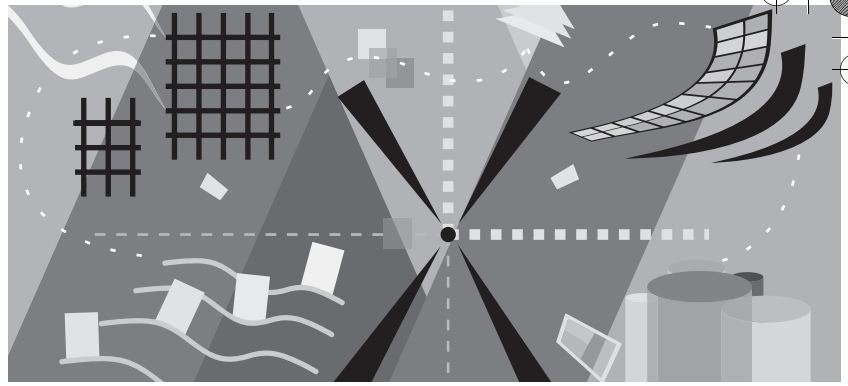


1



Introduction

*I cannot get my sleep to-night; old bones are hard to please;
I'll stand the middle watch up here—alone wi' God an' these
My engines, after ninety days o' race an' rack an' strain
Through all the seas of all Thy world, slam-bangin' home again.*

—Rudyard Kipling, writing on the importance of 24/7
availability under load, “McAndrew’s Hymn,” 1894

The Big Internet

The Internet is Big. (Annoyed Reader: “I paid you HOW MUCH to tell me that?” Royalty-Counting Author: “What, you mean, it’s NOT?”)

On its own, a desktop PC is boring, much as a single-cell amoeba is boring. Sure, you can use it (the PC, not the amoeba) to play a mean game of Solitaire and it won’t let you cheat (this is an advantage?), and Notepad comes in handy on occasion. But unlike the evolutionary value of the amoeba, the economic benefits to society of the stand-alone desktop PC have yet to be satisfactorily proven. It just can’t do that much interesting or useful stuff as long as its horizons remain limited to its own box. However, when you use the Internet to link your PC to every other PC in the world, and to every intelligent non-PC device (palmtops, refrigerators, and so on) as well, for essentially no extra hardware cost, fun things start to happen—just as when enough single cells connect and evolve to form the human brain, which can

Stand-alone PCs are
much less useful than
networked PCs.

2 Introducing Microsoft .NET

The Internet continues to change society ever more rapidly.

compose, play, and appreciate a symphony; fly to the moon; or obliterate its own species. Beats the heck out of Solitaire, doesn't it?

The Web started out as a means for browsing boring physics reports and took off (understatement of the century) from there. It dramatically lowered the friction of distributing all types of data. Expanded content attracted more users to the Internet, and the increasing numbers in the audience drew in more content providers in a virtuous cycle that not only shows no signs of ending but is accelerating even as I write these words. Yesterday I used the Web to watch video highlights of a hockey game that I missed. After that, I used Morpheus to find some good music among 500,000 different songs (I only look at the legal ones, naturally) on 10,000 different users' hard drives. Then I talked to my mother about setting up a video camera link so that she could look into her first grandchild's crib from her home 500 miles away. The Internet has made ours a completely different world from that of even 5 years ago.

Internet hardware and bandwidth are cheap and getting cheaper.

Hardware for connecting to the Internet and bandwidth for transmitting data are cheap and getting cheaper. The Web camera that lets my mother watch my daughter cost only a couple of hundred bucks to install in my PC and essentially nothing extra to operate over my existing cable modem. Think what it would have cost 10 years ago, or even 5, to buy video hardware and lease a dedicated line from Ipswich, Massachusetts, to Orwigsburg, Pennsylvania. The prices of Internet hardware and bandwidth will soon fall to Cracker Jack prize level, if they haven't already.

