Distributed .NET Programming in C#

TOM BARNABY

Apress™

Distributed .NET Programming in C#
Copyright ©2002 by Tom Barnaby

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-039-2

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: Gordon Wilmot

Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Project Manager: Alexa Stuart

Copy Editor: Ami Knox

Production Editor: Kari Brooks

Compositor: Susan Glinert Stevens

Artist: Cara Brunk, Blue Mud Productions

Indexer: Valerie Robbins

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710.

Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 1

The Evolution of Distributed Programming

"It's like, how much more black can this be?
and the answer is none. None more black."

—Nigel Tufnel (*This Is Spinal Tap*)
speaking on the state of software development.

TODAY, BUZZWORDS LIKE enterprise programming, distributed programming, n-tier, and scalability are floated in nearly every product announcement. So before tackling the nuances of distributed development in .NET, this chapter attempts to de-marketize such terms by applying real meaning and context to these ubiquitous words. Also, while this book is primarily a nuts-and-bolts "how to" guide, it is important to have a clear understanding of why you should distribute applications and how to design a distributed application. To this end, this chapter offers five principles to help guide your distributed development in .NET or any other platform.

Finally, in order to drive home the principles of distributed programming, this chapter takes a lighthearted look at past distributed development models and the reasons they were replaced by new models. As you will see, this goes a long way towards explaining why Microsoft created a new development platform called .NET to replace COM.

Overview of Distributed Programming

What is distributed programming? Now, there is a question few dare to ask. The term is so common today that some may be embarrassed to question its meaning. Rest assured there is no need to be. I routinely ask my students to define it, and rarely do I get the same answer.

Distributed programming is characterized by several distinct physical components working together as a single system. Here, "distinct physical components" could mean multiple CPUs or, more commonly, multiple computers on a network. You can apply distributed programming to a wide variety of problems, from predicting the weather to purchasing a book. At its heart, the premise of distributed

programming is this: if one computer can complete a task in 5 seconds, then five computers working together in parallel should complete the task in 1 second.

Of course, it is never quite that easy. The problem is the phrase "working together in parallel." It is difficult to get five computers on a network to cooperate efficiently. In fact, the application software must be specifically designed for this to be effective. As an analogy, consider a single horse pulling a carriage. A horse is a powerful animal, but, in terms of power-to-weight ratios, an ant is many times stronger (we will just assume ten times stronger). So, if I gather and harness enough ants to equal the mass of the horse, I can move ten times the amount of material in the carriage. A perfect example of distributing the workload, right? The calculations are reasonable, but hopefully you are chuckling at the ludicrous vision of millions of ants with tiny harnesses pulling together.

Layering an Application

As demonstrated with the horse-vs.-ant analogy, distributed computing raises the issue of coordinating the work of several computers. There is also the issue of decomposing the application into tasks that can be distributed. Luckily, here you can draw on the lessons learned from earlier applications. Over the years, it has become clear that most business applications consist of three primary sets of logic: presentation, business, and data source.

- **Presentation logic**. This is the part of the application that the end users use to enter orders, look up customer information, and view business reports. To the user, this *is* the application.

- **Business logic**. This is the heart of the application and where developers spend most of their time and effort. It contains the business rules that define the way the business is run. For example, business rules specify when customers receive discounts, how shipping costs are calculated, and what information is required on an order.

- **Data source logic**. This is where orders, customer information, and other facts are saved for future reference. Luckily, database products such as SQL Server and Oracle take care of most of the work. But you still have to design the data layout and the queries you will use to retrieve the data.

The first design point of any nontrivial business application should be to partition these sets of logic into distinct layers. In other words, you should not mix business logic code with the presentation logic code. Do not take this to mean, however, that each layer must run on a separate machine, or in separate process. Instead, code from one layer should only interact with that in another layer through

a well-defined interface. Typically, the layers are physically implemented within separate code libraries (DLLs).

The Five Principles of Distributed Design

Layering allows you to change the implementation of one layer, without affecting another layer. It also provides the flexibility to physically separate the layers in the future. However, as the upcoming sections show, the decision to execute each layer in a separate process or machine should not be made lightly. If you do decide to distribute the layer, you must design it specifically for distribution. Confusing the issue even more is the fact that some of these design tactics contradict classical object-oriented principles. To help clarify the issues, this section describes several principles you can use to effectively distribute an application and why you should use them.

Principle 1: Distribute Sparingly

This may seem like a surprising principle for a book about distributed programming. However, this principle is based on a simple, undeniable fact of computing: invoking a method on an object in a different process is hundreds of times slower than doing the same on an in-process object. Move the object to another machine on a network, and the method call can be another ten times slower.

So when should you distribute? The trite answer is only when you have to. But you are probably looking for a little more detail, so let's consider a few examples, starting with the database layer. Typically an application's database runs on a separate, dedicated server—in other words, it is distributed relative to the other layers. There are several good reasons for this:

- Database software is complicated, expensive, and typically requires high-powered hardware. Therefore, it isn't cost effective to distribute many copies of the database software.
- A database can contain and relate data shared by many applications. This is only possible, however, if each application server is accessing a single database server, rather than their own local copy.
- Databases are designed to run as a separate physical layer. They expose the ultimate "chunky" interface: the Structured Query Language (SQL). (See Principle 3 for details on chunky interfaces.)

Therefore, the decision to distribute the data source logic is typically made the moment you decide to use a database. However, the decision to distribute the presentation logic is a little more complex. First of all, unless all the application users walk up to a common terminal (like an ATM), then some aspect of the presentation layer must be distributed to each user. The question is how much. The trend lately, of course, is to execute the bulk of the logic on the server and send simple HTML to client Web browsers. This is actually in keeping with the principle to distribute sparingly. However, it also requires each user interaction to travel to the server so that it can generate the appropriate response.

