the construction engineer finalized so that they can be retrieved by the maintenance engineer, if necessary.

Security of resource access needs to be considered.

The fact that components are not exclusively used by a single user on a single machine has security implications. It has to be defined who is allowed to access shared data in a distributed system. To be able to do that, a notion of users has to be introduced and systems have to validate that the users actually are who they claim to be. Hence, a secondary requirement is to control access to a resource that needs to be shared. In the Hong Kong Telecom's video-on-demand system, access needs to be controlled so that only authorized users are allowed to download videos; the company wants users to register and pay before they can start to download videos.

The right to access a resource is implemented by resource managers. A *resource manager* is a component that grants access to a shared resource. The video-on-demand server is an example of such a resource manager. It manages the videos owned or licensed by Hong Kong Telecom. It will force users to identify themselves and it will authenticate them against a registered user set. It will only allow authenticated users to download a video.

Distributed objects provide a sophisticated model of resource sharing.

Resource managers and their users can be deployed in different ways in a distributed systems architecture. In a *client-server architecture*, there are servers that manage and provide certain resources and clients that use them. We use a more sophisticated model for resource sharing in this book. It is based on the concept of distributed objects. A distributed object represents and encapsulates a resource that it uses to provide services to other objects. Objects that provide services, in turn, might rely on services provided by other objects. This leads to an architecture that has various layers and is therefore sometimes also referred to as an *n-tier architecture*.

1.3.5 Fault-Tolerance

Hardware, software and networks are not free of failures. They fail because of software errors, failures in the supporting infrastructure (power-supply or air conditioning), abuse by their users, or just because of ageing hardware. The lifetime of a hard disk, for instance, lies between two and five years, much less than the average lifetime of a distributed system.

Operations that continue even in the presence of faults are referred to as fault-tolerant.

An important non-functional requirement that is often demanded from a system is fault-tolerance. *Fault-tolerance* demands that a system continues to operate, even in the presence of faults. Ideally, fault-tolerance should be achieved with limited involvement of users or system administrators. They are an inherent source of failures themselves. Fault-tolerance is a non-functional requirement that often leads to the adoption of a distributed system architecture.

Distributed systems can be more fault-tolerant than centralized systems. Fault-tolerance can be achieved by limiting the effect of failures. If a client in Hong Kong fails to watch a video, it does not necessarily mean that all other users will be affected by this failure. The overall system more or less continues to work. Fault-tolerance of components, such as the video-on-demand server, is achieved by redundant components, a concept referred to as *replication*. If a component fails then a replica of that component can step in and continue to serve.

Given that there are more hosts in a distributed system, each running several processes, it is much more likely that failures occur than in a centralized system. Distributed systems, therefore, have to be built in such a way that they continue to operate, even in the presence of the failure of some components.

Fault-tolerant components are built in such a way that they are able to continue to operate although components that they rely on have failed. They detect these failures and recover from them, e.g. by contacting a replica or an equivalent service.

Distributed systems achieve fault-tolerance by means of replication.

1.4 Transparency in Distributed Systems

The design of a distributed system architecture that meets all or some of the above requirements is rather complicated. Our definition of distributed systems demands that the distributed system appear as a single integrated computing facility to users. In other words, the fact that a system is composed from distributed components should be hidden from users; it has to be transparent.

In addition, it is also highly beneficial to hide the complexity of distributed system construction from the average application engineer as much as possible. They will be able to construct and maintain applications much more efficiently and cost-effectively if they are not slowed down by the complexity introduced through distribution.

There are many dimensions of transparency. These transparency dimensions were first identified in [ANSA, 1989]. Because they were so fundamentally important, they formed an important part of the International Standard on Open Distributed Processing (ODP) [ISO/IEC, 1996]. Figure 1.5 shows an overview. These transparency criteria will be discussed in detail in this section. The figure suggests that there are different levels of transparency. The criteria at lower levels support achieving the transparency criteria at higher levels.

Transparency in distributed systems has several different dimensions.

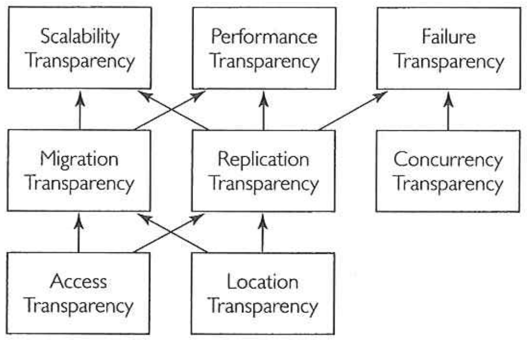

Figure 1.5
Dimensions of Transparency in Distributed Systems

A detailed understanding of the transparency dimensions is useful for several reasons. The dimensions provide a yardstick against which we can measure middleware components. We can use them to review the extent to which middleware systems actually support the application engineer in achieving overall distribution transparency. Moreover, we can use the transparency criteria to review the design of distributed components. They will allow us to distinguish well-designed components from components that cause problems during the operation, administration or maintenance.

