Example 1.10
Example of Scalability Transparency

The Internet is a good example of a distributed system that scales transparently. The size of the Internet increases daily because new hosts are continuously added. These hosts generate additional network traffic. To improve, or at least retain, the quality of service of the existing sites, the bandwidth of routers and backbone connections is being improved continuously. Through the very careful design of the Internet, however, improving the physical layer does not affect existing nodes or even applications of the Internet. Thus, the Internet scales in a transparent way.

Scalability transparency depends on replication and migration transparency.

Achieving scalability transparency is supported by replication transparency and migration transparency in the same way as they support performance transparency. If a system supports replication, we can scale it by adding more replicas. Moreover, we can add new hosts and populate them with existing components that we migrate from overloaded hosts.

If scalability is not transparent to users they will be annoyed by degrading performance as the overall system load gradually increases. There might be situations where a system that functions correctly is rendered unusable because it cannot accommodate the load. In banking applications, for example, systems often have to complete batch jobs overnight. If they fail to complete the batch because it is too complex, the bank cannot resume business on the following day. If scalability is not transparent to application developers, a consequence is that the system (often its architecture) has to be changed. These changes tend to be rather expensive.

1.4.7 Performance Transparency

Performance transparency means that users and programmers are unaware how good system performance is maintained.

Another high-level transparency dimension is *performance transparency*. This demands that it is transparent to users and application programmers how the system performance is actually achieved. When considering performance, we are interested in how efficiently the system uses the resources available. These resources can be the time that elapses between two requests or the bandwidth that the system needs. Again, performance transparency is supported by lower-level concepts as it relies on replication transparency and migration transparency.

Example 1.11
Example of Performance Transparency

As an example of performance transparency, consider a distributed version of the `make` utility [Feldman, 1979] that is capable of performing jobs, such as compiling a source module, on several remote machines in parallel. Complex system compilation can be performed much faster using multiple processors. It not only considers the different processors and their capabilities, but also their actual load. If it can choose from a set of processors, it will delegate the compile jobs to the fastest processor that has the lowest load. In this way, the distributed `make` achieves an even better performance. Programmers using a distributed `make`, however, do not see or choose which machine performs which job. The way in which the actual performance is achieved is transparent for them.

Replication of components provides the basis for a technique called *load balancing*, which achieves performance transparency. A middleware system that implements load balancing selects the replica with the least load to provide a requested service. The middleware system should perform this balancing decision in a way that is transparent to both users and application programmers. The performance of a system can be improved if components are relocated in such a way that remote accesses are minimized. This usually requires migration of components. Achieving an optimal performance would then demand that access patterns to components are monitored and component migration is triggered when necessary. We achieve performance transparency if the underlying middleware triggers component migration in a way that is transparent to users, application programmers and possibly even administrators.

Performance transparency depends on replication and migration transparency.

Performance transparency is rather difficult to achieve. Only a few middleware systems to date are actually capable of performing load balancing. The reason is that it is genuinely difficult to automatically predict the load that a component is going to cause on a host in the future. Because of that, achieving good performance of a distributed system still requires significant intervention of application designers and administrators.

1.4.8 Failure Transparency

We have identified that failures are more likely to occur in distributed systems. *Failure transparency* denotes the concealment of faults from users, client and server components. Hence, failure transparency means that components can be designed without taking into account that services they rely on might fail. The same consideration also applies to server components. Failure transparency implies that server components can recover from failures without the server designer taking measures for such recovery. As shown in Figure 1.5, failure transparency is a high-level transparency criterion. Its achievement is supported by both concurrency and replication transparency.

Failure transparency means that users and programmers are unaware of how distributed systems conceal failures.

If failures have to be concealed, we have to ensure that the integrity of components is not violated by failures. Such integrity violations are often due to related updates that are only partially completed. We will see that transactions are used to ensure that related changes to different components are done in an atomic way. Transactions, in turn, are implemented based on the mechanisms that achieve concurrency control. We will discuss this in Chapter 11 in more detail. In addition to transactions, replication can be employed to achieve failure transparency. Replicas can step in for a failed component and provide a requested service instead of the original component. When this is done in a way that is transparent to users and client components, failures of components are concealed.

Failure transparency depends on concurrency and replication transparency.

Example 1.12
Example of Failure Transparency

Failure transparency is particularly important for the banking application. Customers will not tolerate it if an automatic teller machine does not deliver the cash they wanted. They also worry whether the teller machine actually debited money from the account without dispensing the cash equivalent. Failures that might occur during a cash withdrawal should be concealed from the customer. The design of the equity trading packages or marketing packages used in the bank should not have to be adjusted to recover from faults. Designers of these applications should be able to assume that components they rely on are or will be made available. Moreover, components that provide services to other components should be re-activated and recover to an earlier consistent state

Failures should be transparent to users; otherwise, they will be unsatisfied with the system. If failures of servers are not transparent to designers of client applications, they have to build measures into their clients to achieve failure transparency for their users. This will complicate the design of clients and make them more expensive and difficult to maintain.

Key Points

- Distributed systems consist of a number of networked hosts, each of which executes one or several components. These components use middleware to communicate with each other and the middleware hides distribution and heterogeneity from programmers to some extent.
- Developers often do not have the choice to develop a centralized system architecture because non-functional requirements that include scalability, openness, heterogeneity, resource sharing and fault-tolerance force them to adopt a distributed architecture.
- We have discussed different dimensions in which distribution can be transparent to users, application designers and administrators. These dimensions are access, location, concurrency, failure, migration, replication, performance and scalability transparency. These provide design guidelines for distributed systems.
- The transparency dimensions we identified are not independent of each other. Access, location and concurrency are low-level transparency dimensions. They support higher level dimensions, such as failure, performance and scalability transparency.

Self Assessment

1.1 Name five reasons for building a distributed system.
1.2 What is the difference between a client-server system and a distributed system?
1.3 Is a three-tier architecture a distributed system?
1.4 Why do we not build every system as a distributed system?
1.5 What is the relationship between requirements engineering and distributed systems?
1.6 What are the eight dimensions of transparency in distributed systems?
1.7 What can the transparency dimensions be used for?
1.8 What is the difference between location and access transparency?
1.9 What are the differences between performance and scalability transparency?

Further Reading

Engineering distributed systems follows a development process that is not much different from that for other software systems. Throughout this book, we assume that the reader is familiar with basic software engineering principles and techniques. We recommend [Ghezzi et al., 1991], [Pressman, 1997] or [Jalote, 1991] as a reference. A good introduction to object-oriented software engineering is given in [Jacobson et al., 1992].

The selection of particular distributed system architectures is often driven by non-functional requirements. The requirements engineering process that precedes the design of a distributed system is beyond the scope of this book and we refer to [Sommerville and Sawyer, 1997] as a good reference. Moreover, several software engineering standards, such as those discussed in [Mazza et al., 1994] include practices for good requirements engineering. They define categories of non-functional requirements that influence the architecture of distributed systems.

Other textbooks employ a definition of distributed systems that slightly differs from ours. The reader might deepen the understanding of relevant concepts by comparing our definitions with the ones given in [Sloman and Kramer, 1987], [Colouris et al., 1994] and [Mullender, 1993].

[ANSA, 1989] mentions different dimensions of distributed system transparency for the first time. Transparency dimensions are also discussed in the Open Distributed Processing (ODP) standard of the International Organization for Standardization (ISO) [ISO/IEC, 1996].