1

Distributed Systems

Learning Objectives

In this chapter, we will investigate several case studies in order to acquaint ourselves with the principles of distributed systems. We will use these examples to understand the differences between centralized and distributed systems. The decision to opt for a distributed system architecture often arises from particular sets of non-functional requirements. We will investigate these scalability, reliability, performance, heterogeneity and openness requirements in order to comprehend their impact on distributed system architectures. We will recognize the importance of transparency and ascertain the transparency dimensions that are important for the engineering of distributed systems.

Chapter Outline

This book focuses on techniques and principles for constructing distributed systems. Intuitively, a distributed system consists of components that execute on different computers. For the system to appear as a whole, the components need to be able to interact with each other. This is achieved using a computer network. Before we can start investigating how distributed systems can be constructed, we will have to develop an understanding of what characterizes a distributed system. We need to know why we decide to build a system in a distributed way and what principles should be applied during this construction.

The decision to build a distributed rather than a centralized system is often derived from the requirements that the system's stakeholders express. The requirements that drive the decision towards a distributed system architecture are usually of a non-functional and global nature. Scalability, openness, heterogeneity and fault-tolerance are examples of such non-functional requirements. If systems have to be built in such a way that they can scale beyond the load that can be borne by a single computer, then the system has to be decomposed into components that reside on different computers and communicate via a computer network. Sometimes systems are required to be open and able to communicate with other systems, which are probably owned and operated by different organizations. It is not possible to have these components on the same computer in a centralized system, as organizations would not be willing to give up control. Systems also evolve, which means that new components are added over time. Old components, however, may not employ the latest technology, programming languages, operating systems, hardware and network protocols. It is generally impossible to rebuild components over and over again merely to incorporate the latest technology. This leads to heterogeneous systems, where older components remain on the hardware and operating system platforms for which they have been built and form a distributed system with newer components that are written in modern languages and run on newer platforms. Finally, it may be useful to duplicate critical components on different computers so that if one computer fails another one can take over and the overall system availability is not affected.

We study these requirements from different points of view in this chapter. In the first section, we characterize a distributed system by comparing it to a centralized system. We then review examples of distributed systems that have been built in practice. We use these examples to distil the requirements that lead engineers to adopt distributed system architectures and then finally review the facets of transparency that a well-designed distributed system should exhibit.

1.1 What Is a Distributed System?

1.1.1 Ingredients of a Distributed System

Before we can start to discuss how to engineer a distributed system, we have to understand what it is that we aim to construct; we have to give a definition of a distributed system. We have to reason about the differences between distributed systems and centralized systems. We have to find the typical characteristics of a distributed system. Finally, we should understand the motivation for building distributed rather than centralized systems.

Intuitively, a distributed system will have components that are distributed over various computers. A computer that hosts some components of a distributed system is referred to as a *host*. The concept of a host shall then denote all operational components of that computer including hardware and its network operating system software. Figure 1.1 visualizes this idea.

A host is a computer that executes components that form part of a distributed system.

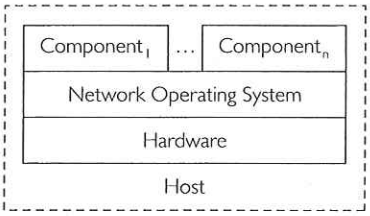

Figure 1.1
Hosts in a Distributed System

A distributed system has more than one component on more than one host. These components need to interact with each other. They need to provide access to each other's services and they need to be able to request services from each other. In theory, the components could do that directly by using the primitives that network operating systems provide. In practice, this would be too complex for many applications. As shown in Figure 1.2, distributed systems usually employ some form of *middleware* that is layered between distributed system components and network operating system components and resolves heterogeneity and distribution.

Middleware is a layer between network operating systems and applications that aims to resolve heterogeneity and distribution.

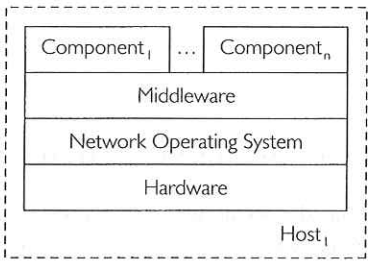

Figure 1.2
Middleware in a Distributed System

Example 1.1
Middleware Approaches

Transaction-oriented middleware supports the integration of transactions across distributed databases. Systems in this category include IBM CICS or Tuxedo. Message-oriented middleware systems enable the exchange of reliable message traffic across distributed components. Examples of this class include IBM MQSeries, the DEC MessageQueue or the Java Message Queue. Remote Procedure Calls (RPCs) were standardized by the OSF and supported the invocation of procedures across host boundaries. RPCs were the starting point for the development of object-oriented middleware, which includes the various available OMG/CORBA implementations, Remote Method Invocation (RMI) for Java, and Microsoft's COM. There are also non-mainstream products. These include mobile agent facilities, such as Aglets, implementations of tuple spaces that distributed threads can use to communicate, and so on.