Before the proliferation of the Web, it was more common to execute the entire presentation logic on each client machine (in keeping with Principle 2). This provides faster interaction with the user since it minimizes round trips to the server, but also requires user interface updates to be deployed throughout the user base. In the end, the choice of which client you use has little to do with distributed design principles, and everything to do with the desired user experience and deployment issues.

So the data logic almost always executes on a separate computer, and the presentation layer frequently does. That leaves us with the business layer, and the most complex set of issues. Sometimes, the business layer is deployed to each client. Other times it is kept on the server. In many cases, the business layer itself is decomposed into two or more components. Those components related to user interface interaction are deployed to the client, and those related to data access are retained on the server. This holds to the next principle, which is to localize related concerns.

As you can see, you have many distribution options. When, why, and how you distribute is driven by a variety of factors—many of which compete. So the next few principles offer further guidelines.

Principle 2: Localize Related Concerns

If you decide or are forced to distribute all or part of the business-logic layer, then you should ensure that those components that frequently interact are kept close together. In other words, you should localize related concerns. For example, consider the e-commerce application shown in Figure 1-1. This application separates `Customer`, `Product`, and `ShoppingCart` components onto dedicated servers, ostensibly to allow parallel execution. However, these components need to interact many times while adding a product to the shopping cart. And each interaction incurs the overhead of a cross-network method call. Therefore, this cross-network activity

will easily eclipse any parallel processing gains. Multiply this by a few thousands users, and you have a scenario that can devastate performance. Relating this to the earlier horse and carriage analogy, this is the equivalent of harnessing each leg of the horse rather than the entire horse.

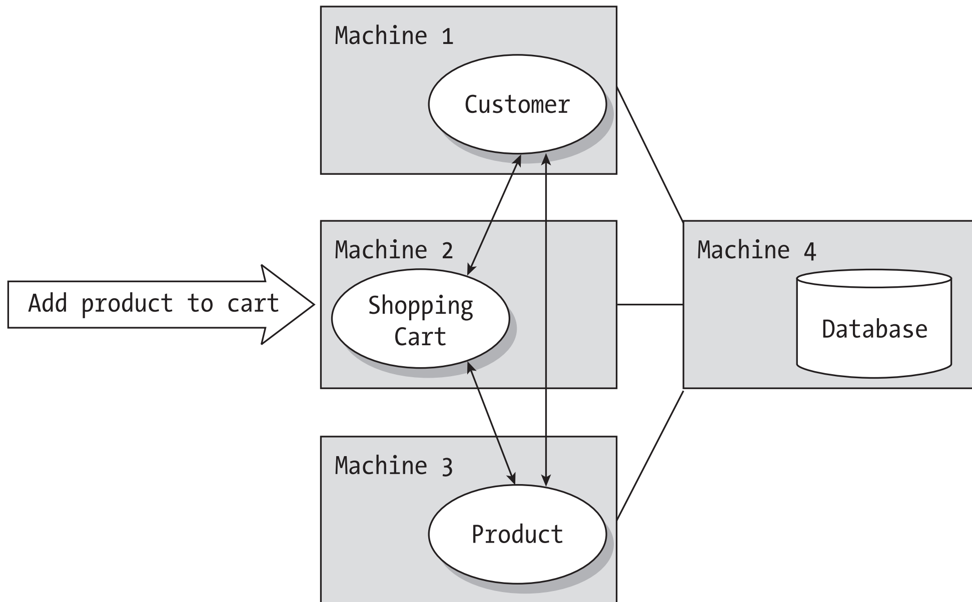

Figure 1-1. How NOT to design a distributed application

So how can you leverage the power of distributed programming, namely parallel processing, while still localizing related concerns? Buy another horse. That is, duplicate the entire application and run it on another dedicated server. You can use load balancing to route each client request to a particular server. This architecture is shown in Figure 1-2. Web-based applications often use this model by hosting the identical Web site on several Web servers, a setup sometimes referred to as a *Web farm*.

Duplicating and load balancing application servers is a great way to increase the capacity, or scale, of an application. You do need to very conscious, however, of how you manage state. For more details, see Principle 4.

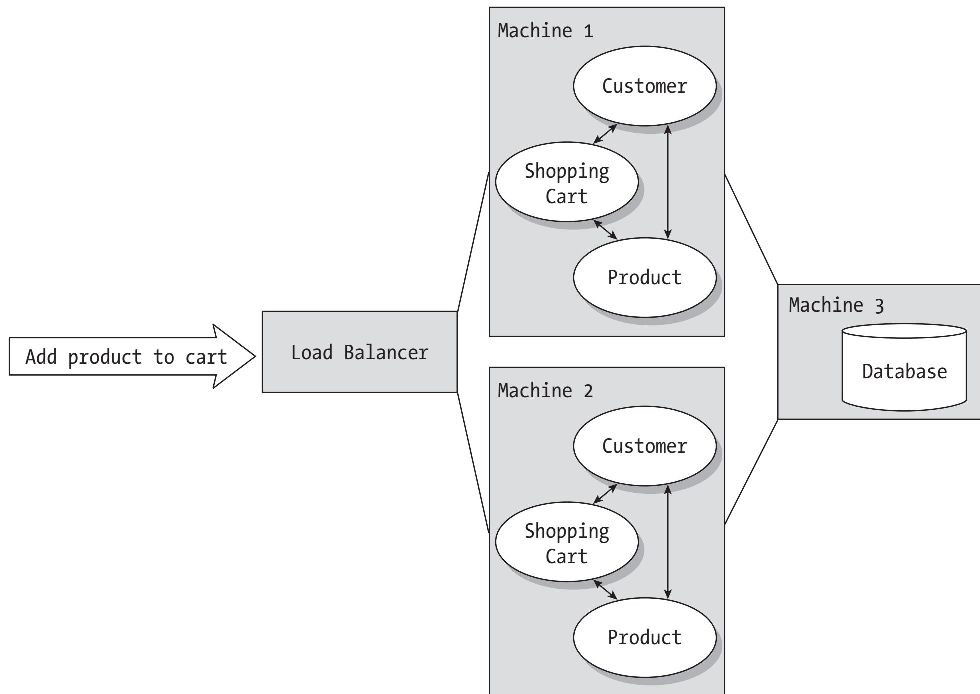

Figure 1-2. How to distribute an application

Principle 3: Use Chunky Instead of Chatty Interfaces

One of the philosophies of object-oriented programming is to create objects with many small methods, each focused on a particular behavior. Consider the following `Customer` class.

```
class Customer
{
   public string FirstName()
   { get; set;}
   public string LastName()
   { get; set;}
   public string Email()
   { get; set;}
   // etc. for Street, State, City, Zip, Phone ...

   public void Create();
   public void Save();
}
```

