, F., Cîmpan, S., Warboys, B. C., Snowdon, B., and Greenwood, R. (2004). Support for Evolving Software Architectures in the ArchWare ADL. In *4th IEEE/IFIP Working Conference on Software Architecture (WICSA 4)*, pages 69–78, Oslo, Norway.

[Mosberger and Peterson 1996] Mosberger, D. and Peterson, L. L. (1996). Making paths explicit in the Scout operating system. In *Operating Systems Design and Implementation (OSDI'96)*, pages 153–167.

[Moss 1985] Moss, J. E. B. (1985). *Nested transactions: an approach to reliable distributed computing*. Massachusetts Institute of Technology, Cambridge, MA, USA. Based on the author's PhD thesis, 1981. 160 p.

[Mostefaoui et al. 2003] Mostefaoui, A., Mourgaya, E., and Raynal, M. (2003). Asynchronous Implementation of Failure Detectors. In *International Conference on Dependable Systems and Networks (DSN'03)*, pages 35–360, Los Alamitos, CA, USA. IEEE Computer Society.

[Mullender et al. 1990] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. (1990). Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53.

[Naur and Randell 1969] Naur, P. and Randell, B., editors (1969). *Software Engineering: A Report On a Conference Sponsored by the NATO Science Committee, 7-11 Oct. 1968*. Scientific Affairs Division, NATO. 231 pp.

[Needham 1993] Needham, R. M. (1993). Names. In Mullender, S., editor, *Distributed Systems*, chapter 12, pages 315–327. ACM Press Books, Addison-Wesley.

[.NET] .NET. Microsoft Corp. http://www.microsoft.com/net.

[OASIS] OASIS. Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org/.

[OASIS WS-TX TC 2007a] OASIS WS-TX TC (2007a). Web Services Atomic Transaction Version 1.1. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[OASIS WS-TX TC 2007b] OASIS WS-TX TC (2007b). Web Services Business Activity Version 1.1. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[OASIS WS-TX TC 2007c] OASIS WS-TX TC (2007c). Web Services Coordination Version 1.1. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[Obermarck 1982] Obermarck, R. (1982). Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208.

[ODMG] ODMG. The Object Data Management Group. http://www.odmg.org.

[ODP 1995a] ODP (1995a). ITU-T & ISO/IEC, Recommendation X.902 & International Standard 10746-2: "ODP Reference Model: Foundations".
http://archive.dstc.edu.au/AU/research_news/odp/ref_model/standards.html.

[ODP 1995b] ODP (1995b). ITU-T & ISO/IEC, Recommendation X.903 & International Standard 10746-3: "ODP Reference Model: Architecture".
http://archive.dstc.edu.au/AU/research_news/odp/ref_model/standards.html.

[Oki et al. 1993] Oki, B., Pfluegl, M., Siegel, A., and Skeen, D. (1993). The Information Bus – An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, volume 27 of *Operating Systems Review*, pages 58–68, Asheville, NC (USA).

[Oki and Liskov 1988] Oki, B. M. and Liskov, B. H. (1988). Viewstamped Replication: A New Primary Copy Method to Support. Highly-Available Distributed Systems. In *Proceedings of the Seventh annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 8–17, New York, NY, USA. ACM.

[OMG] OMG. The Object Management Group. http://www.omg.org.

[OMG 2003] OMG (2003). CORBA/IIOP Specification. Object Management Group. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[OMG C&D] OMG C&D. Deployment and Configuration of Component-based Distributed Applications Specification. Object Management Group Specification, document ptc/05-01-07. http://www.omg.org/technology/documents/corba_spec_catalog.htm.

[OMG MDA] OMG MDA. Object Management Group Model Driven Architecture. http://www.omg.org/mda.

[Open Group] Open Group. http://www.opengroup.org/.

[Oppen and Dalal 1983] Oppen, D. C. and Dalal, Y. K. (1983). The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. *ACM Transactions on Information Systems*, 1(3):230–253.

