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

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.

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

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.

Example 1.7
Example of Migration 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 depends on access and location transparency.

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

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

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

performance and to make systems more scalable. Moreover, if one replica fails other replicas can stand in.

Replication transparency means that users and programmers do not know whether a replica or a master provides a service.

Replication should be transparent in a distributed system. *Replication transparency* refers to the fact that users and application programmers are not aware that a service they are using is not provided by an "original" master component, but by a replica. Replication transparency also refers to the fact that application engineers who build a component should not have to take into account that the component might be replicated.

Example 1.8
Example of Replication Transparency

In the video-on-demand service, we could imagine that it would be beneficial to replicate the video database server. In that way, the video-on-demand system scales to accommodate many concurrent users. Replication transparency means that customers who watch a video are not aware that the video is downloaded from a replica server rather than the original server. It also means that the engineers who wrote the client software did not have to design them in such a way that they were capable of contacting a replica rather than the original database. Finally, replication transparency means that the engineer who wrote the database server did not have to build the server in such a way that it could be replicated.

Replication transparency depends on access and location transparency.

Replication transparency depends on access and location transparency in the same way as migration transparency does. Replicas are typically held on different machines from that which hosts the master component in order to distribute the load. To achieve replication transparency, we need the way that services are requested to be the same for the replica and the master and we must avoid clients needing to know the location of either the replica or the master component.

Without replication transparency, the resulting systems will be difficult to use and even more difficult to scale to accommodate a growing load. If replication is not transparent for the construction of components providing a service, it will not be possible to replicate existing components that have not been constructed to be replicated. This complicates administration, as components have to be changed when replication becomes necessary.

1.4.5 Concurrency Transparency

Concurrency transparency means that users and programmers are unaware that components request services concurrently.

When we characterized distributed systems, we pointed out that they are inherently concurrent. Hence, multiple components may exist and operate concurrently. Concurrency of components, however, should be transparent to users and application engineers alike. *Concurrency transparency* means that several components may concurrently request services from a shared component while the shared component's integrity is preserved and neither users nor application engineers have to see how concurrency is controlled. This definition is rather abstract and it needs to be illustrated with a number of examples.

Example 1.9
Examples of Concurrency Transparency

In the bank application (see Figure 1.4), the mainframe will host a component that manages the current accounts of customers. When an equity trader buys equities on behalf of a customer, the costs are debited to the customer's current account. Concurrently, an advisor in the bank might print a statement of the same account. It is transparent for the two users that they are accessing the account at the same time. The application engineers who developed the two components should not have to consider the potential of concurrent accesses to the same account, either. In the soccer league management application, the trainer of a club team might use an application to compose training and playing schedules. That application would request a service to make appointments from individual player components. Concurrently, the trainer of the national soccer team. The users are unaware of each other and the engineers who build the team management applications do not have to take measures to avoid concurrency conflict.

The integrity of resources in the presence of concurrent accesses and updates has to be retained. Potential implications of integrity violations include incorrect financial accounts, money that is lost during financial transactions or soccer players who are unavailable for important games due to a double booking.[1] Hence, concurrency has to be controlled somehow. If concurrency were made visible to users, they would recognize that they are not the sole users of the system. They might even see what other users do. This is only acceptable in exceptional cases (e.g. in computer supported co-operative work systems) but users in general would not be willing to implement concurrency control. If concurrency control were to be implemented by the application engineers, the construction of applications would be much more complicated, time-consuming and error-prone. Fortunately, there are techniques available so that concurrency control can be implemented in a middleware component in such a way that concurrency can be transparent to both users and application engineers. We discuss these techniques and the use of middleware systems that implement them in Chapter 11.

1.4.6 Scalability Transparency

Scalability transparency means that users and programmers do not know how scalability of a distributed system is achieved.

As discussed above, one of the principle objectives to build a distributed rather than a centralized system is to achieve scalability. *Scalability transparency* denotes another high-level transparency criterion and demands that it should be transparent to designers and users how the system scales to accommodate a growing load. Scalability transparency is very similar to performance transparency in that both are concerned with the quality of service that is provided by applications. While performance is viewed from the perspective of a single request, scalability transparency considers how the system behaves if more components and more concurrent requests are introduced.

[1] I leave the reader to decide which of these is the worst fault.