This implementation would garner nods of approval from most object-oriented experts. My first reaction, however, is to question where this object will run relative to the calling code. If this `Customer` class is accessed strictly in process, then the design is fine—correct, even, by most standards. However, if it is called by code executing in another process or on another machine, now or in the future, then this is a dreadful design. To see why, consider the following code and imagine it running on a client in New York while the `Customer` object runs on a server in London.

```
static void ClientCode()
{
   Customer cust = new Customer();
   cust.Create();
   cust.FirstName = "Nigel";
   cust.LastName = "Tufnel";
   cust.Email = "ntufnel@spinaltap.com";
   // etc. for Street, State, City, Zip, Phone ...

   cust.Save();
}
```

Again, if the `Customer` object were in the client's process, this example would pose no problems. But imagine each property and method call traveling over the Atlantic Ocean, and you can see serious performance issues.

This `Customer` class is a classic example of a class with a *chatty interface*, or, in more highbrowed terminology, a *fine-grained interface*. In contrast, objects that are accessed even occasionally by out-of-process code should be designed with *chunky interfaces,* or *course-grained interfaces*. Here is a chunky version of the `Customer` class.

```
class Customer
{
   public void Create(string FirstName, string LastName, string Email,
                      // etc for Street, State, City, Zip, Phone ...
                     );
   public void Save(string FirstName, string LastName, string Email,
                    // etc for Street, State, City, Zip, Phone ...
                   );
}
```

Granted, this is not nearly as elegant as the first `Customer` class. But while the former earns you object-oriented accolades, the latter protects your job when your employer's Web site suddenly becomes the next Amazon.com and needs to scale to support the influx of new customers.

As a slight tangent, I should mention that it is possible to simplify the chunky `Customer` class interface. Instead of passing each piece of customer data as a separate parameter, you can encapsulate the data into a custom class and pass it instead. Here is an example.

```
[Serializable] // <-- Explained in Chapter 2!
class CustomerData
{
   public string FirstName()
   { get; set;}
   public string LastName()
   { get; set;}
   public string Email()
   { get; set;}
   // etc for Street, State, City, Zip, Phone ...
}

class Customer
{
   public void Create(CustomerData data);
   public void Save(CustomerData data);
}
```

At first glance, this example simply moves that chatty interface from the `Customer` to the `CustomerData` class. So what is gained? The key lies in the `Serializable` attribute just above the `CustomerData` class definition. This tells the .NET runtime to copy the entire object whenever it is passed across a process boundary. Therefore, when the client code calls the `CustomerData` properties, it is actually accessing a local object. I will discuss serialization and serializable objects further in Chapter 2 and Chapter 3.

Principle 4: Prefer Stateless Over Stateful Objects

If the last principle simply annoys object-oriented purists, this principle infuriates them. Measuring against strict object-oriented definitions, the term *stateless object* is an oxymoron. However, if you want to take advantage of the load-balanced architecture shown previously in Figure 1-2, you need to either manage state very carefully in your distributed objects or not have state at all. Keep in mind that this principle, like the previous one, only applies to those objects that are on a distributed boundary. In-process objects can live happy, state-filled lives without jeopardizing the application's scalability.

The term *stateless object* seems to cause a lot of confusion among developers. So allow me to define it as succinctly as possible: a stateless object is one that can be safely created and destroyed between method calls. This is a simple definition, but it has many implications. First, note the word "can." It is not necessary for an application to destroy an object after a method call for it to be stateless. But if the application chooses to, it can destroy the object without affecting any clients. This characteristic does not occur for free. You have to specifically implement your class such that it does not rely on instance fields lasting beyond the invocation of a publicly exposed method. Because there is no reliance upon instance fields, stateless objects are predisposed to chunky interfaces.

Stateful objects negatively affect scalability for two reasons. First, a stateful object typically exists on the server for an extended period of time. During its lifetime, it can accumulate and consume server resources. Thus it prevents other objects from using the resources even if it is idling and waiting for another user request. Although some have pointed to memory as the key resource in contention, this is really a relatively minor consideration. As Figure 1-3 shows, a stateful object that consumes scarce resources such as database connections are the true culprits.

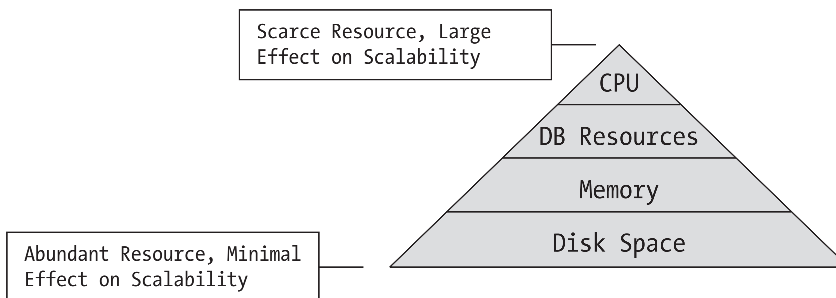

Figure 1-3. The relative quantity of computer resources

The second reason stateful objects negatively impact scalability is that they minimize the effectiveness of duplicating and load balancing an application across several servers. Consider the scenario in Figure 1-4.