[Oppenheimer et al. 2003] Oppenheimer, D., Ganapathi, A., and Patterson, D. A. (2003). Why do Internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*.

[Oscar 2005] Oscar (2005). The Oscar Project. http://forge.objectweb.org/projects/oscar/.

[OSGI Alliance 2005] OSGI Alliance (2005). OSGi Service Platform Release 4. http://www.osgi.org.

[Pai et al. 1998] Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. M. (1998). Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 205–216.

[Papadimitriou 1979] Papadimitriou, C. H. (1979). The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653.

[Papadimitriou 1986] Papadimitriou, C. H. (1986). *The Theory of Database Concurrency Control*. Computer Science Press.

[Parnas 1972] Parnas, D. L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058.

[Parrington et al. 1995] Parrington, G. D., Shrivastava, S. K., Wheater, S. M., and Little, M. C. (1995). The design and implementation of Arjuna. *Computing Systems*, 8(2):255–308.

[Parrish et al. 2001] Parrish, A., Dixon, B., and Cordes, D. (2001). A conceptual foundation for component-based software deployment. *The Journal of Systems and Software*, 57:193–200.

[Patterson et al. 1988] Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (raid). In *SIGMOD'88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, New York, NY, USA. ACM.

[Pawlak et al. 2001] Pawlak, R., Duchien, L., Florin, G., and Seinturier, L. (2001). JAC : a flexible solution for aspect oriented programming in Java. In Yonezawa, A. and Matsuoka, S., editors, *Proceedings of Reflection 2001, the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24, Kyoto, Japan. Springer-Verlag.

[Pedone et al. 2003] Pedone, F., Guerraoui, R., and Schiper, A. (2003). The Database State Machine Approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1):71–98.

[Perkins 1998] Perkins, C. E. (1998). Mobile networking through Mobile IP. *IEEE Internet Computing*, 2(1):58–69.

[Pessemier et al. 2006a] Pessemier, N., Seinturier, L., Coupaye, T., and Duchien, L. (2006a). A Model for Developing Component-based and Aspect-oriented Systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, volume 4089 of *Lecture Notes in Computer Science*, pages 259–273. Springer-Verlag.

[Pessemier et al. 2006b] Pessemier, N., Seinturier, L., Coupaye, T., and Duchien, L. (2006b). A Safe Aspect-oriented Programming Support for Component-oriented Programming. In *Proc. Eleventh International Workshop on Component-Oriented Programming (WCOP 2006)*, Nantes, France.

[Peterson et al. 2004] Peterson, L., Shenker, S., and Turner, J. (2004). Overcoming the Internet impasse through virtualization. In *Third Workshop on Hot Topics in Networking (HotNets-III)*, San Diego, CA.

[Peterson and Davie 2003] Peterson, L. L. and Davie, B. S. (2003). *Computer Networks – a Systems Approach*. Morgan Kaufmann, 3rd edition. 815 pp.

[PicoContainer] PicoContainer. PicoContainer. http://www.picocontainer.org.

[Pike et al. 1995] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. (1995). Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254.

[Platt 1999] Platt, D. S. (1999). *Understanding COM+*. Microsoft Press. 256 pp.

[Plattner and Alonso 2004] Plattner, C. and Alonso, G. (2004). Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware (Middleware'04)*, pages 155–174, Toronto, Canada. Springer-Verlag New York, Inc.

[PLoP] PLoP. The Pattern Languages of Programs (PLoP) Conference Series. http://www.hillside.net/conferences/plop.htm.

[Polastre et al. 2005] Polastre, J., Hui, J., Levis, P., Zhao, J., Culler, D., Shenker, S., and Stoica, I. (2005). A unifying link abstraction for wireless sensor networks. In *Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

[Randell 1975] Randell, B. (1975). System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable software*, pages 437–449, New York, NY, USA. ACM.

[Ratnasamy et al. 2001] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker., S. (2001). A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architecture, and Protocols*, pages 161–172, San Diego, CA, USA.

