Part 1

Architecture

Part 1 offers an overview of application server architecture, describing its benefits in the business environment and providing an overview of its fundamental technologies including multi-tiered client/server computing, distributed applications and middleware.

Chapter 1

What Is an Application Server and Why Do I Need One?

Over the past year or so, quite a few software vendors have released packages they call application servers. Inprise, Oracle, BEA, and a number of others all have jumped onto the application server bandwagon, extending their product lines with products that target enterprise computing. So, what exactly is an application server?

This chapter will explore the reasons why application server technology will play an important role in the next generation of enterprise computing. Topics include:

- Two-tiered vs. multi-tiered computing
- Why I chose multi-tiered client/server
- What can an application server do?
- Costs and disadvantages of application servers
- Moving from traditional client/server to *n*-tier computing

Two-Tiered vs. Multi-Tiered Computing

There are quite a few advantages to traditional two-tiered client/server. The database products are very mature with heavy competition to constantly

improve performance and features. Client-side development tools like Microsoft Access, Borland Delphi, and C++ Builder have become so easy to use that much of the code writes itself. Even the networks are easier to install and maintain.

But as most client/server developers soon discover, it is almost too easy. New applications multiply on the server and, with the constantly plunging price of computers, more clients keep coming on board. In no time at all, the server is overloaded. Even after all of the memory slots have been filled, more CPUs have been added, and thousands of dollars have been spent to upgrade the network, the users still complain that response time is too slow.

To solve this problem, the industry is now touting three-tier and *n*-tier (multi-tiered) client/server. The server itself becomes a network of computers that can grow to meet the increasing client demands. Instead of the client software communicating directly with the database server, a middle layer of software, called an application server, provides services to the client (see Figure 1-1). This minimizes the number of connections to the database server and spreads the processing over several computers. It also allows the client software to shrink, because much of the processing is passed to the application server. The client software can now become as simple as a form running on a Web page.

Since so much of the processing moves to the middle layer, building an application server can be a difficult, complex process requiring a whole new set of tools and skills. Every software vendor in the business is rushing to sell middleware tools, but the techniques to use these tools are still in the early stages of development. Many sources of information are available on how the tools have been implemented, their architecture, services, protocols, and how to get them running. But even now, little practical knowledge exists on how to build the software. This book will look at some of the principles and practices that can help software developers make middleware tools solve these business problems.

Why I Chose Multi-Tiered Client/Server

Most of my consulting work revolves around managed healthcare, an industry where large volumes of information are scrutinized against a constantly changing set of business rules. The software must handle large

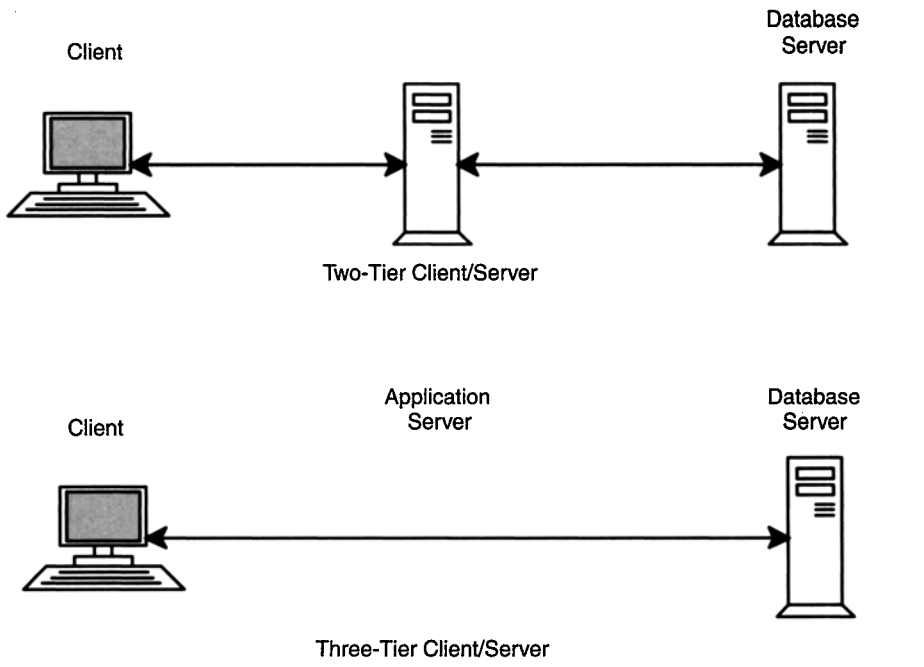

Figure 1-1. *Two tier vs. three tier client/server*

amounts of data efficiently, yet be flexible, because the industry tries to balance ever-increasing costs with demands for quality care while fielding an ever-increasing barrage of government regulations. My work began with mainframe-based systems, but several years ago, I began creating two-tiered client/server systems for small start-ups and medical specialty groups.