Think of Figure 1-4 as a snapshot of the application at a certain point in time. At some point prior to this snapshot the system was under a heavy load. However, several clients have since disconnected, leaving only three. Unfortunately, this application uses stateful objects, and all three objects were created on Server A during the period of heavy load. Now, even though Server B is completely idle, requests from the existing clients must be routed to the heavily taxed Server A because it contains the client state. If this application used stateless objects, however, the load balancer could direct each client request to the server with the lightest load, regardless of which server was used before.

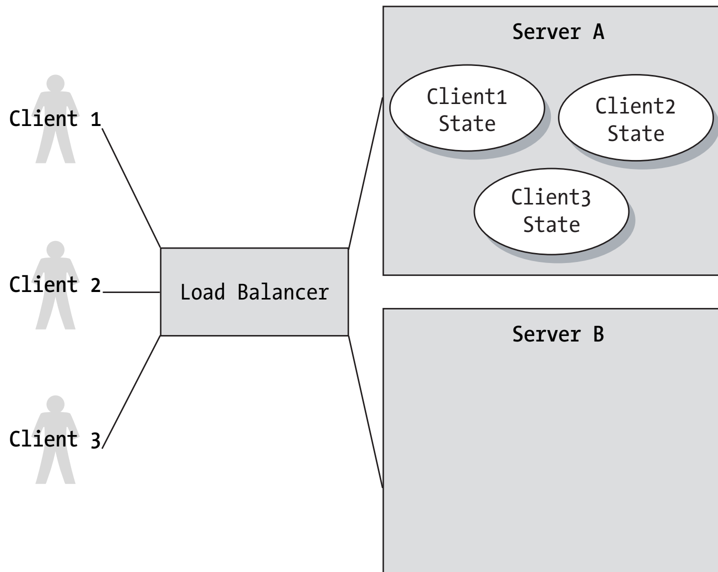

Figure 1-4. Stateful objects don't work well in load-balanced scenarios.

With some clever caching, session management, and load balancing, you can avoid or at least minimize the issues that come with using stateful objects. This is why Principle 4 states that stateless objects are preferred, not required. I also want to reiterate that this principle applies only to objects exposed to code executing in another process or on another machine.

Principle 5: Program to an Interface, Not an Implementation

Since the previous two principles directly contradict typical object-oriented practices, it may seem as though object-oriented programming has no place in distributed programming. I'm not trying to suggest that at all. Rather, I am suggesting that certain object-oriented principles, namely chatty interfaces and stateful objects, should not be applied to objects living on the distributed boundaries of an application.

Other object-oriented principles translate extremely well into the distributed world. In particular, the principle of programming to an interface rather than an implementation resonates within the universe of distributed programming. The issues this solves do not relate to performance or scalability. Instead, interfaces provide easier versioning and thus less frequent and less problematic deployment.

Given that COM was purely interfaced based, the move to .NET has caused some speculation that interface-based programming is also falling out of favor. This is not true. Although .NET allows direct object references, it also fully supports interface-based programming. And, as you will learn in Chapter 5, interfaces provide a convenient way to deploy type information to a client.

Defining Scalability

Throughout this section, I have been using the term *scale* or *scalability*. Since this is yet another nebulous term used far too often in product announcements, it may be helpful to study exactly what this means.

First, although scalability is related to performance, they are not the same thing. Performance is a measure of how fast the current system is. Scalability is a measure of how much performance improves when you add resources, such as CPUs, memory, or computers (see Figure 1-5).

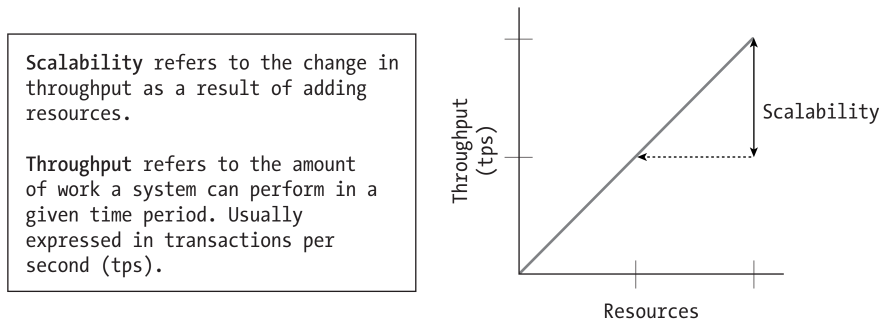

Figure 1-5. Scalability is related to performance.

There are two types of scaling:

- **Vertical scaling (scaling up)** occurs when you remove slower hardware and replace it with new, faster hardware. For example, moving from a Pentium 500 to a Pentium 1G is scaling up. For poorly designed applications, this is often the only way to scale. However, it is typically more expensive and exposes a single point of failure.

- **Horizontal scaling (scaling out)** occurs when you add an additional, load-balanced server to the existing application. This protects your current hardware investments and provides a failover if the other server goes down. While this is far cheaper in terms of hardware, the application must support the five principles discussed earlier in order to scale horizontally.

Some techniques that optimize performance in the short run diminish scalability in the long run. As an example, take a look at Figure 1-6, which compares two applications. The first optimizes performance at the expense of scalability. The second optimizes scalability. At first, the performance-optimized application performs well, because it squeezes every ounce of throughput from the current hardware. However, as additional hardware is added, the inflexibility necessitated in this first application starts to take its toll. Soon it plateaus, while the scalable application continues to rise in throughput.

One classic example of a performance maximizing, but scalability limiting technique is the use of the ASP Session object to cache per-user information. For a Web site with just a single server, this technique increases application performance since it minimizes the number of times the application must call the database. However, if the popularity of the Web site outgrows the capacity of the single server, the typical reaction is to add another Web server. Unfortunately, using the ASP Session object violates the principle to prefer stateless objects. Therefore, it cannot scale out.