[Raz 1992] Raz, Y. (1992). The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92)*, pages 292–312, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Reid et al. 2000] Reid, A., Flatt, M., Stoller, L., Lepreau, J., and Eide, E. (2000). Knit: component composition for systems software. In *Proceedings of the 4th Symposium on Operating System Design & Implementation (OSDI'00)*, pages 24–24, Berkeley, CA, USA. USENIX Association.

[Reiser and Lavenberg 1980] Reiser, M. and Lavenberg, S. S. (1980). Mean-value analysis of closed multichain queuing networks. *Journal of the Association for Computing Machinery*, 27(2):313–322.

[Reiss 1990] Reiss, S. P. (1990). Connecting Tools Using Message Passing in the Field Program Development Environment. *IEEE Software*, 7(4):57–66.

[Rellermeyer et al. 2007] Rellermeyer, J. S., Alonso, G., and Roscoe, T. (2007). R-OSGi: Distributed Applications through Software Modularization. In *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference (Middleware 2007)*, Newport Beach, CA.

[RM 2000] RM (2000). *Workshop on Reflective Middleware*. Held in conjunction with Middleware 2000, 7-8 April 2000. http://www.comp.lancs.ac.uk/computing/RM2000/.

[ROC 2005] ROC (2005). The Berkeley/Stanford Recovery-Oriented Computing (ROC) Project. http://roc.cs.berkeley.edu.

[Rosenblum and Wolf 1997] Rosenblum, D. S. and Wolf, A. L. (1997). A Design Framework for Internet-Scale Event Observation and Notification. In Jazayeri, M. and Schauer, H., editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 344–360. Springer–Verlag.

[Rosenkrantz et al. 1978] Rosenkrantz, D. J., Stearns, R. E., and Philip M. Lewis, I. (1978). System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198.

[Roshandel et al. 2004] Roshandel, R., van der Hoek, A., Mikic-Rakic, M., and Medvidovic, N. (2004). Mae—a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology*, 13(2):240–276.

[Rouvoy 2006] Rouvoy, R. (2006). *Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables: application aux services de transactions.* PhD thesis, Université des Sciences et Technologies de Lille. 248 pp.

[Rouvoy and Merle 2003] Rouvoy, R. and Merle, Ph. (2003). Abstraction of transaction demarcation in component-oriented platforms. In *Proceedings of the 4th ACM/IFIP/USENIX Middleware Conference (Middleware 2003)*, pages 305–323. LNCS 2672 (Springer).

[Rouvoy and Merle 2007] Rouvoy, R. and Merle, Ph. (2007). Using microcomponents and design patterns to build evolutionary transaction services. *Electronic Notes in Theoretical Computer Science*, 166:111–125. Proceedings of the ERCIM Working Group on Software Evolution (2006).

[Rouvoy et al. 2006a] Rouvoy, R., Serrano-Alvarado, P., and Merle, Ph. (2006a). A component-based approach to compose transaction standards. In *Software Composition*, pages 114–130. LNCS 4089 (Springer).

[Rouvoy et al. 2006b] Rouvoy, R., Serrano-Alvarado, P., and Merle, Ph. (2006b). Towards context-aware transaction services. In *Proceedings of the 6th IFIP Conference on Distributed Applications and Interoperable Services (DAIS)*, pages 272–288. LNCS 4025 (Springer).

[Rowstron and Druschel 2001] Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany.

[Sadjadi and McKinley 2004] Sadjadi, S. M. and McKinley, P. K. (2004). ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan.

[Saito and Shapiro 2005] Saito, Y. and Shapiro, M. (2005). Optimistic replication. *ACM Computing Surveys*, 37(1):42–81.

[Saltzer 1979] Saltzer, J. H. (1979). Naming and binding of objects. In Hayer, R., Graham, R. M., and Seegmüller, G., editors, *Operating Systems. An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 99–208. Springer-Verlag, Munich, Germany.

[Saltzer et al. 1984] Saltzer, J. H., Reed, D. P., and Clark, D. D. (1984). End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288.