Raising the Bar: Common Infrastructure Problems

Internet software poses new classes of problems that are difficult and expensive to solve.

The hardware and bandwidth are cheap, but there's a snag. Platt's Second Law states that the amount of crap in the universe is conserved.¹ If someone has less crap to deal with, it's because he's managed to dump his ration on someone else's head, but there's no such thing as making it disappear. If hardware and bandwidth are so much easier and cheaper to get, that means writing the software to run that environment must, by the laws of the universe, be harder and more expensive by a corresponding amount. And so indeed it has proved, as anyone who's tried lately will confirm. The problem in your Internet application isn't the business logic, which is much the same as a desktop case (a certain number is less than zero, indicating that your checking account is overdrawn). However, the fact that you're implementing an application on different boxes connected by the Internet introduces new

1. Platt's First Law is called "Exponential Estimation Explosion." It states that every software project takes three times as long as your best estimate, even if you apply this law to it.

classes of problems, because of the Internet's public, uncontrolled, and heterogeneous nature. Think how (relatively) easy it is to handle a toddler in your own living room. Then think how much harder it is to handle that same toddler in Grand Central Station. Same kid, same goals (safety, fun), entirely different requirements.

Consider the question of security, for example. Many users keep their personal financial records on stand-alone PCs, using Quicken or similar products. Developers of early versions of Quicken didn't write any security code. They were comfortable, or more properly, their customers were comfortable, that if they kept their PCs physically locked up, no one would steal their money. Paranoid users could buy a product that would password-protect their entire PC, but hardly anyone did this.

Desktop applications generally didn't have any security at all.

But Quicken on the desktop wasn't that useful, as many users discovered once the novelty wore off. It did very little more than what your paper check register already did—often more quickly and more easily than the silly program. Quicken didn't become a net benefit to users until it could connect to the Internet and interact with other financial parties, with features such as electronic bill receipt and payment and automatic downloading of bank and credit card statements. (That is, it would become a net benefit to users if its user interface wasn't so lame. It doesn't handle the complexity of these new features well at all, overwhelming the user with far too many indistinguishable choices at any given time. But that's not an Internet problem.)

However, once Quicken's operations leave the secure cocoon of the single box on which they're running, they run smack into a massive new need for security. For example, when the user tells the electronic bill paying center to write a check to the phone company, the bill paying center needs to ensure that the request comes from the genuine owner of the account and not from the phone company desperately trying to stave off bankruptcy by advancing itself a couple of months' worth of payments. We also need to encrypt all the data that flows between the parties. You don't want your neighbor's geeky teenage son using a packet sniffer on your cable modem line to see your account numbers and what you bought—"\$295 to Hunky Escort Service, eh? Wonder if her husband knows about that?"

Internet applications need security for all phases of their operations.

This security code is extremely difficult to write—and to test, and debug, and deploy, and support, and maintain, while employees come and go. You have to hire people who know everything about security—how to authenticate users, how to decide whether a user is allowed to do this or that, how to

Security code is extremely difficult to develop.

4 Introducing Microsoft .NET

Security and similar problems in distributed computing belong to the generic category of infrastructure.

Infrastructure, not business logic, is what kills projects.

encrypt data so that authorized users can easily read it but snoopers can't, how to design tools that administrators can use to set or remove users' security permissions, and so on.

Internet computing raises other similar classes of problems, which I'll discuss later in this chapter. All of them share the characteristic that, as I first explained in my book *Understanding COM+* (Microsoft Press, 1999), they have nothing whatsoever to do with your business process, the stuff that your clients are paying you to get done. Instead, these problems represent infrastructure, like the highway system or the electric power grid, the framework on which you and other people weave your day-to-day lives.

