

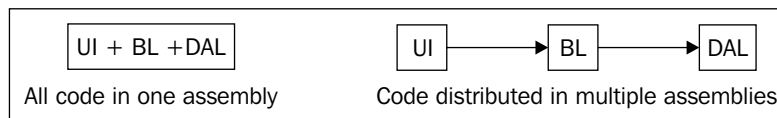
When we talk about n-tier applications, we are referring to medium-size to enterprise-size applications. Many beginner developers have this urge to make every software system they develop an n-tier system, even when the project doesn't need to be tiered at all. For example, it would not be worth developing a simple guestbook application for a personal website on an n-tier architecture as it would take a lot of time and money, besides complicating the simple system for no tangible benefits.

But we need to think beyond layers and 1-tier applications when we deal with applications such as commercial websites with large user bases, medium to large software systems that need built-in interoperability and redundancy, and flexibly-distributed solutions. We need to separate out the business logic and data access code into their own assemblies to make the application further distributed and loosely-coupled in nature.

The parameters described in the following sections can be used to decide whether we want to go for a n-tier system or a simple layered solution.

Performance

Application performance is always a prime consideration when working on any project. The more the code is separated into different assemblies, the slower it becomes. See this diagram here:



The reason for this slow performance is simple. It takes longer for a method in one assembly to call a method in another assembly than it would take if the method was in the same assembly. Some time is needed to refer to the other assembly, and read, find, and execute the required code. This time can be avoided if we have the code in the same physical DLL.

So if we separate the logically-layered code into different physical assemblies, then we will suffer a mild performance hit. I used the word **mild** because modern machines have a lot of computing power, and the performance hit is almost negligible. So if we consider this mild performance hit, then how can an architecture based on n-tier actually increase performance of the application?

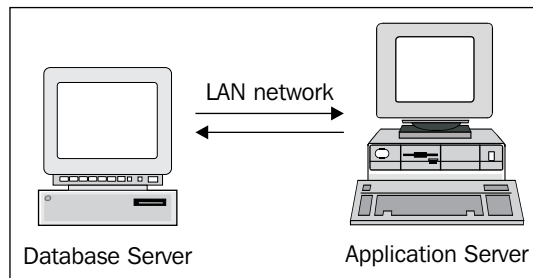
It all depends on the way a single-processor on a machine works. On a single-processor machine, all operations, including the code executing the UI and the business logic, the connection to the database server (which is another application), fetching data, and so on, are handled by a single CPU. So it can handle only one instruction at a time. If the application has a long list of pending operations, then the processor will be hogged and will run slow, causing a bottleneck. But if we can distribute the application load so that we can use multiple processors (on multiple machines), then we can have substantial performance gains. Of course, to reap this benefit, the load itself should be quite high in the first place. If the load is always low, then we will lose more from having distributed tiers "talk" to each other than we will gain from distributing the workload.


So should we put the DAL and BL assemblies on different machines to balance load? The answer is no. The reason being the fact that before deciding to allocate the tiers to their own processors (by putting them on different machines with own CPU), we first need to identify the main load bearing components.

Before we go further, we need to understand the term "load". In web applications, load refers to the quantum of computing power required to serve client requests. In most web based applications, the load is usually handled by:

- A database
- The ASP.NET worker process (`w3wp.exe` in Windows 2003/2008, or `aspnet_wp.exe` in Windows XP)

For large load applications, it is advisable to have the database on a separate, dedicated machine, so that it does not compete with the ASP.NET worker process for CPU cycles. Here is the configuration:



 ASP.NET worker process is an OS-level process which handles the .NET runtime. IIS handles the client requests and passes them on to the ASP.NET worker process. 