[Sandberg et al. 1985] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. (1985). Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA).

[Schantz et al. 1986] Schantz, R., Thomas, R., and Bono, G. (1986). The architecture of the Cronus distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 250–259. IEEE.

[Schiper 2003] Schiper, A. (2003). Practical impact of group communication theory. In et al., A. S., editor, *Future Directions in Distributed Computing (FuDiCo'02)*, volume 2584 of *Lecture Notes in Computer Science*, pages 1–10. Springer-Verlag.

[Schiper 2006a] Schiper, A. (2006a). Dynamic Group Communication. *Distributed Computing*, 18(5):359–374.

[Schiper 2006b] Schiper, A. (2006b). Group communication: From practice to theory. In *SOFSEM 2006: Theory and Practice of Computer Science*, number 3831 in Lecture Notes in Computer Science, pages 117–136. Springer-Verlag.

[Schiper and Toueg 2006] Schiper, A. and Toueg, S. (2006). From Set Membership to Group Membership: A Separation of Concerns. *IEEE Transactions on Dependable and Secure Computing*, 13(2):2–12.

[Schmidt and Buschmann 2003] Schmidt, D. C. and Buschmann, F. (2003). Patterns, frameworks, and middleware: Their synergistic relationships. In *25th International Conference on Software Engineering*, pages 694–704, Portland, Oregon.

[Schmidt et al. 2000] Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* John Wiley & Sons. 666 pp.

[Schneider 1990] Schneider, F. B. (1990). Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319.

[Schneider 1993a] Schneider, F. B. (1993a). The Primary-Backup Approach. In Mullender, S., editor, *Distributed Systems*, chapter 8, pages 199–216. ACM Press Books, Addison-Wesley.

[Schneider 1993b] Schneider, F. B. (1993b). What Good are Models and What Models are Good? In Mullender, S., editor, *Distributed Systems*, chapter 2, pages 17–26. ACM Press Books, Addison-Wesley.

[Schönwälder et al. 2003] Schönwälder, J., Pras, A., and Martin-Flatin, J.-Ph. (2003). On the Future of Internet Management Technologies. *IEEE Communications Magazine*, 41(10):90–97.

[Schroeder et al. 2006] Schroeder, B., Wierman, A., and Harchol-Balter, M. (2006). Open versus closed: a cautionary tale. In *Third Symposium on Networked Systems Design & Implementation (NSDI'06)*, pages 239–252. USENIX Association.

[Seinturier et al. 2006] Seinturier, L., Pessemier, N., Duchien, L., and Coupaye, T. (2006). A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153. Springer-VerlagXS.

[Serrano-Alvarado et al. 2005] Serrano-Alvarado, P., Rouvoy, R., and Merle, P. (2005). Self-adaptive component-based transaction commit management. In *ARM '05: Proceedings of the 4th Workshop on Reflective and Adaptive Middleware Systems*, New York, NY, USA. ACM.

[Shapiro 1986] Shapiro, M. (1986). Structure and encapsulation in distributed systems: The proxy principle. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA). IEEE.

[Shapiro 1994] Shapiro, M. (1994). A binding protocol for distributed shared objects. In *Proc. Int. Conf. on Distributed Computing Systems*, Poznań (Poland).

[Shapiro et al. 1989] Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M., and Valot, C. (1989). SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–337.

[Shaw and Garlan 1996] Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

[Sheldon et al. 1991] Sheldon, M. A., Gifford, D. K., Jouvelot, P., and O'Toole, Jr., J. W. (1991). Semantic File Systems. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25.

[Skeen 1981] Skeen, D. (1981). Non-blocking commit protocols. In *SIGMOD'81: Proceedings of the 1981 ACM-SIGMOD International Conference on Management of Data*, pages 133–142, Orlando, FL, USA.

[Sloman and Twidle 1994] Sloman, M. and Twidle, K. (1994). Domains: A Framework for Structuring Management Policy. In Sloman, M., editor, *Network and Distributed Systems Management*, Chapter 16, pages 433–453. Addison-Wesley.