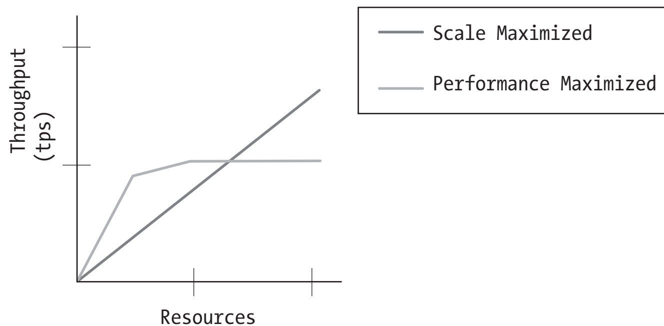

Figure 1-6. A performance vs. scalability optimized application

A Short History of Distributed Programming

The five principles of distributed programming did not come easily. These were established through years of pioneering work, ever-changing technologies, and thousands of failed projects. To better appreciate these principles and to define some other common distributed programming terms, let's review some of the history of distributed computing.

Centralized Computing

In the beginning, applications were built around a central mainframe computer. By their nature, mainframes are large, expensive, and proprietary. The mainframe manages and controls all aspects of the application including business processing, data management, and screen presentation. Typically, a user interacts with the mainframe application using a *dumb terminal*, which consists of just a screen, a keyboard, and a wire to the mainframe. Because the terminal has no processing power, it relies on the mainframe for everything, including presentation. The user interfaces are simple character-based screens that in many cases require weeks of training for users to become proficient.

Centralized computing is the antithesis to distributed computing. Yet many recent trends mimic some of the concepts. For example, the dumb terminal is really the ultimate *thin client*. And it was only a few years ago that some computer vendors were promoting the use of cheap computers called *Web appliances* that had limited hardware and a full-time Internet connection. These trends highlight the primary benefit of centralized computing: low deployment costs. Because the entire application, including presentation logic, is contained within one machine, applying updates and installing new versions is quick and easy. Furthermore, as soon as the update is applied, all users can immediately use the new, improved application.

In spite of the deployment advantage, centralized computing suffers from many problems, including these:

- The entire processing burden, including data access, business logic, and presentation logic, is placed on one computer.
- Monolithic applications are extremely difficult to maintain because of their sheer size.
- Their proprietary nature makes integration with other applications on other platforms difficult.

Eventually, many companies replaced mainframes with cheaper minicomputers, which often ran a version of the Unix operating system instead of a proprietary operating system. The applications, however, were still centralized because it was still too expensive to outfit users with hardware other than dumb terminals. The availability of relatively low-cost minicomputers, however, did portend the coming of the client/server paradigm and the beginning of distributed programming.

Two-tier Client/Server Architecture

As hardware became cheaper, it became feasible to provide users with personal computers, which were far more powerful than dumb terminals. In fact, early PCs had enough power to handle all or at least significant portions of the processing load. Most importantly, these PCs could provide users with graphical user interfaces that were more intuitive than the text-based interfaces of dumb terminals. The client/server model, in all its forms, tries to leverage the computing power of the PC. In other words, part of the load is distributed to the PC. This frees up processing cycles on the mainframe while providing the user with a more aesthetic and intuitive interface, which can drastically reduce user training costs.

Early client/server systems were two-tiered. In this architecture, the processing is spread across two machines: a client and a server. The client machine typically executes both the presentation and business logic, while the server machine provides access to the data. The server machine is usually a dedicated server running a relational database management system (RDMS), such as Oracle or SQL Server. On the client side, tools like Visual Basic opened the world of Windows user-interface development to mere mortals, making it possible for businesses to create custom applications for their employees. Indeed, tools like Visual Basic were so effective in increasing developer productivity that a new development philosophy was created, termed *rapid application development* (RAD).

In the late ’80s through the early ’90s, companies rushed to adopt the two-tier architecture. It was cheap, applications could be built quickly, and users were happy with the flashy new interfaces. It wasn’t too long, however, before the industry started to discover the disadvantages of this architecture:

- Two-tier tools and culture promote RAD techniques. However, its propensity to intermix business and presentation logic on the client creates maintenance nightmares as the system evolves.

- On a related note, updates to business and presentation logic must be deployed throughout the user base, which may include thousands of employee computers. In addition to deploying application updates, you must also consider updates to database drivers, network stacks, and other third-party components. In general, deployment is a huge time- (and money-) consuming effort.

- If the application accesses data from several data sources, then specialized client-side logic is required. This even further complicates the previous issues.

- It is impossible for clients to share scarce resources such as database connections. Because it can take several seconds to establish a database connection, two-tier clients typically open connections early and hold them for the duration of a session. So a database licensed for 20 concurrent connections can only serve 20 client applications, even if many of them are idling.

In spite of these disadvantages, two-tier client/server systems perform fine for small applications with few concurrent users. However, an application and its user base tends to grow over time, so if you decide on a two-tier architecture for the current project iteration, make sure the implementation does not prevent an easy transition to a three-tier architecture. The best practice, in this case, is to logically separate the business layer from the presentation and data layers.

Three-tier and N-tier Client/Server Architecture

A popular adage in computer science is that any problem can be solved with another level of indirection. In the '90s, this philosophy was applied to two-tiered architecture to solve its, by now, well-known issues. Due to the additional level of indirection, this new architecture was dubbed *three-tier architecture*.

In two-tier architectures, the business layer is rarely implemented as a separate logical entity. Instead, it is mixed within the presentation logic or within the database as stored procedures. In three-tier computing, the business logic becomes a first-class citizen. At a minimum it is logically separated from the presentation and data layers, and most times it is physically separated and hosted on a dedicated server called an *application server*. If an application server hosts the business logic, then many clients can connect to the server and share the business logic.