As the client/server software began to grow, I found several problems. The first was that response time started to bog down with the number of workstations in use. Upgrading the database server helped, but only until more workstations were added. Next, I found that the development tools worked well for interactive, screen-oriented software development, but lacked the ability to do mainframe style batch processing, working with large amounts of data quickly. Tools like Crystal Reports helped, but these did not provide the flexibility in calculations that were necessary for the application.

The problem that finally led me to investigate multi-tiered computing was business rule processing. As a claim is submitted into the database, a few critical pieces of information (for example, patient, doctor, diagnosis, date of service and medical procedure code) must be checked against a large set of business rules such as:

- Is the member eligible for services?
- Has the service been authorized?
- Is this an appropriate service for the diagnosis?
- Is the doctor allowed to perform these services?
- Is this an appropriate service for the age and gender of the patient?

There are often two or three hundred rules that must be verified before a claim can be paid. Once these rules are checked, the appropriate fee and benefit schedules must be matched to determine how much the doctor should receive, and what portion must be paid by the patient. Try coding this in Visual BASIC or Delphi. It can be done (we've done it), but without some form of back-end processing, there is no way it can be done within the limits of reasonable response time. Our solution was to process the rules off-hours in batch mode, but this limited the ability of the claims processors to get their work done.

With multi-tiered computing, I can begin to offload business rule processing to a separate server and test these business rules as claim information is entered. Many of the data tables can be loaded into streamlined business objects in memory, which will speed up processing. The more complex checks can be run in the background at a lower priority. Batch processing for decision support and reporting can be moved to separate machines where the data can be replicated into a data warehouse application.

What Can an Application Server Do?

In addition to the issues described above, an application server can also solve many other weaknesses of traditional two-tiered client/server computing and provide many new benefits as well. An application server

helps the *system administrator* by providing scalable software that can be spread over multiple machines for better system performance. It helps the *software designer* by providing clearly defined logical boundaries that enable the designers to create business objects that model the business closely. In some ways, software development is also easier, because the code is broken up into smaller, more granular modules and services that are much easier to reuse. Application integration is also much easier, using middleware services to translate data formats and simplify communication between different vendor's machines.

Scalability

The most apparent benefit of multi-tiered client/server is scalability, because the workload is spread among several computers. No matter how much is spent on the latest leading-edge mega-server, there is a finite amount of processing power that any one computer can produce. Spending the same amount of money on several medium-grade servers will generate more computing horsepower and may even cost less.

Where the savings really show is in the incremental costs of upgrades. The mega-server may have some limited upgrade capabilities, but when it maxes out, it has to be replaced with an even more expensive super-mega-server. Not only does the company have to absorb the cost of a new server, it has to write off the old one. With distributed servers, the only cost is the incremental cost of an additional medium-grade server.

Distributed processing

Another advantage is that the databases and application servers can be distributed closest to where the work needs to be done. If order entry is done in San Francisco and production and inventory are done in Atlanta, it makes sense to keep the databases where the majority of the work is done instead of keeping all the data at the corporate office in Chicago. Network traffic will be minimized, because order entry will be done locally in San Francisco, with a much smaller amount of traffic routed between San Francisco and Atlanta to check inventory levels. If Chicago wants management reporting, data can be summarized in San Francisco and Atlanta, then sent in summary form to Chicago.

Distributed processing can also be used to hold local instances of remote data. This will minimize network traffic even more and allow processing even when the remote connection goes down. An inventory item object residing in San Francisco could hold the current number of items in stock and the number reserved by recent orders. Periodically, it would send a message to its counterpart in Atlanta to reserve the items and get a new update of the number actually in stock. If the network goes down, order entry does not have to stop, because the local computer has a close approximation of how many items are available. Once the network comes back up, the update message can correct any discrepancies.

Reusable business objects

An application server is a repository of services and objects that reflect business processes. Since these processes can be described in business language, rather than computer language, it is much easier for the developer to translate business requirements into effective software design. With clearer communication between software developers and business people, the design will come closer to reflecting the real business needs. This results in software that is delivered sooner and costs less to produce.

Once the application server is in place, the objects and services already developed are available for reuse in other applications. Instead of a tightly integrated, closed application, much of the functionality is exposed to the development team. Just as Visual Basic provides a set of GUI components that are used over and over again, most middleware implementations require a common, standardized interface and component model that makes reuse much easier and more cost effective.