[Slony-I] Slony-I. Enterprise-level replication system. http://slony.info/.

[SmartFrog 2003] SmartFrog (2003). SmartFrog: Smart Framework for Object Groups. HP Labs. http://www.hpl.hp.com/research/smartfrog/.

[Smith 1982] Smith, B. C. (1982). *Reflection And Semantics In A Procedural Language*. PhD thesis, Massachusetts Institute of Technology. MIT/LCS/TR-272.

[Smith and Nair 2005] Smith, J. E. and Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann. 638 pp.

[Soules et al. 2003] Soules, C. A. N., Appavoo, J., Hui, K., Da Silva, D., Ganger, G. R., Krieger, O., Stumm, M., Wisniewski, R. W., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. (2003). System Support for Online Reconfiguration. In *Proceedings Usenix Annual Technical Conference*, San Antonio, Texas.

[Spiegel 1998] Spiegel, A. (1998). Objects by value: Evaluating the trade-off. In *Proceedings Int. Conf. on Parallel and Distributed Computing and Networks (PDCN)*, pages 542– 548, Brisbane, Australia. IASTED, ACTA Press.

[Spitznagel and Garlan 2001] Spitznagel, B. and Garlan, D. (2001). A Compositional Approach for Constructing Connectors. In *The 1st Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Amsterdam.

[Spread] Spread. The Spread toolkit. http://www.spread.org/.

[Spring 2006] Spring (2006). The Spring Framework. http://www.springframework.org.

[Stallings 2005] Stallings, W. (2005). *Wireless Communications & Networks*. Prentice Hall, 2nd edition.

[Stearns et al. 1976] Stearns, R. E., Lewis, P. M., and Rosenkrantz, D. J. (1976). Concurrency control in database systems. In *Proceedings of the 17th Symposium on Foundations of Computer Science (FOCS'76)*, pages 19–32, Houston, Texas, USA.

[Stevens et al. 2004] Stevens, W. R., Fenner, B., and Rudoff, A. M. (2004). *Unix Network Programming, vol. 1*. Addison-Wesley.

[Stoica et al. 2003] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M., Dabek, F., and Balakrishnan, H. (2003). Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32.

[Sullivan et al. 1999] Sullivan, D. G., Haas, R., and Seltzer, M. I. (1999). Tickets and Currencies Revisited: Extensions to Multi-Resource Lottery Scheduling. In *Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 148–152.

[Sutherland 1968] Sutherland, I. E. (1968). A futures market in computer time. *Commun. ACM*, 11(6):449–451.

[Svetz et al. 1996] Svetz, K., Randall, N., and Lepage, Y. (1996). *MBone: Multicasting Tomorrows's Internet*. IDG Books Worldwide. Online at http://www.savetz.com/mbone/.

[Szyperski 2002] Szyperski, C. (2002). *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley. 2nd ed., 589 pp.

[Tai and Rouvellou 2000] Tai, S. and Rouvellou, I. (2000). Strategies for integrating messaging and distributed object transactions. In Sventek, J. and Coulson, G., editors, *Middleware 2000, Proceedings IFIP/ACM International Conference on Distributed Systems Platforms*, volume 1795 of *Lecture Notes in Computer Science*, pages 308–330. Springer-Verlag.

[Tanenbaum and van Steen 2006] Tanenbaum, A. S. and van Steen, M. (2006). *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edition. 686 pp.

[Tatsubori et al. 2001] Tatsubori, M., Sasaki, T., Chiba, S., and Itano, K. (2001). A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 236–255. Springer Verlag.

[Thomas 1979] Thomas, R. H. (1979). A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209.

[TPC 2006] TPC (2006). TPC Benchmarks. Transaction Processing Performance Council. http://www.tpc.org/.

[UPnP] UPnP. Universal Plug and Play. The UPnP Forum. http://www.upnp.org.

[Urgaonkar et al. 2007] Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2007). An Analytical Model for Multi-tier Internet Services and its Applications. *ACM Transactions on the Web*, 1(1).

