Chapter 2 Internet Application Technologies

The first thing that you'll notice when starting to build Web applications is new terminology. Building Internet applications is more involved than building stand-alone applications because you have to deal with multiple environments simultaneously. With this additional complexity comes terminology. The bad news is that you have to become familiar with lots of three- and four-letter acronyms; the good news is that most of these technologies are easy to work with once you have a broad view of how they fit together. This chapter provides this broad overview. It talks about the technologies and information flow for distributed and Web applications, and discusses a number of the terms you need to know.

When you look back a few years, it's quite amazing to think how quickly Internet technology has taken over the network and network application environments. Four years ago, relatively few PC-based networks were running the TCP/IP protocol, which is the base network protocol used by all high-level Internet protocols. Today, other network protocols have been pushed to the sidelines or now run on top of TCP/IP. You probably use Internet protocols like Hyper Text Transfer Protocol (HTTP) , File Transfer Protocol (FTP) , and Simple Mail Transfer Protocol (SMTP) throughout your workday without giving them much thought.

Open standards open doors to the world

One major reason why TCP/IP has taken over is that it provides an open and standards-based protocol governed by a standards body rather than by a single company or vendor. The protocol can be freely implemented by any operating system or network solution provider, which makes it accessible from just about any machine that needs to communicate over a network. This common network protocol is a crucial requirement to allow for ubiquitous access from anywhere, and a key reason that the Internet has grown in popularity so quickly.

TCP/IP was around for a long time before the Internet started taking off. But the combination of an open network architecture, along with higher-level and equally open protocols and application implementations of these protocols, brought TCP/IP into the PC mainstream. In particular, the HTTP protocol used for accessing the World Wide Web and the first implementation of a graphic Web browser triggered the focus toward the TCP/IP-centric world of the Internet and its supporting technologies.

The success of TCP/IP is based on the openness of the standards as well as the relative simplicity of the protocols implemented on top of it. Most of the popular protocols like HTTP, FTP and SMTP are text-based and are fairly easy to implement even at the application level—simple protocols make for wide adaptation because more products can support the standards. Simple protocols also make for flexibility—if it's easy to access information at the protocol level, programmers have control over changing the way data is sent or communicated over the given protocol. You'll see some examples of this once we start talking about building Web applications and modifying the Request headers when returning results to the browser, in order to tell the browser to display the information in a certain way.

Of clients and servers

Most Internet technology works based on client and server relationships. A *client* is a consumer of services or content provided by a *server*. Typical Internet server applications have multiple simultaneous clients that request data from servers. The servers respond by sending information back to the client, which uses this information. The most familiar scenario is a Web browser requesting data from the Web server and then displaying the returned information. The server functionality is hidden from the browser, which doesn't know or care how the Web server provides the information. The Web server might provide the information as a simple HTML file loaded from disk, or it might ask a dynamic back-end application to create a data-driven table for display. The browser requests, while the Web server responds and supplies the content.

This simple concept gets skewed a bit by the fact that machines or even applications can be both servers and clients at the same time. For example, a Web server responds to a dynamic request that requires it to call another application to perform the actual processing. A common example is the Web server making a request to SQL Server to retrieve some data. In this case the Web server is a *server* to the Web browser but also a *client* to SQL Server, which is the server providing the database data results for the Web server.

As you can see, the client/server concept is often applied to more than physical machine boundaries, and deals with logical aspects of an application and the relationships between the interacting components. Let's take a closer look at the server and client definitions to put these concepts into perspective as they apply to the development process.

Servers

For database applications, most of the application logic continues to reside on the server. Pure server-side applications running HTML-based interfaces are by far the most common Internet applications running today. When you visit a Web site that displays dynamic data brought back by criteria you supply, or when you fill out a form that captures your contact or registration information, you're running a server-side application that generates all the application and HTML on the server. The client side acts only as a terminal to display and navigate the data. Focus is on the server because the database almost always resides on the server. In most cases, clients don't have direct access to the database, both for performance and security reasons.