1.4.1 Access Transparency

Access transparency means that the interfaces for local and remote communication are the same.

For a distributed system to appear as a single integrated computing facility, the system components have to interact with each other. We assume that a component interacts with another component by requesting execution of a service. *Access transparency* demands that the interface to a service request is the same for communication between components on the same host and components on different hosts. This definition is rather abstract. Let us review some examples of access transparency in Example 1.5.

Example 1.5
Example of Access Transparency

Users of the Unix network file system (NFS) can use the same commands and parameters for file system operations, such as copying or deleting files, regardless of whether the accessed files are on a local or a remote disk. Likewise, application programmers use the same library function calls to manipulate local and remote files on an NFS.

In our soccer management application, a team component offers some services that are implemented by operations. An operation has a name, a parameter list and a return type. The way any of these operations is invoked is independent of the location. Invocation is identical for a club component that resides on the same machine as the team component and for a national component that resides on the host machine of the soccer association.

Access transparency is an important property of a distributed system. A component whose access is not transparent cannot easily be moved from one host to another. All other components that request services would first have to be changed to use a different interface. We will review how access transparency is achieved with middleware in Chapters 3 and 4.

1.4.2 Location Transparency

Location transparency means that service requesters do not need to know physical component locations.

To request a service, components have to be able to address other components. They have to identify the component from which they want to request the service. *Location transparency* demands that components can be identified in service requests without knowing the physical location (that is the host) of the component.

Example 1.6
Examples of Location Transparency

A user of a Unix NFS can access files by addressing them with their full pathname. Users do not have to know the host name or the IP address of the machine that serves the partition of the file system that contains the file. Likewise, application programmers just pass the pathname identifying a file to the Unix library function to open a file and they can then access the file using an internal identifier.

The users of a video-on-demand system do not have to know the name or the IP address from which their client component downloads a video. Similarly, the application programmer who wrote the client component should not have to know the physical location of the video-on-demand server that provides the video.

Location transparency is another primary property that any distributed system should have. If location is not transparent, the administration of the system will become very difficult. In particular, it becomes next to impossible to move components from one server to another. When an NFS administrator decides to move a partition, for instance because a disk is full, application programs accessing files in that partition would have to be changed if file location is not transparent for them. Hence, middleware systems have to support identifying and locating components in such a way that the physical component location remains transparent to the application engineer. We discuss techniques and their implementations for achieving location transparency in Chapter 8.

1.4.3 Migration Transparency

Migration transparency means that a component can be relocated without users or clients noticing it.

It sometimes becomes necessary to move a component from one host to another. This may be due to an overload of the host or to a replacement of the host hardware. Moving could also be needed due to a change in the access pattern of that component; the component might have to be relocated closer to its users. We refer to this removal of components as *migration*. *Migration transparency* refers to the fact that components can be migrated without users recognising it and without designers of client components or the component to be migrated taking special considerations.

Example 1.7
Example of Migration Transparency

In the soccer management application, players tend to move from one team to another. Migration transparency means that components, such as the application managing the national soccer team, do not have to be aware of the fact that the player component has moved to a different team.

Migration transparency depends on access and location transparency.

Migration transparency depends on both access and location transparency. Migration cannot be transparent if the interface of a service request changes when a local component is moved to a remote host or vice versa. Likewise, migration cannot be transparent if clients need to know the physical location of a component.

Migration transparency is a rather important property. It is very difficult to administer a distributed system in which components cannot be freely relocated. Such a distributed system becomes very inflexible as components are tied to particular machines and moving them requires changes in other components. We discuss how migration transparency is achieved in Chapter 9 when we discuss the life cycle of distributed objects.

1.4.4 Replication Transparency

A replica is a component copy that remains synchronized with its original.

It is sometimes advantageous to keep copies of components on different hosts. These copies, however, need to be tied together. If they maintain an internal state, that state needs to be synchronized in all copies. We refer to copies of components that meet this requirement as *replicas*. The process of creating a replica and keeping the replicas up-to-date with the original is referred to as *replication*. Replication is used to improve the overall