There is some disagreement over the exact definition of n-tier computing. Some say that adding any additional tier to the three-tier model, such as a Web-server tier, constitutes n-tier development. Others equate it to the partitioning of the business logic across many application servers. Sometimes each application server is dedicated to a particular aspect of the business process—for example, a dedicated customer management server or a dedicated order entry server. Complicating issues even more are those who contend that any logical layer constitutes a tier. It is pointless to argue all the semantics. So, in the end, it is best to think of n-tier architecture as any distributed design consisting of three or more tiers.

An n-tier architecture provides the following benefits:

- Clients are thinner because they do not contain business logic. This makes deployment and maintenance far easier, because updates to the business logic need only be applied to the application server. This is especially noteworthy given that the business-logic layer tends to be the most dynamic layer.

- Clients are isolated from database details. The application server can coordinate several different data sources and expose a single point of access for clients.

- N-tier programming promotes disciplined partitioning of the application layers and communication between layers through well-defined interfaces. In the long term, this provides easier maintenance since you can update a layer's implementation without changing its interface.

- N-tier applications can scale horizontally. That is, if designed correctly, the business logic can be duplicated and distributed to several load-balanced application servers. You can add more servers as user demands increase.

- The application server can pool scarce enterprise resources and share them among many clients.

The last point in this list deserves a little more explanation, since I feel it is the most important benefit of n-tier programming. The canonical example of a scarce resource is a database connection. Using the earlier scenario where you have a database license for only 20 concurrent connections, an application server can open all 20 connections and use them to fulfill incoming client requests. Since client applications have a large amount of "think time" between requests, the application server can effectively handle hundreds of clients with the same 20 connections. While one client is processing the results from its last request (that is, "thinking"), the application server can use the open database connection to serve another client's request.

Although these advantages are significant, n-tier applications have a key disadvantage: they are extremely difficult to implement. Furthermore, bad design decisions at critical points can cripple an n-tier application, making it perform and scale no better than the two-tier application it replaced. Many of the issues surrounding n-tier development, however, are shared across all n-tier implementations. These include database connection management, thread pooling, security, and transaction monitoring. So it wasn't long before software vendors were shipping products that simplified these tasks, allowing developers to focus on the business problem rather than the infrastructure. Microsoft in particular has been very active in this marketplace, providing technologies such as COM, DCOM, MTS, COM+, and finally .NET. These technologies are explained more in the next section.

The Web Architecture

Obviously, the Web has played a key role in distributed programming since the mid '90s. Ironically, the Web began very modestly with Web servers transmitting static HTML to browsers. Everything about the Web was designed to be simple. The network protocol, HTTP, is a simplification of TCP/IP. HTML is a simple implementation of SGML. Web browsers (at least the early ones) simply render text-based HTML graphically. And Web servers (at least the early ones) simply listen on port 80 for incoming HTTP requests and send back the requested HTML document. The success of the Web, however, lies in this original state of simplicity. Because of the simplicity of HTTP, HTML, and browser software, Web browsers were soon ubiquitous. This allowed information contained in static HTML documents to be delivered and aesthetically displayed to users regardless of their chosen hardware or operating system.

My how things change. Today's Web browsers are anything but simple. In addition to rendering HTML, they can interpret and execute embedded script code that responds to user action on the page. They expose complicated object models and can host binary components via plug-ins or ActiveX technology. Not all browsers support the more advanced technologies, but even the least-common-denominator level of support is significantly more complex than that of the original browsers.

Web servers have also increased in complexity, to the point where the Web server has become the application server. Today's Web servers can host server-side business logic, access databases, validate security credentials, and integrate with transaction monitors such as COM+. Unlike the application server in the n-tier model, however, a Web server does more than host the business logic; it also builds the user interface by generating a mixture of HTML and embedded client-side script and sending it to the browser.

Interestingly, in this architecture the presentation, business, and data logic are all server side. In this respect, it is similar to the centralized model, so I have been known to call Web browsers "glorified dumb terminals." This does not endear me to my Web programming coworkers, however, who point out how much more interactive and pleasing browser-based interfaces are over dumb terminals. Furthermore, if you have updates to any part of the application, including the user interface, you need only make the update on the server side. Therefore, Web architectures enjoy all the benefits of the n-tier model, plus the ease of client deployment enjoyed by the centralized model.

However, the Web model not only suffers from the same issues as n-tier applications, it tends to exacerbate them. In a traditional (that is, non–Web-based) n-tier application, you typically know how large the target user base is and how many concurrent users to expect. However, the ease of deployment and omnipresent browsers makes it feasible to expose a Web application to users across the entire

organization, nation, or world. Therefore, the load on a Web application is unpredictable; it may serve anywhere from a few to thousands of concurrent users. This makes scalability a much higher priority in Web applications designed for public consumption. This level of scalability can only be achieved with careful design.

The other issue with Web architectures is the reliance on the browser as the client. Today's advanced browsers allow a browser-based user interface to be *almost* as rich and interactive as the traditional fat client. However, when implementing a sophisticated browser-based user interface, you must contend with the following issues:

- Although browsers make it easy to create simple user interfaces, it is difficult to create really sophisticated presentations. User interface idioms that are easy to implement in Visual Basic, for example, can be extremely difficult using DHTML and JavaScript.

- Creating a sophisticated user interface often requires techniques that are browser dependent. For example, only Internet Explorer can host ActiveX controls. This minimizes one of the key advantages of Web architectures: its ability to reach users regardless of platform or browser.

- Even if only a simple interface were required, it takes time for fat client developers to become accustomed to the statelessness of the Web and how data must be passed from page to page.

In spite of these challenges, many IT departments have implemented the Web architecture for internal, or intranet, applications used by company employees. Though some of this was due to hype and mindless marches to "Web enable" applications, it also demonstrates how serious the issue of deployment is. IT groups are willing to tackle the difficulties of browser-based interface development because it saves deployment costs and its myriad headaches.