Developing infrastructure kills projects. I have never seen a project fail over business logic. You know your business process better than anyone else in the world; that's why you're writing software to assist it. But you don't know the infrastructure (unless that's your product, as it is for Microsoft). Almost no one is an expert on security authentication algorithms or encryption. If you try to write that code yourself, one of two things will happen. You'll either write a lame implementation because you don't know what you're doing (and you had better hope no bad guys notice it), or you'll try to write an implementation and fail to complete it before the money runs out. When I worked on a pre-Internet distributed foreign exchange application twelve years ago (also described in *Understanding COM+*), we did both.

The Best Laid Plans

Software developers delude themselves.

Software developers always begin a project with the best of intentions and the loftiest of promises. Like an alcoholic heading to a party, we swear that we'll avoid bugs through careful design and that we'll document our code thoroughly. We'll test it all the way through, and we won't add features after the code freeze. Above all, we'll make realistic schedule estimates and then stick to them. ("That operating system feature doesn't do exactly what we want, but I can write a better one in a week, so there's no need to modify our program requirements. Two weeks at the outside. Unless I hit a snag. Oh, will we need testing time, too?") A robust, useful application (sobriety) is within our grasp, all we have to do is stay disciplined. No Solitaire at all. At least not until after five. OK, one game at lunch. "Dang, I lost; just one more to get

even, OK? Hey, how'd it get to be 4:00?" Every single project I've ever seen has begun with this rosy glow.

Has any developer ever kept these promises? No. Never has, never will. We know in our hearts that we're lying when we promise. It's a disease. Like addicts, there's only one way out (not counting dying, which everyone does at exactly the same rate: 1.000 per person.) To write successful Internet applications, we of the software development community need to do what recovering addicts do and embrace at least two steps on our path to righteousness:

1. Admit that we are powerless—that our lives have become unmanageable.
2. Come to believe that a power greater than ourselves can restore us to sanity.

Application programmers must admit that we are powerless over Internet infrastructure. It makes our projects unmanageable. We cannot build it; it takes too long, it costs too much, and we don't know how. We must not even try, for that way lies sure and certain doom. In the extremely unlikely event that we do accidentally write some decent infrastructure code, our competitors who don't will have long since eaten our lunches, and breakfasts and dinners besides. You don't build your own highway to drive your car on, nor do you install your own power generating equipment (unless you lived in California in early 2001).

You can't afford to build infrastructure yourself.

Fortunately, your Internet application's infrastructure requirements are the same as everyone else's, just as your requirements for highways and electricity are similar to those of many other people. Because of this large common need, governments build highways and power companies build power plants, which you use in return for a fee, either directly through tolls and bills, or indirectly through taxes. Governments and utilities reap large economies of scale because they can hire or develop the best talent for accomplishing these goals and because they can spread the development cost over many more units than you ever could.

What we really need is for someone to do for distributed computing what the government does for highways (maybe not exactly what the government does for highways, but you get the basic idea). As recovering addicts believe that only the power who created the universe in which we compute can restore their lives, so developers need a higher power in computing to provide our Internet infrastructure, restoring our development efforts to sanity.

You want someone else to build it and for you just to use it.

Microsoft .NET provides prefabricated infrastructure for solving the common problems of writing Internet software.

What the Heck Is .NET, Anyway?

That's what Microsoft .NET is—prefabricated infrastructure for solving common problems in Internet applications. Microsoft .NET has been getting an enormous amount of publicity lately, even for this industry. That's why 5000 rabid geeks, crazed on Jolt Cola, converged on Orlando, Florida, in July 2000. Not because they can't pass up a bargain off-season airfare, even if it's not somewhere they want to go, or because they enjoy punishing heat and sunstroke. It was to hear about Microsoft .NET for the first time.

The server-side features of Microsoft .NET run on Windows NT, Windows 2000, and Windows XP Professional. The client-side features run on these plus Windows 98, Windows Me, and Windows XP Home. While it's currently an add-on service pack, later versions of .NET will probably be made part of the operating system. Later versions may or may not be announced to allow at least portions of it to run on other versions of Windows or, as we shall see, perhaps for other operating platforms as well. Microsoft .NET provides the following services, all discussed later in this book.