There are different approaches to servers, from direct client access to a server over a TCP/IP socket connection, to the more common Web-based interfaces that use Web-based processing engines to access back-end applications that can create dynamic output. Servers are workhorses—they are asked to perform tasks and return results as follows:

- A browser asks a Web server to provide HTML.
- An application calls a COM object to retrieve a result.
- A Java applet talks to a server-side Java application to download some data.
- An e-mail client makes a request to download all messages from a mail server or asks it to send messages.

In all of these examples, the server is asked to perform a task and often returns a result to a client.

This book discusses high-level server tools, which continue to be the most popular and most scalable solutions both in terms of load and accessibility. In particular we'll look at server-side Web back-end solutions that allow you to create HTML/HTTP applications with Microsoft Active Server Pages (ASP) and FoxISAPI. With these tools, code runs on the server to build HTML that gets sent back to the browser for display. The tools allow flexibility in output creation and can be extended by using custom Visual FoxPro code to build business logic and handle database access. Here, everything happens on the server: business logic, database access, and generation of the display logic as HTML. The browser merely acts as reader for the information sent back by the Web server (that's not entirely true—you can write scripting code and use Java and ActiveX, but for the sake of discussion here we're dealing with server technology). I call these *high-level tools* because they work through the Web server rather than allowing the client to directly access the application using a standard protocol: HTTP. The Web server takes the request and routes it to the appropriate back-end application, which provides the actual application logic.

Contrasting with high-level Web server applications are low-level servers, which are directly accessed and use straight network connections. For example, a Java Applet or ActiveX control running on a browser might access a Java application over a direct TCP/IP socket connection. In this scenario, the client directly accesses the back-end application by programming a network connection directly, rather than talking to an intermediary like the Web server. Although this can provide better performance, it also involves a lot more work because you have to come up with your own protocols instead of using existing, proven ones such as HTTP and the formatting provided by the Web server. Client and server share responsibility in this scenario, but even well-designed applications will likely leave most of the hard-core business logic on the server, with the client side code providing the front-end user interface services.

Some technologies, such as Microsoft Remote Data Service (RDS) use both client-side logic and server-side components. With this technology the code is written on the client, but the actual data operations are *marshaled* over the Web and actually performed on the server. Marshalling means taking requests on one end of a connection and then passing the request to the other end, which does the actual work. The server returns the result and the client can then use the result data. The point here is that the technology spreads some logic to both ends of the connection. Most tools in this category implement the low-level server and then wrap the logic into a simpler interface (RDS uses an ActiveX control) that can be directly used on the client side in the Web browser or a stand-alone application. Microsoft is implementing several technologies along these lines.

Finally, there's a drive to make it possible to instantiate objects over the Internet just as you can with local COM objects on your computer. Of particular interest is Distributed COM (DCOM) over HTTP, which makes it possible to efficiently access COM objects over a regular HTTP connection. Java also implements this sort of connectivity using a standard protocol called Remote Method Invocation (RMI) over CORBA (a non-Microsoft object model implementation similar to COM).

Server contexts

When discussing servers, it's crucial to understand in which context a client/server relationship exists. As pointed out above, servers can exist in two entirely different contexts:

- **Physical (Machine)**
  For an application user who accesses a Web server via a browser, the local browser is the client and the remote Web server is the server. There's a clear separation between the client and the server by way of the separate machines that handle the client and server chores.

- **Logical (Application)**
  Client and server applications can exist on the same machine. One application can make requests to another to retrieve information.

This book focuses extensively on logical client/server relationships between applications. The transfer mechanism most prominently addressed is Microsoft's Component Object Model (COM). Note that an application server can be physically deployed on either a local or remote machine. A good example is a COM object that you can access via DCOM.. The COM object is an application server, but it can also be called from a remote client that accesses the server over a network.

Physical connections between machines are essentially hidden by network protocols. You can access an application either locally or across the wire with very few changes in the way the tools are used. All that changes is the network address. This applies both to physical network access (such as browsing a Web page) as well as the lower level, direct access using COM or custom created clients and servers. With all of these, the network abstracts the fact that the application might be talking to a local application or one that's running across the globe in Singapore.