Microsoft and Distributed Computing

Given that Microsoft was the primary software vendor for PC applications, it follows that the company also played a major role in client/server computing. Initially, Microsoft tools and technologies were focused on the client tier, including Visual Basic, Access, and FoxPro. With the release of Windows NT, however, Microsoft turned its attention to the server side, providing various technologies that ease n-tier application development.

Throughout the last decade, COM was the foundation for nearly all of Microsoft's technologies. .NET, however, completely replaces COM and therefore marks a radical departure for the company and the millions of developers (like you and

me) who use its tools. Still, it is instructive to look back through the lineage that begat Microsoft's current .NET technology.

The Era of PC Dominance

Since Microsoft was a pioneer in early PC software and development tools, the company was quick to promote the benefits of two-tier client/server development. It was during this era that the Windows operating system attained dominance in the PC market. Furthermore, Visual Basic made it easy for almost anyone to develop a Windows user interface and interact with an RDMS like Oracle or SQL Server.

At the same time, Microsoft was refining its Dynamic Data Exchange (DDE) technology, looking for more flexible ways to share data between applications, in particular between applications in its Office suite. Out of this work came Object Linking and Embedding (OLE), a technology that is long dead but whose legacy (and acronym) continues to survive. OLE is worth mentioning here because Microsoft soon realized that the problems it solves transcend spreadsheets and word processors, and could be used as the foundation for a new style of programming called *component-based programming*. Which brings us into the next era.

The Age of Enlightenment

Two major shifts occurred to mark what I call the "age of enlightenment." First, there was the move to component-based programming, where monolithic applications were decomposed into smaller cooperating binary components. This philosophy fit naturally into the world of distributed three- and n-tier applications and built upon accepted object-oriented principles. To be effective, however, the component model required technologies that made it easy for separate components to interoperate, that is, to call methods on and share data with objects residing within another component. When you boil it down, this is exactly what OLE did. Therefore, Microsoft refined OLE to create the Component Object Model (COM), which soon became the primary technological enabler in the Microsoft world for the rest of the decade.

COM is partly a specification. Developers can write COM components using any language that can create binary images matching the COM specification. In theory, COM components built in different languages can interoperate. In practice, however, C++ COM developers must take care to only expose types that can be consumed by less powerful languages such as JavaScript or VBScript. Visual Basic was reengineered between versions 3 and 4 in an attempt to make COM development more accessible to the average programmer—and was a huge success.

When Microsoft pushed into the server-side market, it extended COM to create Distributed COM (DCOM). DCOM provides the infrastructure necessary for COM components to interoperate over a network as if they were on the same machine. With DCOM, you can take the COM business objects that were deployed to each client machine and move them to a central server without changing the client layer or the business layer's code. Unfortunately, this describes exactly what companies did in many cases—but that story will have to wait until the next section.

The second monumental shift in the age of enlightenment was the emergence of the Web. Unlike component technology, Microsoft was relatively slow to embrace Web technology. It wasn't until the Web was clearly here to stay that the company starting searching for a way to enter the market with a bang. It didn't have to look far. Microsoft took what was then a little-known, low-level technology with a boring name and repackaged it with a sexy new name: ActiveX. Of course, the little-known technology I'm referring to is COM. Regardless, Microsoft hit the Web running with a volley of "Active" technologies: ActiveX Controls, ActiveX Documents, Active Server Pages (ASP), and Active Data Objects (ADO). All of these were built from or relied upon COM in one way or another.

Microsoft also provided Web browser and server implementation in the form of Internet Explore (IE) and Internet Information Server (IIS), respectively. IE was unique among browsers in that it could host ActiveX Controls and ActiveX Documents. Ironically, although IE came to dominate the browser market, these client-side ActiveX technologies never gained much traction.

IIS, on the other hand, had to fight for every bit of its modest Web server market share. Yet ASP, a server-side scripting engine exclusively available with IIS, gained broad acceptance with IIS users. Before IIS, Web servers interacted with applications through the Common Gateway Interface (CGI) or through proprietary Web server APIs. In both cases, the Web server forwards an incoming HTTP request to some application that generates a dynamic response. Typically, the response is static HTML mixed with a small amount of dynamic HTML. This requires a language that allows quick-and-easy text manipulation. Therefore, CGI popularized the use of scripting languages, like Perl, that provide powerful string and regular expression utilities.

As with other Web servers, IIS exposes an API. The IIS API is called ISAPI, and any application that uses this API is called an ISAPI extension. Soon after its introduction, IIS shipped with a useful ISAPI extension called ASP. With this, Web page developers do not have to use CGI or understand ISAPI to write dynamic content. Instead, you can write static HTML as normal, and insert some script code where dynamic HTML generation is necessary. When IIS processes the page, the ASP ISAPI extension interprets any embedded script code and inserts the generated HTML from the script into the stream of HTML flowing back to the browser. Thus, from the browser's perspective, all of the HTML comes from a simple static page.

ASP effectively turned dynamic Web page development upside down. Whereas CGI forces programmatic code with a lot of HTML embedded within, ASP

pages contain static HTML with some programmatic code (VBScript or JavaScript) embedded within. This proved to be a far more productive way to build dynamic Web content. However, the interpreted nature of ASP slows down complex calculations, and the mixture of script and HTML within one page tends to get overly complex and difficult to maintain. Enter COM, again. ASP script code can create COM objects and invoke their methods. Therefore, you can compile complex business rules and data access logic into COM components and access them from ASP. This minimizes the scripting code in the ASP page and keeps it focused on a single task: the dynamic creation of HTML.

Thanks to (D)COM, some creative thinking, and endless marketing, Microsoft appeared ready to dominate on the server side. But things did not quite work out that way, as explained in the next section. Before moving on to the next section, it is important to point out again that, during this era, COM technology was woven into the entire Microsoft architecture, and this practice continued into the next period, which I call . . .