- **A new run-time environment, the .NET Framework.** The .NET Framework is a run-time environment that makes it much easier for programmers to write good, robust code quickly, and to manage, deploy, and revise the code. The programs and components that you write execute inside this environment. It provides programmers with cool run-time features such as automatic memory management (garbage collection) and easier access to all system services. It adds many utility features such as easy Internet and database access. It also provides a new mechanism for code reuse—easier to use and at the same time more powerful and flexible than COM. The .NET Framework is easier to deploy because it doesn't require registry settings. It also provides standardized, system-level support for versioning. All of these features are available to programmers in any .NET-compliant language. I discuss the .NET Framework in Chapter 2.
- **A new programming model for constructing HTML pages, named ASP.NET.** Even though intelligent single-use programs are on the rise, most Internet traffic for the near- to middle-term future will use a generic browser as a front end. This requires a server to construct a page using the HTML language that browsers understand and can display to a user. ASP.NET (the next version of Active Server Pages) is a new environment that runs on Internet

Information Services (IIS) and makes it much easier for programmers to write code that constructs HTML-based Web pages for browser viewing. ASP.NET features a new language-independent way of writing code and tying it to Web page requests. It features .NET Web Forms, which is an event-driven programming model of interacting with controls that makes programming a Web page feel very much like programming a Visual Basic form. ASP.NET contains good session state management and security features. It is more robust and contains many performance enhancements over original ASP. I discuss ASP.NET in Chapter 3.

- **A new way for Internet servers to expose functions to any client, named XML Web services.** While generic browsers will remain important, I think that the future really belongs to dedicated applications and appliances. The Web will become more of a place where, instead of data being rendered in a generic browser, a dedicated client (say, Napster, for music searching) will make cross-Internet function calls to a server and receive data to be displayed in a dedicated user interface or perhaps without a user interface at all for machine-to-machine communications. Microsoft .NET provides a new set of services that allows a server to expose its functions to any client on any machine running any operating system. The client makes calls to the server using the Internet's lowest common denominator of XML and HTTP. A set of functions exposed in this manner is called an XML Web service. Instead of sitting around waiting for customers to see the light and embrace the One True Operating System (Hallelujah!), the new design seems to say, "Buy our operating system because we provide lots of prefabricated support that makes it much easier to write applications that talk to anyone else in the entire world, no matter what or where they're running." I discuss XML Web services in Chapter 4.
- **Windows Forms, a new way of writing rich client applications using the .NET Framework.** A dedicated client application that uses XML Web services needs to provide a good user interface. A high-quality interface can provide a much better user experience, as the dedicated interface of Microsoft Outlook is better than the generic Web user interface of Hotmail. Microsoft .NET provides a new package, called .NET Windows Forms, that makes it easy to write dedicated Windows client applications using the

8 Introducing Microsoft .NET

.NET Framework. Think of Visual Basic on steroids, available in any language, and you'll have imagined the right model. I describe .NET Windows Forms in Chapter 5.

- **ADO.NET, which provides good support for database access within the .NET Framework.** No Internet programming environment would be complete without some mention of database access. Most Internet programs, at least today, spend most of their time gathering information from a client, making a database query, and presenting the results to the client. .NET provides good support for database operations using ADO.NET. I cover ADO.NET in Chapter 6.
- **Outstanding support for handling XML documents and streams.** Operating in the modern distributed computing environments requires applications to handle XML. The .NET Framework contains outstanding support for writing applications that handle XML documents and streams. I discuss XML in Chapter 7.
- **A standardized mechanism for signaling asynchronous events.** Providing a standardized mechanism for callbacks from a server to its client was a large stumbling block in pre-.NET COM-based programming. The .NET Framework provides a standardized mechanism for one party to make an asynchronous call to another. I discuss this eventing mechanism in Chapter 8.
- **Support for writing multithreaded code.** The Windows operating system acquired preemptive multithreading in 1993 with the release of 32-bit Windows NT. Unfortunately, multithreaded programs have been difficult to write due to the low level of support from the operating system. The .NET Framework contains much more support for allowing everyday programmers to make use of the operating system's multithreading capabilities. I discuss threading in Chapter 9.
- **Support for writing your own Windows Forms and Web Forms controls.** The concept of a control, a prepackaged unit of functionality dealing with a user interface, has been fantastically successful. Both Windows Forms (Chapter 5) and ASP.NET Web Forms (Chapter 3) get most of their functionality from their ability to host controls. .NET also provides excellent support for users to develop their own controls, either for internal use or for sale to

