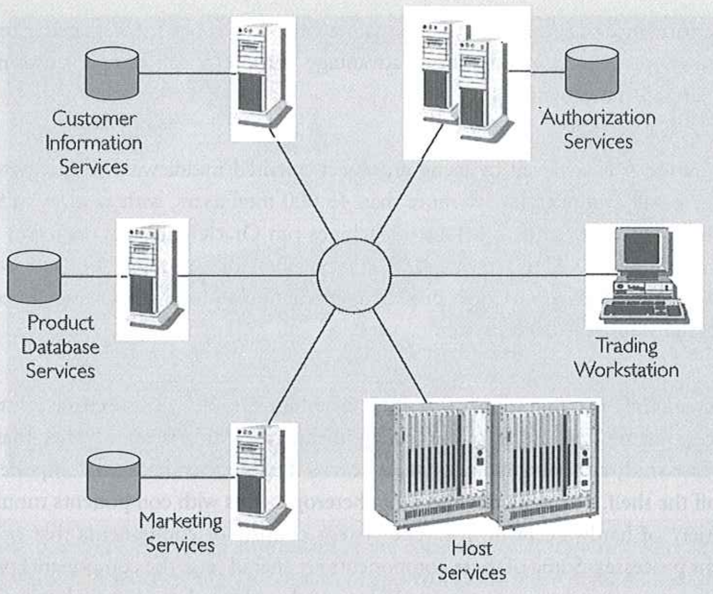
Example 1.4
IT Service Architecture of a Swiss Bank



The bank runs a highly-integrated set of applications on large Unisys and IBM mainframes. The mainframes process approximately 2 million transactions each day and they host approximately 13,000 different applications. Recently, hardware platforms, such as Unix servers and PCs running various versions of the Windows operating system, were added to the infrastructure. The IT infrastructure used to be product-orientated. Every product that the bank offered was supported by an application. Before a product could enter the market, applications supporting the product had to be written. These applications evolved on different hardware and operating system platforms. Recently the bank has shifted the focus from products to customers. Agents in the bank now want to be able to obtain a customer profile that displays information about all products that the customer has with a bank. This demands integration between the various product-orientated applications. Moreover, agents expect applications to be integrated. By integrating the share trading application with the account management application, the purchase of an equity package can be debited instantaneously to the current account, rather than in an overnight batch. In order to meet these challenges, the bank defined the above service architecture.

requests in order to provide a higher-level service. The object-oriented middleware is used to broker service requests between the heterogeneous objects.

The implementation of this architecture has brought the bank several substantial advantages. It can now integrate applications with each other although they might not operate on one of the mainframes. Hence, the new architecture reduces the time that is needed for the creation of applications supporting new products as now the bank can use modern programming languages and operating systems to write and execute new components. The components that encapsulate applications each have very well-defined interfaces leading to a more maintainable infrastructure. Yet the mainframes and the large body of applications that run on them remain operational and the bank's huge investment in them is preserved.

Again, the bank's IT infrastructure exhibits all the distributed system characteristics that we suggested above. The components are rather autonomous. Some of the legacy components cannot be modified any more. There is a substantial heterogeneity among the components. Hardware platforms range from Unisys mainframes to Unix servers to Windows NT workstations. The bank also used a variety of programming languages including Assembler, Cobol, C, and C++. There are some components that are shared (e.g. the account databases) and others that are not shared (the user interface components to various applications). The system supports some 2,000 users and therefore consists of multiple concurrent processes. The bank is also aware of the possibility of multiple points of failure. To overcome this problem they have made a transaction management system an essential part of the middleware platform upon which they constructed their infrastructure.

The service infrastructure exhibits all characteristics of a distributed system.

1.2.4 Distributed Soccer League Management

While the three previous case studies are real, the distributed soccer league management system that we introduce now is invented. We will use it as a running example throughout this book. Its use is motivated by didactic considerations; we will be able to twist the case study in such a way that we can demonstrate all the concepts relevant to this book. We could not do this with an existing case study.

The purpose of the application is to support a national soccer association in managing soccer leagues. The application has to keep an account of soccer clubs, each of which might have several teams. The soccer clubs maintain the composition of teams. Players need to obtain a license from the soccer association. Teams have to be registered with the soccer association, too. The soccer association also runs national teams that draw their squads by selecting the best players from the teams according to certain selection criteria.

The national soccer league is inherently distributed. The clubs are geographically dispersed. The league management will, therefore, need to provide services for soccer teams to register players and to register teams. The clubs need to provide services to the soccer association so that it can make appointments with players for games of national soccer teams.

The soccer league management application is likely to be rather heterogeneous. This stems from the fact that the different clubs are autonomous. They want to use their existing computing infrastructure to provide the services demanded by the soccer association. Hence, the system will include different flavours of the Windows operating system, some clubs might use Unix machines and the traditional clubs might even have a mainframe. Likewise, different programming languages will be used for the implementation of services demanded by the soccer associations. Some clubs will already have a relational database with all the details that the association wants. They will just use a programming language with embedded SQL. Others will start from scratch; they may buy a package, such as Microsoft Office, off-the-shelf and then use Visual Basic to program an Access database.

