Original Text to Transparencies (from the Book):

Access transparency deals with hiding differences in data representation and the way that objects can be accessed. At a basic level, we want to hide differences in machine architectures, but more important is that we reach agreement on how data is to be represented by different machines and operating systems. For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions. Differences in naming conventions, differences in file operations, or differences in how low-level communication with other processes is to take place, are examples of access issues that should preferably be hidden from users and applications.

An important group of transparency types concerns the location of a process or resource. **Location transparency** refers to the fact that users cannot tell where an object is physically located in the system. Naming plays an important role in achieving location transparency. In particular, location transparency can often be achieved by assigning only logical names to resources, that is, names in which the location of a resource is not secretly encoded. An example of a such a name is the **uniform resource locator** (URL) `http://www.prenhall.com/index.html`, which gives no clue about the actual location of Prentice Hall's main Web server. The URL also gives no clue as to whether the file `index.html` has always been at its current location or was recently moved there. For example, the entire site may have been moved from one data center to another, yet users should not notice. The latter is an example of **relocation transparency**, which is becoming increasingly important in the context of cloud computing to which we return later in this chapter.

Where relocation transparency refers to *being* moved by the distributed system, **migration transparency** is offered by a distributed system when it supports the mobility of processes and resources initiated by users, without affecting ongoing communication and operations. A typical example is communication between mobile phones: regardless whether two people are actually moving, mobile phones will allow them to continue their conversation. Other examples that come to mind include online tracking and tracing of goods as they are being transported from one place to another, and teleconferencing (partly) using devices that are equipped with mobile Internet.

As we shall see, replication plays an important role in distributed systems. For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed. **Replication transparency** deals with hiding the fact that several copies of a resource exist, or that several processes are operating in some form of lockstep mode so that one can take over when another fails. To hide replication from users, it is necessary that all replicas have the same name. Consequently, a system that supports replication transparency should generally support location transparency as well, because it would otherwise be impossible to refer to replicas at different locations.

We already mentioned that an important goal of distributed systems is to allow sharing of resources. In many cases, sharing resources is done in a cooperative way, as in the case of communication channels. However, there are also many examples of competitive sharing of resources. For example,

two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database. In such cases, it is important that each user does not notice that the other is making use of the same resource. This phenomenon is called **concurrency transparency**. An important issue is that concurrent access to a shared resource leaves that resource in a consistent state. Consistency can be achieved through locking mechanisms, by which users are, in turn, given exclusive access to the desired resource. A more refined mechanism is to make use of transactions, but these may be difficult to implement in a distributed system, notably when scalability is an issue.

Last, but certainly not least, it is important that a distributed system provides **failure transparency**. This means that a user or application does not notice that some piece of the system fails to work properly, and that the system subsequently (and automatically) recovers from that failure. Masking failures is one of the hardest issues in distributed systems and is even impossible when certain apparently realistic assumptions are made, as we will discuss in Chapter 8. The main difficulty in masking and transparently recovering from failures lies in the inability to distinguish between a dead process and a painfully slowly responding one. For example, when contacting a busy Web server, a browser will eventually time out and report that the Web page is unavailable. At that point, the user cannot tell whether the server is actually down or that the network is badly congested.

1. Distribution Transparency

   You need to design a distributed system, which should achieve maximal *scalability*. Assume that only desktop machines will access to the system. Given the transparencies: access, location, migration, relocation, replication, concurrency and failure, which transparency(ies) will you implement? Explain your answers! **(2 points)**

   Expected output:
   I will select *transparency1* because …
   I do not need to choose *transparency2* because …

Transparency	**Reason**
Access	I do not select access transparency as the data representation even though it doesn't do harm to the scalability, but it doesn't help either. (even though there could be an overhead in transforming parameters [due to varying representation of transmitted information])
Location	I do select location transparency as showing a fixed port or IP is not an option if a system should be scalable (it would be better if it is done dynamically depending on server load).
Relocation	I do select relocation transparency as the user should be unaware if server sends objects to another location (ex. Another server via RMI). As in a scalable system it could go over several tiers this could cause severe delays.
Migration	I do select migration transparency as it is more convenient for a user if he has no delays or similar things if the server in the background changes abruptly.
Replication	I do select replication transparency as there in a scalable system there are several replicas and with each one it gets harder for the client to manage the connections to the replicas. With replication transparency the client has less overhead.
Concurrency	I do select concurrency transparency as there is a high need of concurrency for every scalable system. Concurrency transparency isn't implemented for example if there is

	only one thread that manages all clients and there is more than one client at a given time. Therefore, in a scalable system there is a need for high concurrency to decrease waiting times on client side and get a better usage of resources on the server side.
Failure	I do not select failure transparency even though it is important for a working scalable system, it will cause an overhead due to replicas for possible failures. Which lead to more cost in scaling. A load balancer has the same problem.

Not part of the Question:

A P2P Overlay network would be an option here. It scales better than a client-software architecture.

2. Scalability

Let *T(R, L)* denotes the execution time of an application processing a problem of size *L* when being distributed among *R* nodes in a distributed system. For example, a response time of a service assuming the presence of *L* users and *R* servers.

a. Define *efficiency E(pR, L)* and *scalability S(pR, NL)* of an application through *T*. Assume that resources will be increased *p* times in order to handle the increased load of *N* times. **(2 points)**

b. Write possible values of *E(pR, L)* and *S(pR, NL)*. Assume a well/badly-scaling system! **(2 points)**

Expected output b) : a < *E(pR, L)* < b; c < *S(pR, NL)* < d

a.

$$\frac{T(R,L)}{p} \leq E(R,L) \leq T(1,L) + (p-1) * overhead$$

Minimum = no overhead, perfect parallelisation

Maximum = only one resource is used at any given time (no parallelisation), the rest of the resources (p-1) is idle and may create an overhead due to monitoring or similar.

$$\frac{N \cdot T(R,L)}{p} \leq S(pR,NL) \leq N \cdot T(1,L) + (p-1) * overhead$$

Minimum = no overhead, perfect parallelisation

Maximum = only one resource is used at any given time (no parallelisation), the rest of the resources (p-1) is idle and may create an overhead due to monitoring or similar.

b. $R = 2,\ L = 4,\ p = 2,\ N = 2\,,\ T(R,L) = 2,\ T(1,L) = 4$

well scaling system:

$$E(pR,L) = \frac{T(R,L)}{p} = \frac{2}{2} = 1$$

$$S(pR,NL) = \frac{N \cdot T(R,L)}{p} = \frac{2 \cdot 2}{2} = 2$$

badly scaling system:

$$E(pR, L) = T(1, L) + (p - 1)\cdot overhead = 4 + 3\cdot overhead$$

$$S(pR, NL) = N\cdot T(1, L) + (p - 1)\cdot overhead = 2\cdot 4 + 3\cdot overhead$$

Result:

An efficient system runs 100% faster if resources are doubled. A non-efficient system stays the same and has additional overhead.

A well scaling system scales 100% if the resources are doubled and the problem stays the same (in our case the time stays the same as also our problem gets doubled). A badly scaled system needs the same time as with one resource and has additional overhead.