Server states

Server programming tends to be a little different from standard application programming. Stand-alone applications deal extensively with responding to events that occur when the user performs certain operations. In server programming, this event-driven model is replaced by a transaction-based model in which requests are treated as self-contained requests to the server that are completed and returned to the client.

Think of a server as a typical non-visual class in Visual FoxPro. The class has methods that you call and that have return values of some sort of result data or content (I use the term *content* here for returning document data that gets embedded in the output stream). Servers are very similar to this scenario, and in fact this is how you built COM servers in Visual FoxPro. But because of the way the distributed architecture works, servers can be either stateless or stateful:

- **Stateless**
  A *stateless* server does not retain any context for the client. The server does not know what action the previous request performed and does not make any assumptions about the current environment. The client typically must provide some mechanism for the server to establish state as part of the request on the server. For example, a client requests a record

object from a server and passes in a primary key (PK) to let the server know which record to retrieve. Based on the PK, the server can establish the state required to retrieve the record. In a stateless server, the PK would be required even for an operation like MoveToNextRecord() because the server doesn't explicitly know what record the previously retrieved record was. So the required parameter to MoveToNextRecord() is the primary key of the current record, and the server then uses that record as its starting point. The concept behind a stateless server is to connect to the server, let it establish context, get your data and then disconnect—losing state at that time. For this reason, stateless servers are often referred to as *connectionless servers*.

- **Stateful**
  Servers that can retain state between requests are appropriately called *stateful*. Each client that uses a stateful server gets a persistent connection to the server, and the client and server both can make assumptions about the values set on the server. If you set a property on a stateful server, the property will still be set in the next request. An example of a stateful server is a Visual FoxPro application creating an instance of Word to create a form letter. Each call to the Word COM object causes the document to be filled or manipulated. Because the document sticks around for the lifetime of the object and the user holds a unique copy of the object, the server is considered stateful. The continued existence of the changing document is the server's state.

Most Internet solutions call for stateless servers because they provide much better scalability than stateful servers for large numbers of connected users. The reason for this is simple: Keeping state requires each user to have his own copy of an object with its own private address space. If you have 5000 simultaneous users of your Web site, a stateful server would use 5000 instances of an object or, at the very least, 5000 copies of the unique data for that server. As you can imagine, this places a huge load on the server's resources. It's also inefficient because only 10 or 20 users might actually be using the server at any given time.

With stateless servers, resources are recycled. Rather than having 5000 objects you might have only 10 or 20 that are cycled to service the immediate incoming requests. This is known as *resource management* or, in more technical terms, as a *pool manager*. Pool managers are implemented at all levels:

- Web servers use HTTP connection pooling to provide services to large numbers of users without giving each user a dedicated connection.
- ODBC uses connection pooling to handle multiple connections to the same data source without having to reconnect each time.
- *Microsoft Transaction Server* (MTS) implements a resource management feature called *Just In Time Activation* that pools server resources by unloading and reloading servers as needed. (A future version of MTS will provide a real pool manager for pooling references to stateless objects.)
- FoxISAPI includes a built-in pool manager to manage multiple, simultaneous Visual FoxPro COM objects for back-end operation. With a pool manager, a few objects can service thousands of simultaneous clients.

Clients

Although distributed architecture places a heavy emphasis on server-side processing, the client side also has a lot to contribute. Client-side code can work in combination with server-side code, which is happening more and more now with the richer Dynamic HTML object models becoming available in the HTML 4 and 5 and XML standards. Still, client-side code suffers from the lack of direct access to the database and typically just sits on the Web server. Without data access, the client side is fairly limited in what it can do beyond providing a "smart" user interface. It is possible to access data directly over the Web, but only at the cost of building applications that move more code (both application and system) to the client. In many cases, they also become platform specific, as is the case with most of Microsoft's client-side technologies like Dynamic HTML and RDS that only work in Internet Explorer 4.0 or later on Windows platforms.

Client-side technology makes a lot of sense if you are not focused entirely on building solutions that have to run on every conceivable platform. In fact, by using the Internet protocols discussed earlier in this chapter, you can build extremely powerful applications that take advantage of the distributed platform without getting shoehorned into a limited, HTML-only implementation. By implementing more code on the client side, you make the client more flexible. But at the same time the solution becomes less cross-platform-capable and requires more administration because the proper software needs to be installed on the client's machine.