The soccer league management system has multiple points of control. Each club manages its teams and players independently from and concurrently with the league management. The component run by the soccer association needs to be multi-threaded; it needs to be able to react to multiple concurrent requests of clubs registering players for a license. Similarly, the components in the clubs need to be able to react to a player appointment request from the component that manages the composition of national soccer teams.

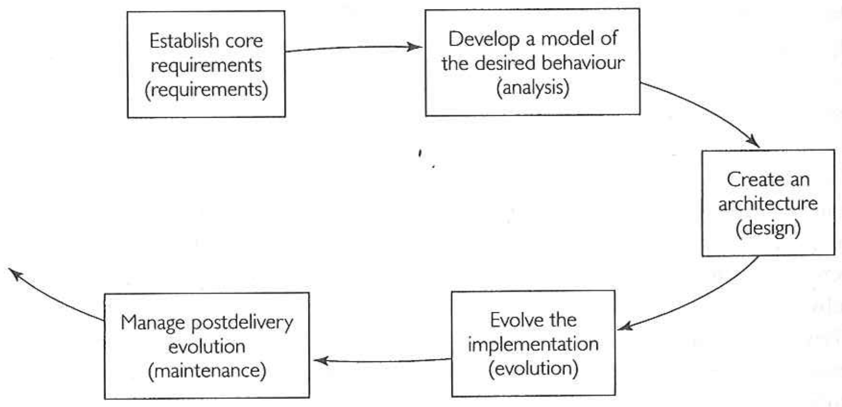
There may be multiple points of failure in our distributed soccer management application. Clubs might not be connected to the soccer association by high-quality network connections, but rather by slow modem or ISDN connections. Computers in particular clubs might be switched off occasionally. This may lead to situations in which the league management system is only partially operational. Thus, a distributed system must be designed in such a way that a failure in one component must not bring down the whole system.

1.3 Distributed System Requirements

In the first section of this chapter, we have briefly compared centralized and distributed systems. The second section has shown common properties of some case studies. In this section we look at distributed systems from yet a different angle. We take an engineering perspective and look at the software engineering process in order to see when the factors that influence the choice of a distributed system architecture need to be identified. We then discuss the requirements that lead to the adoption of a distributed system architecture.

Figure 1.4 shows a system development process that we adopted from [Booch, 1995]. It includes a requirements stage where the system is conceptualized and where the user's requirements are determined. The purpose is to find out what the customer really wants and needs. In the analysis stage, the behaviour of the system functionality is formalized. The results of the conceptualization and analysis stages determine any further stages in the process.

Figure 1.4
Booch's Macro Development Process


Establish core requirements (requirements)
Develop a model of the desired behaviour (analysis)
Create an architecture (design)
Evolve the implementation (evolution)
Manage postdelivery evolution (maintenance)

The need to distribute a system is often derived from non-functional requirements.

The requirements stage involves identifying the functional and non-functional properties that stakeholders demand from a system. Functional requirements are concerned with the functions that the system can perform for its users. Functional requirements can usually be localized in particular components. Following the Boochean approach, functional requirements are then formalized during the object-oriented analysis. Non-functional requirements are concerned with the quality of the system. It tends to be difficult to attribute them to particular parts of the system; they are, rather, global. Non-functional requirements, therefore, have a serious impact on which overall architecture is chosen for the system. This is done during the design stage.

The non-functional requirements that typically lead to the adoption of distributed system architectures are:

1. Scalability
2. Openness
3. Heterogeneity
4. Resource Sharing
5. Fault-Tolerance

It should have become clear by now that the construction of a centralized system is considerably simpler, and therefore cheaper, than building a distributed system. Hence, it is beneficial not to build a distributed system if it can be avoided. However, some non-functional requirements just cannot be achieved by a centralized system. Let us now develop a better understanding of these requirements.

1.3.1 Scalability

When designing a power system, an important factor that engineers have to take into account is the amount of power that the users of the system will draw. This is the same for software architectures. A single generator is inappropriate as the power supply for the National Grid for the same reason that a single PC is inappropriate as a video-on-demand server: neither could bear the load.

The load that a software system has to bear can be determined in many different dimensions. It can be expressed by the maximum number of concurrent users. The Hong Kong video-on-demand service might have to serve up to 90,000 concurrent users when it goes into service. Load can also be defined by the number of transactions that the system needs to execute in a given period. The Swiss bank stated that it would perform up to two million transactions per day. Finally, load can be determined by estimating the data volume that has to be handled by the system. The data volume that Boeing's aircraft configuration management system has to maintain increases by engineering documents for approximately 1.5 billion aircraft parts each year.

For a power system, the meaning of bearing the load is obvious: we expect the system to provide the power when we need it. The same holds for computer systems; in addition to continuing the service, we expect certain qualities of service. *Qualities of service* are concerned with how the function is performed. We expect these qualities even if the system is operating under high load. A clerk in a bank expects the transaction to be completed within a couple of seconds. Likewise, a user of a video-on-demand service expects videos to arrive at a rate of at least 10 frames per second. Otherwise, the service is considered inappropriate.