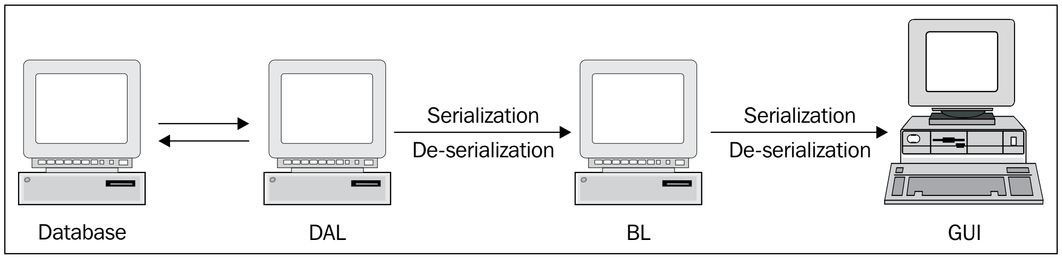
In the previous diagram, the database is now on a different machine and is communicating with the application server (where the ASP.NET worker process runs) across high-speed Local Area Network (LAN). There would be a little latency involved here because data has to go through the LAN to reach the database, but that would be negligible because the LAN would usually be based on a fiber optics network and would be super-fast as both systems are on a local network. But we would gain considerable performance benefits because the database tier now has its own dedicated processor and memory.

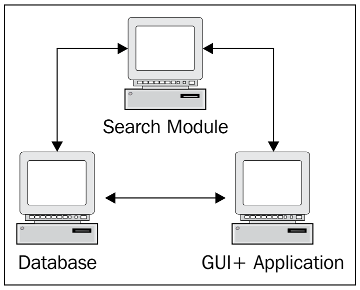
Now, the UI, BL and DAL DLLs execute under the ASP.NET worker process application domain. So they share the same application domain and hence form a single component handling the server load. If we put the BL and DAL assemblies on separate machines then we will suffer a big performance hit, because the assemblies individually don't handle much load. Moreover, a lot of CPU cycles would be wasted in serializing data across cross-application boundaries.

Serialization is the process of converting an object to some persistent medium so that we can transfer it across the network. For example, consider a Customer class object in our OMS application. Say we want to send this object to another computer across the network where another .NET application is expecting it. This object right now is in memory, so we can convert it into an XML string (serialize it) and then transfer this XML string to another machine over the network, where the .NET runtime will catch this XML string and convert it back to an object in memory (de-serialization).

This is how the configuration will look if we put the BL and DAL tiers on different machines:



This serialization process is CPU intensive, and would hurt the performance more because it is actually not reducing any load but adding to it. The reason is that the BL or DAL alone does not handle the load individually. DAL is simply a utility layer that talks to the database and gets data from or sends data to BL. BL processes it and passes it on the GUI. In most systems, it is best to keep these three under the same application domain (worker process) so that we can do away with serialization overheads. There are cases where we might want to put some application components on a different server, for example, imagine a catalog management website with a large number of users. The search catalog part of this website would be heavily used and can be treated as a load-bearing component. In this case, it can be advantageous to move the search into its own machine, and return the results to the application tier using XML, or binary serialization, or similar methods.



As shown in this diagram, we have placed another load bearing component on a separate machine to increase the overall performance of the web application. But if the application is small or does not get many hits, then cross-application serialization will hurt performance, and using such a configuration will be worthless.

These configurations are not possible without an n-tier architecture. But one must know when to use an n-tier architecture and when to go for a much simpler architectural configuration, depending on the actual project needs. There must be a balance between performance and application complexity.

Scalability

Application scalability is another important architectural aspect. Being scalable means being able to handle an increased load in future. For example, a community-based web application architecture should be capable of supporting an increased number of simultaneous users over time. If the number of concurrent users grows rapidly, then it's better to separate the components of your application onto different servers because CPU bandwidth is limited (as explained in the performance section). Components here not only mean the business logic and the data access code, but also accessing other static resources such as images, videos, and so on.

Scalability is closely tied to performance. Sometimes, for small applications such as a guestbook, we design a 1-tier 1-layer architecture, which is good performance-wise, but will not be scalable. We don't need scalability in such small applications as there is no requirement for it. But for bigger applications, a 1-tier configuration will not work, and we will need to break it into an n-tier based architecture so that the application becomes scalable and also more performance-efficient during peak load times.