An excellent understanding of the functionality and behaviour of middleware components is essential for the engineering of distributed systems. Hence, we have dedicated Part II of this book to a detailed discussion and comparison of those middleware systems that support the object-oriented paradigm, that is CORBA, RMI and COM.

A distributed system consists of components on networked hosts that interact via middleware so that they appear as an integrated facility.

For the purpose of this book, we employ a definition that we adapted from [Colouris et al., 1994]. As shown in Figure 1.3, a *distributed system* is a collection of autonomous hosts that that are connected through a computer network. Each host executes components and operates a distribution middleware, which enables the components to coordinate their activities in such a way that users perceive the system as a single, integrated computing facility.

A certain aspect of distribution is transparent if it is not visible to users, application programmers or administrators.

This definition demands a very important property of a distributed system: distribution shall be hidden from users. They shall rather perceive the system as a single computing facility. As we shall see later, it is also desirable to hide the complexity of distribution as much as possible from application programmers. This hiding is referred to as *transparency*. If we achieve this, application programmers can develop applications for distributed systems in very much the same way as they develop applications for centralized systems.

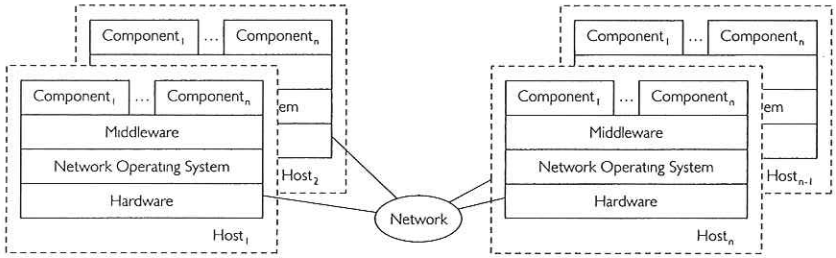

Figure 1.3
A Working Model of a Distributed System

1.1.2 Centralized Versus Distributed Systems

Let us now compare distributed systems with the centralized systems familiar to most readers. The aim of this comparison is to highlight the distinctive characteristics that are often found in distributed systems.

Centralized Systems

Centralized systems have non-autonomous components.

A centralized system may be composed of different parts. However, these parts, such as classes in an object-oriented program, are not *autonomous*; the system possesses full control over its parts at any one time. The parts become operational at the time the system is started and if the system is deactivated or deleted, then all parts of the system will be deactivated or deleted, too.

Centralized systems are often built using homogeneous technology.

Centralized systems are often fairly *homogeneous*. They tend to be constructed using the same technology. Very often the same programming language and the same compiler is used for all parts. The compiler generates the same type of machine code and all parts use the same representation for data. Parts are then either statically or dynamically loaded by the

same loader or linker. Communication between parts can be achieved very efficiently using programming language primitives, such as procedure calls or method invocations.

Some centralized systems support multiple users. Database applications are a good example. The database and its forms and reports are a *shared resource*. All users share a centralized system all the time and the system might become overloaded at times.

Multiple users share the resources of a centralized system at all times.

A centralized system can be built to run in a single process. It is even possible to construct it single-threaded and designers can build centralized systems that are not concurrent programs.

There is only a single *point of control* in a centralized system. The state of a centralized system can be characterized by the program counter of the processor, the register variable contents and the state of the virtual memory that is used by the process that runs the system. Centralized systems have a single *point of failure*; either they work or they do not work. Situations where parts of the system are unreachable due to a communication failure generally do not occur.

Centralized systems have a single point of control and of failure.

Distributed Systems

Distributed systems use a more coarse-grained structuring concept; distributed systems have multiple *components* and these may be decomposed further into parts. Components are autonomous; they possess full control over their parts at all times. There is, however, no master component that possesses control over all the components of a distributed system. In order for the distributed system to appear as an integrated computing facility, components need to define interfaces by means of which they communicate with each other.

Distributed systems have autonomous components.

The components of a distributed system need not be homogeneous. In fact, they are often rather heterogeneous. Heterogeneity arises, for example, from a need to integrate components on a legacy hardware platform, such as an IBM mainframe, with components newly-written to operate on a Unix workstation or a Windows NT machine. Components then might be written in different programming languages; mainframe programs are often written in Assembler, RPG or Cobol while new components will be written in Visual Basic, Java or C++. The source of components might be compiled into heterogeneous machine code. The machine code might use different data representation formats. IBM mainframes use a big-endian representation for integers and EBCDIC character codes while most Unix workstations work with a little-endian representation for integers and a 7-bit ASCII or an 8-bit ISO character encoding.

Distributed systems may be built using heterogeneous technology.

In a distributed system there may be components that are used exclusively by a single user (they are known as unshared components). In fact, this is one of the major advantages of distributed systems. If a component becomes overloaded by too many users or too many requests from other components, another component capable of performing the same services can be added to the distributed system and the load can be split among them. Moreover, components can be located so that they are local to the users and other components with which they interact. This locality improves the overall performance of the

Distributed system components may be used exclusively.