third parties. Chapter 10 and Chapter 11, respectively, discuss writing Windows Forms controls and Web Forms controls.

About This Book

Until I wrote *Understanding COM+*, all of my books had been low-level how-to manuals and tutorials, with the samples written in C++. This worked beautifully for hard-core programmers who write in C++, but unfortunately this is a small percentage of the people who buy computer books, which made my creditors very unhappy. I wanted to make this book accessible to developers who didn't know or didn't like C++. Furthermore, I found that managers got essentially nothing out of my C++-based approach because they never worked with the sample programs (with only one exception I know of, and I'm sending him a free copy of this book for working so hard to understand my last one). I really wanted to reach that audience, even more than programmers. An ignorant (or worse, half-educated) manager is an extremely dangerous beast. Eliminating that species would be my grand contribution to civilization.

This book uses the basic style I experimented with in my last book, in which I adapted the format that David Chappell used so successfully in his book *Understanding ActiveX and OLE* (Microsoft Press, 1996): lots of explanations, lots of diagrams, and very little code in the text descriptions. As much as I liked David Chappell's book, I still felt hungry for code (as I often need a piece of chocolate cake to top off a meal of delicate sushi). I found myself writing code to help me understand his ideas, much as I wrote equations to understand the textual descriptions in Stephen Hawking's *A Brief History of Time*. (OK, I'm a geek.) So my book comes with sample programs for all the chapters, some of which I wrote myself and some of which I adapted from Microsoft's samples. These sample programs are available on this book's Web site, which is <http://www.introducingmicrosoft.net>, naturally. Managers and architects will be able to read the book without drowning in code, while code-hungry programmers will still be able to slake their appetites. Most of the sample code I present in the text of this book is written in Visual Basic .NET because that's the language that most of my readers are familiar with. However, I got so many requests for C# code after the first edition of this book that I've also provided C# versions of all the samples on the book's Web site. If you want to run the sample programs, you'll need a computer running

Sample programs and installation instructions are available on this book's Web site.

10 Introducing Microsoft .NET

Each chapter of this book presents a single topic from the top down.

Windows 2000 Server or Windows XP Professional, the Microsoft .NET Framework SDK, and Visual Studio .NET. Detailed system and installation requirements for the sample programs are available on the book's Web site.

Each chapter presents a single topic from the top down. I start by describing the architectural problem that needs to be solved. I then explain the high-level architecture of the infrastructure that .NET provides to help you solve that problem with a minimum amount of code. I next walk you through the simplest example I can imagine that employs the solution. Managers may want to stop reading after this section. I then continue with a discussion of finer points—other possibilities, boundary cases, and the like. Throughout, I've tried to follow Pournelle's Law, coined by Jerry Pournelle in his "Chaos Manor" computing column in the original *Byte* magazine, which states simply, "You can never have too many examples."

Sing a Song of Silicon

I can't stand modern poetry.

Modern poetry bores me silly. I find most of it indistinguishable from pompous politicized prose strewn with random carriage returns. It has no rhyme, no rhythm, just an author (usually with a private income or taxpayer's grant, else he'd starve to death) who suffers from the fatal delusion that he has Something Important To Say. Maybe my feeble mind just doesn't want to make the effort of parsing his intentional dislocations. Don't know about you, but I've got other things to do with my few remaining brain cycles.