At the highest level there's the Web browser as the client to the Web server. The browser has become the "Universal Client" interface. For better or for worse, the focus is now on getting everything to run inside a browser. Today you can view your daily Web HTML, graphics and multimedia in the browser as well as word processing, spreadsheets, graphics and presentation documents. Even full-blown applications built with Visual FoxPro using the new 6.0 Active Documents can run inside the browser as a type of specialty document. Keep in mind that that these Active Documents are not real Web applications, but simply local applications hosted in the browser running against local data in most cases. The applications make no provision for communicating with the server—it is simply a specialized viewer. In the case of VFP, it's a viewer for a local Visual FoxPro application.

Yet HTML and graphics continue to be the most common use for the browser, with many an interface forced into this rather limiting display environment. What makes the browser so powerful is its ability to mix content of different types in one place and to allow hosting of additional functionality in form of Java applets, ActiveX controls, plug-ins and embedded documents. The browser is simply a host container, and if a new technology comes along, viewers can be snapped onto the existing interface to make that content available in the browser as well.

The next step toward dynamic content is direct browser integration of scripting languages such as JavaScript and VBScript, and object model support for *Cascading Style Sheets* (CSS) that expose each object in the HTML document and make it scriptable. These scripting languages make it possible to build simple logic that deals with the user interface displayed in the browser. Scripting languages by themselves are useful but quite limited because they don't support direct access to the server and data. For this reason the browser model is extended through Java applets and ActiveX components that can be controlled by the scripting language. Applets and controls can either be applications in their own right or interact with the browser environment by letting themselves be controlled by scripts. The combination of scripting plus applets and components is a powerful one that has not been fully realized to date. However, significant moves are being made in that direction, especially in relation to doing direct database connectivity over the Web. The middle layer of components is responsible for abstracting the low-level interface with the server, making it possible to use HTTP as a data transfer mechanism. Applets and components are useful in more than just browsers. For example, ActiveX controls run most Windows-based applications. Furthermore, applets can be hosted inside a browser hosted in an application, which provides the full browser functionality directly to client applications including the ability to control the browser and the HTML document object inside of it through code. It's possible to build VFP code that entirely controls actions that occur inside of the browser (see Chapter 6). As you can see, these technologies have reach beyond the limited pure browser interface.

Finally, clients can run plain Windows applications built with high-level tools like Visual FoxPro. For example, it's possible for Visual FoxPro to act as a browser by using HTTP directly from within a VFP application. Rather than being limited to non-interactive HTML interfaces, you could have forms that fire validation events against locally cached data and send updates to the server. You can use HTTP protocols directly as you'll see in Chapter 7. Or you can use higher level technologies like Microsoft Remote Data Service, which allows you to talk to server-side data server objects from a VFP client app. With the RDS control, you essentially are allowed to run SQL statements over the Web. This type of setup allows the greatest flexibility because you get the power of a full-fledged GUI application as well as the ability to take advantage of the distributed network. But with this approach you also get away from some of the administration benefits because you must create GUI applications that need to be updated on every desktop that runs them. Compared to the browser-only requirement of pure HTML applications, this makes for much greater maintenance overhead. These solutions also often are platform specific because they take advantage of the operating system on which they run. Java promises a way to build client applications that can run on any platform, but the reality of creating powerful, cross-platform Java applications is one of very difficult and limiting implementations, which shows in the lack of many cross-platform Java applications to date. So much hype, yet so few actual applications run Java today. However, don't count on that being true in the future as Java tools get better and the technology matures and speeds up.

On a lower level, clients can talk to custom servers across the Internet. As with stand-alone applications, it's possible to build ActiveX controls and Java applets that talk to custom servers on the server side to retrieve information. In this scenario, your own custom ActiveX or Java code acts as the middleware that uses lower-level Internet protocols to abstract complex operations such as data transfers, resulting in a simple familiar interface using object syntax and SQL commands. This approach is very powerful, but also more complex because the client/server communication and protocols need to be handled by your own code.