[Urgaonkar and Shenoy 2004] Urgaonkar, B. and Shenoy, P. (2004). Sharc: Managing CPU and Network Bandwidth in Shared Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 15(1):2–17.

[Urgaonkar et al. 2002] Urgaonkar, B., Shenoy, P., and Roscoe, T. (2002). Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 239–254, Boston, MA.

[Uttamchandani et al. 2005] Uttamchandani, S., Yin, L., Alvarez, G., Palmer, J., and Agha, G. (2005). Chameleon: a self-evolving, fully-adaptive resource arbitrator for storage systems. In *Proceedings of the 2005 USENIX Technical Conference*, pages 75–88, Anaheim, CA, USA.

[Valetto et al. 2005] Valetto, G., Kaiser, G., and Phung, D. (2005). A Uniform Programming Abstraction for Effecting Autonomic Adaptation onto Software Systems. In *2nd International Conference on Autonomic Computing (ICAC'05)*, pages 286–297, Seattle, WA, USA.

[van der Hoek 2001] van der Hoek, A. (2001). Integrating Configuration Management and Software Deployment. In *In Proceedings of the Working Conference on Complex and Dynamic Systems Architecture (CDSA 2001)*, Brisbane, Australia.

[van der Hoek 2004] van der Hoek, A. (2004). Design-time product line architectures for any-time variability. *Science of Computer Programming*, 53(30):285–304. special issue on Software Variability Management.

[van der Hoek et al. 1998] van der Hoek, A., Heimbigner, D., and Wolf, A. L. (1998). Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage à Trois. Technical Report CU-CS-849-98, Department of Computer Science, University of Colorado, Boulder, Colo., USA.

[van der Linden and Sventek 1992] van der Linden, R. J. and Sventek, J. S. (1992). The ansa trading service. *IEEE Comput. Soc. Tech. Comm. Newsl. Distrib. Process.*, 14(1):28–34.

[van Gurp et al. 2001] van Gurp, J., Bosch, J., and Svahnberg, M. (2001). On the notion of variability in software product lines. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 45–53, Washington, DC, USA. IEEE Computer Society.

[van Ommering et al. 2000] van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. (2000). The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85.

[van Renesse et al. 1998] van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., and Karr, D. (1998). Building adaptive systems using Ensemble. *Software–Practice and Experience*, 28(9):963–979.

[van Renesse et al. 1996] van Renesse, R., Birman, K. P., and Maffeis, S. (1996). Horus: a flexible group communication system. *Communications of the ACM*, 39(4):76–83.

[van Renesse et al. 2003] van Renesse, R., Birman, K. P., and Vogels, W. (2003). Astrolabe: a Robust and Scalable Technology for Distributed Systems Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206.

[van Steen et al. 1999] van Steen, M., Homburg, P., and Tanenbaum, A. (1999). Globe: A Wide-Area Distributed System. *IEEE Concurrency*, pages 70–78.

[Veríssimo and Rodrigues 2001] Veríssimo, P. and Rodrigues, L. (2001). *Distributed Systems for Systems Architects*. Kluwer Academic Publishers. 623 pp.

[Völter et al. 2004] Völter, M., Kircher, M., and Zdun, U. (2004). *Remoting Patterns: Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. John Wiley & Sons.

[Völter et al. 2002] Völter, M., Schmid, A., and Wolff, E. (2002). *Server Component Patterns*. John Wiley & Sons. 462 pp.

[W3C-WSA 2004] W3C-WSA (2004). W3C-WSA Group, Web Services Architecture. http://www.w3.org/TR/ws-arch/.

[Waldo 1999] Waldo, J. (1999). The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82.

[Waldo et al. 1997] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1997). A Note on Distributed Computing. In Vitek, J. and Tschudin, C., editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag.

[Waldspurger and Weihl 1994] Waldspurger, C. A. and Weihl, W. E. (1994). Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 1–11, Monterey, California.