The Days of Disillusionment

It was only a matter of time before Microsoft and its customers learned what mainframe and other Online Transaction Processing (OLTP) vendors always knew. The problems associated with an OLTP application, that is an application allowing a large number of remote clients to read and modify a central data store, are very different from the problems associated with a desktop application. Nothing bore this out more than the early practice of moving "chatty" COM objects from the client to an application server without redesign. While this did result in a three-tier architecture, it also resulted in poor performance as the user interface layer made multiple calls over the network to communicate with the COM object. Worse, those who held fast to object-oriented principles when building their business objects were hit the hardest when they tried to distribute the objects to the middle tier. Confusion reigned as developers struggled to correlate object-oriented dogma to the world of distributed programming.

When Microsoft introduced Visual Basic 4, it immediately created millions of new COM programmers. But during this period, developers were learning that COM development in Visual Basic had its downfalls. Compared to Visual C++, Visual Basic COM development is far simpler. However, some of the design decisions that made Visual Basic COM development easier hindered its use in the middle tier. For example, all Visual Basic COM objects are apartment threaded. This causes serious scalability problems when storing Visual Basic objects in the ASP Session object and prohibits them from participating in object pooling. To be fair, in many business scenarios, these issues could be avoided or had minimal impact

on the application. But the limitations were enough for many to decide they would rather tackle the complexity of COM development in Visual C++.

This period also saw COM itself coming under fire. A few of the major complaints follow:

- COM component versioning and registration is complex and poorly understood, contributing to the malaise known as DLL hell, in which installations of one product break another.

- COM objects developed for ASP script consumption require special consideration because interfaces—a key part of COM programming—cannot be consumed by ASP scripts.

- In DCOM, remote COM objects are marshaled by reference by default. This means a client's method calls must travel over the network to the COM object. It is decidedly nontrivial to implement a by value marshaling scheme whereby a custom COM object is copied from one application to another, thus allowing the client to call methods on the local copy.

In spite of my sobering title for this period, I should note that some of Microsoft's most impressive work came out of this era. One example is Microsoft Transaction Server (MTS). This product bridged the gap between the realities of programming OLTP applications and the idealism of object-oriented programming by associating the lifetime of a transaction with that of an object. Microsoft refined MTS into what is now known as COM+. Microsoft also started promoting the Distributed interNet Architecture (DNA), an umbrella term for its n-tier services and technologies such as COM+, IIS, ASP, and SQL Server. More importantly, DNA included architecture advice and recommendations so that others could avoid the mistakes made by the early n-tier adopters. Finally, Microsoft delivered Windows 2000, which proved that Microsoft could build a stable, feature-rich operating system for server-side computing.

Armed with these technologies and a new awareness of the perils of distributed programming, developers delivered many high-quality applications using DNA. But in spite of these and other strong products, Microsoft's server market share plateaued and even started to shrink. Strong competition from Java's J2EE technology and open source Linux started to take its toll. When faced with such a challenge from the Web, Microsoft turned to COM for the solution. Now, they realized, COM was the problem. Advertised as the glue that connected the pieces of DNA together, COM development was too complex in C++, and too restrictive in Visual Basic. Furthermore, Microsoft found that server-side applications often

involved hardware and operating system platforms from a variety of vendors. COM, however, was designed from the start as a Windows technology, and attempts to port COM to other operating systems were dismal failures.

In the end, it was clear that after a nine-year reign as the foundation of all that was Microsoft, COM needed to be replaced. This, of course, brings us to the present, .NET, and the purpose of this book.

The Present: .NET

After nine years of touting the benefits of COM, it is understandable why Microsoft refers to .NET as an evolution rather than a revolution. And honestly, looking at it from a goal-oriented point of view, this description is accurate. Both COM and .NET share the same goals. .NET just does it better.

- **Language Independence.** You can build .NET applications in any language that supports the Common Language Specification. Currently, you can choose from about 20 different languages.

- **Component interoperability.** .NET components share a common type system, therefore, .NET achieves nearly seamless interoperability between components regardless of the implementation language. In fact, you can derive a VB .NET class from a C# class.

- **Location Transparency.** Code accessing a local (in-proc) object is identical to the code accessing a remote (out-of-proc) object. Details are handled through configuration.

- **Robust Versioning.** .NET provides and enforces a flexible and robust versioning scheme. DLL hell is solved.

In Chapter 2, you will see more detail regarding these items. For now, understand that while the goals of COM and .NET may be similar, the underlying technology is completely different. For example, `IUnknown`, `IDispatch`, and the other standard COM interfaces are not part of .NET. .NET components are not registered in the system registry. And object lifetime is determined through garbage collection instead of reference counting.

Distributed applications share the same set of problems regardless of development platform. So many of the new .NET technologies simply replace the functionality of existing COM-based technologies. Table 1-1 shows the relationship between COM and .NET technologies in the context of distributed programming.

Table 1-1. Comparing COM Technologies to .NET Technologies

Distributed Problem	**COM Solution**	**.NET Solution**
How does the application interact with the database?	ADO	ADO.NET
How does an application access the services of another application?	DCOM	DCOM, Remoting, or Web services
How does data get passed from one application to another? `IMarshal` or brute force serialization.	CLR Serialization	
What provides distributed transactions, Just-In-Time activation, and other services required by an application server?	COM+	COM+
What provides asynchronous messaging with guaranteed delivery?	MSMQ	MSMQ

Of course, these technologies are the focus of the rest of the book. And as you will see, .NET provides a productive, powerful, and extensible framework for developing distributed applications.

Summary

Designing and implementing distributed applications is hard. The available options and their inevitable tradeoffs can be overwhelming. This chapter discussed several topics that hopefully have brought some clarity to what is inherently a difficult subject:

- You have learned five key distributed design principles and how they affect the application.
- You have seen how the distributed programming has evolved and what mistakes were made in the past.
- You have learned why Microsoft created .NET to replace COM.

In the next chapter, you will learn the basics of .NET. In particular, the chapter focuses on the fundamental .NET technologies that have the largest impact on distributed programming.