Rudyard Kipling's poetry is great.

On the other hand, I love older poetry, especially Rudyard Kipling. He's not politically correct these days—read his poem "The White Man's Burden" if you want to know why. In his defense, I'll say that he was a product of his times, as we all are. And he won the Nobel Prize for Literature in 1907, so someone must have liked him then. My grandparents gave me a copy of his *Just So Stories*. My parents used that book to read me to sleep, and it was one of the first books I learned to read myself. I graduated to Kipling's poetry in high school English class, where I found reading his section of the literature book far more interesting than listening to the teacher. His poems still sing to me as no one else's ever have, before or since.

Kipling wrote a poem celebrating a marine engineer named McAndrew, almost all of which applies to programmers today.

What does this have to do with computer geekery, you ask? The incredible acceleration of technological innovation in the last few years brings to my mind Kipling's poem "McAndrew's Hymn," published in 1894. Most of us think of modern times as different from a hundred years ago and nowhere

more so than in technology. Still, I'm astounded at how much of McAndrew's feelings resonate with me today. The title character is an old oceangoing Scottish engineer musing on the most brilliant technological accomplishment of his day: the marine steam engine. That was the beginning of the death of distance, a process that you and I, my fellow geeks, will complete ere we rest. I like this poem so much that I've started every chapter with an excerpt from it. You can read the whole thing on line at <http://home.pacifier.com/~rboggs/KIPLING.HTML>. You may think of Scotty on Star Trek as the prototypical Scottish engineer, but I'm convinced that Gene Roddenberry based him on Kipling's McAndrew.

Every programmer, for example, knows Moore's law, right? It says that computing power at a given price point doubles every eighteen months. Many programmers also know its reciprocal, Grosch's law, which states that it doesn't matter how good the hardware boys are because the software boys will piss it away. A few even know Jablokow's corollary, which states simply, "And then some." But McAndrew figured this out a hundred years ago, way before some plagiarist stuck Moore's name on the idea and called it a law. I think of Kipling's words as I contemplate the original 4.77 MHz IBM PC (with two floppy drives and 256 KB of memory) that I use as a planter:

The poem includes an early formulation of Moore's Law.

*[I] started as a boiler-whelp when steam and [I] were low.
I mind the time we used to serve a broken pipe wi' tow.
Ten pound was all the pressure then - Eh! Eh! - a man wad drive;
An' here, our workin' gauges give one hunder' fifty-five!
We're creepin' on wi' each new rig - less weight an' larger power:
There'll be the loco-boiler next an' thirty mile an hour!*

Like Rodney Dangerfield, we geeks yearn for respect and appreciation. Society has looked askance at us ever since the first cave-geek examined a sharp stone and said, "Cool fractal patterns. I wonder if it would scale to spearhead size?" Remember how girls in high school flocked around football players, most of whom (not all, Brian) were dumb as rocks? A straight-A average was uncool (mine would have been if I'd had one), and even my chess championship trophy couldn't compete with a varsity letter. Even though I knew that in the long run I'd make far more money than the high-school jocks (which my father pointed out is far more attractive to the opposite sex),

12 Introducing Microsoft .NET

it still burned. McAndrew cried aloud for the same thing, only far more eloquently (my emphasis added):

*Romance! Those first-class passengers they like it very well,
Printed an' bound in little books; but why don't poets tell?
I'm sick of all their quirks an' turns-the loves an' doves they dream-
Lord, send a man like Robbie Burns to sing the Song o' Steam!*

"Lord, send a man like
Robbie Burns to sing the
Song o' Steam!"

No Robbie Burns am I, and not even my mother likes hearing me sing. But I've done my best to tell the story as I see it today. I hope you enjoy reading it.