Business rule processing

Most two-tiered client/server tools emphasized a data-centric view of software design in which the client software provided a user interface to manipulate information stored on a database server. Almost all processing had to be done on the client side away from the database; this arrangement required additional network traffic. Some processing could be moved to the database server, but this required proprietary stored-procedure languages that were limited to each particular database server vendor.

Application server development stresses business object construction rather than data storage. Each component contains services as well as data, which allows a much wider range of data integrity checks as well as the ability to derive additional information from the data contained in the object. The services can also encapsulate business rules and processes that model the actual business. When an invoice is entered, the invoice object has the intelligence to check the customer object to ensure that credit can be authorized. These rules and processes are performed automatically when a service is requested to store the data.

Cross-platform integration

Since most organizations already have a large inventory of applications in place, the middleware vendors have invested much of their effort into cross-platform integration. The developer does not have to be concerned with translating low-level data formats, byte-order representations or other vendor-specific data. The middleware can also bridge multiple programming languages by using a high level interface definition language (IDL). Once the functions are declared in this language, the IDL compiler will generate translation code in a variety of programming languages. This allows programs in one language to call functions or access objects written in another language even when they are located on a different computer.

Costs and Disadvantages of Application Servers

Although there are many advantages to implementing an application server, the technology is not appropriate for every application. Multitiered development requires a substantial up-front commitment that may not immediately show results. The application server is a complex piece of software that requires a whole new set of skills and tools. Most middleware packages are based on object-oriented design and programming concepts that require a higher level of abstraction and have higher learning curves. Many also rely on component architectures that must conform to rigid new programming standards. Components and modules must also be general enough to allow later reuse. The technology solves many problems but also brings its own set of difficulties.

Long-term commitment

Implementing an application server architecture is a long-term, enterprise scale commitment. This is not the appropriate choice for a project that must be developed in "Internet time" or for a single, limited use application. This is an enterprise architecture that requires new hardware configurations, middleware, programming models, administrative tools, and, most of all, a new way of looking at software development. The first application will not be easy. Much time will be spent in trial and error, evaluating tools, learning the idiosyncrasies of middleware, and creating infrastructure instead of applications. Viewed as a single application, it will definitely not be cost-effective. This technology can only be justified when seen as the first step in building the foundation to a new enterprise architecture.

Middleware acquisition

One or more middleware packages are probably already sitting on your hard disk. Microsoft bundles its Component Object Model architecture (COM and DCOM) with its most recent versions of Windows. Microsoft Transaction Server (MTS) is also making its way onto the Windows NT platform with the release of SQL Server 7. Java development packages provide a simple object request broker (ORB) called RMI (Remote Method Invocation) that is included with the Java Software Developers Kit (SDK) version 1.1 or higher. So why pay for another middleware solution?

Most of these middleware packages are bound to one proprietary platform, but a comprehensive middleware solution must span a variety of computer platforms, programming languages and databases. The choice of middleware depends on current hardware and programming languages that are already in place, as well as future expansion requirements. COM, MTS and RMI are each vendor-, language-, or platform-specific. This may not be a problem if the organization is already standardized on Microsoft or Java platforms, but each may limit future scaleability and growth.

The initial purchase price is also just the beginning of the middleware cost. Any choice must also take into account staff training, hardware and network acquisition, programming, and administration costs. Training and start-up costs can often eclipse the purchase price of even the most expensive middleware package.

New ways to twist the brain

Multi-tiered client/server also requires new ways of thinking about software. Although programming is a fairly abstract ability, object-oriented software design and programming require an even higher level of abstraction. Instead of a single sequential flow of execution, object orientation requires visualizing the interaction of multiple processes running in parallel on several computers at the same time.

Consultants are available to act as guides through the project and to provide training, but at a very high cost. Many tools are also available to help manage the transition, but each of these add acquisition and training costs to the project. Money spent wisely in this area can greatly increase the chances of success, but costs can quickly mount with little benefit if spent in the wrong direction.

The end of the coding cowboy

In "Coding Cowboys and Interdependent Systems," Warren Keuffel and Bryce Carey (Keuffel 1998) make an analogy to the Old West. The cowboys out on the range worked on their own, independent, untroubled by the rest of the world. When the day came that the railroads ran tracks across the range, that independent spirit suddenly changed. If the cowboys could not coordinate their track-crossing schedule with the railroad's standard of time, the cows would be caught on the cowcatcher. Until recently, programmers could make up their own rules, too; but with the advent of the Internet and electronic commerce, software developers must now adhere to common standards or the bits they herd will be roadkill too.

Components and middleware architectures require much more discipline and standardization. Objects and components must conform to rigid standards and implement tightly defined interface methods. Much of this is provided by the programming tools, but design and structure must conform to these standards or the application will not run. Developers must also work closely together to ensure that interfaces and communication paths match and that objects and modules are coordinated to each other.