[Wang et al. 2008] Wang, T., Vonk, J., Kratz, B., and Grefen, P. (2008). A survey on the history of transaction management: from flat to grid transactions. *Distributed and Parallel Databases*, 23(3):235–270.

[Wang 2001] Wang, Z. (2001). *Internet QoS*. Morgan Kaufmann. 240 pp.

[Warren et al. 2006] Warren, I., Sun, J., Krishnamohan, S., and Weerasinghe, T. (2006). An Automated Formal Approach to Managing Dynamic Reconfiguration. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 37–46, Washington, DC, USA. IEEE Computer Society.

[Watson 1981] Watson, R. W. (1981). Identifiers (naming) in distributed systems. In Lampson, B. W., Paul, M., and Siegert, H. J., editors, *Distributed Systems – Architecture and Implementation (An Advanced Course)*, volume 105 of *Lecture Notes in Computer Science*, pages 191–210. Springer-Verlag, Berlin.

[Weihl 1993] Weihl, W. E. (1993). Specifications of Concurrent and Distributed Systems. In Mullender, S., editor, *Distributed Systems*, chapter 3, pages 27–53. ACM Press Books, Addison-Wesley.

[Weikum and Vossen 2002] Weikum, G. and Vossen, G. (2002). *Transactional Information Systems*. Morgan Kaufmann. 853 pp.

[Weinand et al. 1988] Weinand, A., Gamma, E., and Marty, R. (1988). ET++ - An Object-Oriented Application Framework in C++. In *Proceedings of OOPSLA 1988*, pages 46–57.

[Weiser 1993] Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84.

[Welsh and Culler 2003] Welsh, M. and Culler, D. (2003). Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS-03)*, Seattle, WA, USA.

[Welsh et al. 2001] Welsh, M., Culler, D., and Brewer, E. (2001). SEDA: An Architecture for Well-Conditioned Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 230–243, Banff, Alberta, Canada.

[Wensley et al. 1976] Wensley, J. H., Green, M. W., Levitt, K. N., and Shostak, R. E. (1976). The design, analysis, and verification of the SIFT fault tolerant system. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*, pages 458–469, Los Alamitos, CA, USA. IEEE Computer Society Press.

[White 1976] White, J. E. (1976). A high-level framework for network-based resource sharing. In *National Computer Conference*, pages 561–570.

[Wieringa and de Jonge 1995] Wieringa, R. and de Jonge, W. (1995). Object identifiers, keys, and surrogates: Object identifiers revisited. *Theory and Practice of Object Systems*, 1(2):101–114.

[Wiesmann et al. 2000a] Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G. (2000a). Database replication techniques: a three parameter classification. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nürenberg, Germany. IEEE Computer Society.

[Wiesmann et al. 2000b] Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G. (2000b). Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan. IEEE Computer Society Technical Commitee on Distributed Processing.

[Wirth 1985] Wirth, N. (1985). *Programming in Modula-2, 3rd ed.* Springer Verlag, Berlin.

[Wollrath et al. 1996] Wollrath, A., Riggs, R., and Waldo, J. (1996). A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290.

[Wu and Kemme 2005] Wu, S. and Kemme, B. (2005). Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In *IEEE International Conference on Data Engineering (ICDE)*, Tokyo, Japan.

[X-OPEN/DTP] X-OPEN/DTP. Distributed Transaction Processing. The Open Group. http://www.opengroup.org/products/publications/catalog/tp.htm.

[X.500 1993] X.500 (1993). International Telecommunication Union – Telecommunication Standardization Sector, "The Directory – Overview of concepts, models and services", X.500(1993) (also ISO/IEC 9594-1:1994).

[Zhang et al. 2005] Zhang, Q., Sun, W., Riska, A., Smirni, E., and Ciardo, G. (2005). Workload-Aware Load Balancing for Clustered Web Servers. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):219–233.

[Zhao and Guibas 2004] Zhao, F. and Guibas, L. (2004). *Wireless Sensor Networks: An Information Processing Approach.* Morgan Kaufmann.

[Zimmermann 1980] Zimmermann, H. (1980). OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432.