Clients on a diet

There's been a lot of discussion in the press about *thin clients* and *fat clients*. These terms relate to the amount of code that runs on the client side of the client/server connection and the amount of hardware resources needed to run the application. Here is a breakdown of the terminology:

- **Thin Client**
  Thin-client technology suggests that all or most of the application logic resides on the

server and that the client runs little or no custom-written software. Hardware requirements should be minimal and software requirements should use standard software such as a Web browser or terminal program. A Web browser running a pure HTML application driven by server-side code, possibly assisted by some browser scripting code, is considered a thin-client solution.

- **Fat Client**
  Fat-client solutions are typically used to describe stand-alone, traditional, file-based or two-tier client/server systems where a rich GUI application talks to a SQL Server or other server back end. In most of these applications, the logic resides mostly on the client, which uses the server for special purposes but not as the primary application server. Fat-client solutions tend to require lots of resources on the client machine—typically a full installation of an application written in languages such as Visual FoxPro. The "fat" term comes from these resource requirements and the amount of code residing on the client.

- **Medium Client**
  Recently there's been a distinct shift toward medium-client solutions that focus on taking advantage of the horsepower and operating system that the client provides, while still leaving the heavy processing and logic on the server. Medium-client applications might focus on using system tools like ActiveX controls and DHTML to enhance the user's experience as well as providing much of the interface logic on the client. These apps typically have a three-tier architecture where the client side is responsible for controlling the server side as well as handling much of the user interface. What really separates medium client from fat client is that medium client focuses on standard technologies and leaves all heavy processing tasks to the server. In particular, most of Microsoft's new technologies like RDS fall into this category because they require significant resources and the ability to script the components on the client.

The lines between these three modi operandi are not always clear, but each mode has its advantages. Fat-client technology is definitely on its way out because there's a big drive under way to minimize the administrative headaches involved with maintaining and administering installed applications. Thin-client solutions have been around for a long time in the form of terminal apps; with the Web, many people have been introduced to a new kind of terminal app hosted in a slightly smarter and much prettier terminal application called a Web browser. For Web apps, thin client is by far the most popular implementation because it allows access to anyone who has a browser, from a little notepad computer to a 20-processor HP Unix server in the glass box in the IT department. Medium client walks the middle ground and is becoming much more popular for intranet applications where the hardware and software can be standardized and forced to conform to the application requirements. Still, fat-client applications will be around for a long time to come. I don't think stand-alone, monolithic applications are going away any time soon—not as long as users ask for new features, more speed and convenience in their computer applications.

HTTP makes clients and servers go 'round

Between the client and the server sits the Internet with its TCP/IP architecture and the increasingly universal HTTP. Because of the popularity of the Web, HTTP has become the most prominent of the Internet protocols. HTTP has traditionally been a *connectionless protocol*, which means that each time you make a request over HTTP the server establishes a connection, performs the request, and then releases the connection. This architecture makes it possible to handle large numbers of simultaneous clients because users are connected only while they process a request. HTTP is also optimized for this type of on-again, off-again connection by caching connection information and DNS information to speed transfers over the Web. Recent changes in HTTP 1.1 support automatic keep-alive connections, where the server tries to maintain connections as long as resources allow in order to optimize performance by not requiring reconnects for every request. These improvements provide better performance under most circumstances, but can be a burden for servers carrying large volumes of user connection loads.

When you think of HTTP, don't think that it's only for serving HTML content. You can provide any kind of content over HTTP. Not only does it handle ordinary Web traffic like documents and multimedia content, but it can also serve raw data such as query results. It is also emerging as a transport mechanism to allow communication of components over the wire. Several key Microsoft technologies are already working over HTTP: DCOM can now be marshaled over HTTP, and RDS uses packaged data streams over HTTP to handle the communication between the client and server for a SQL-based data object.

The reason HTTP is popular for these types of solutions is that it's optimized for connectionless communications, unlike straight TCP/IP connections, which can take a long time to establish over the Internet especially if not listed in Domain Naming Service (DNS) tables. For example, connecting over TCP/IP to a standard back-end application like SQL Server can take an excruciating amount of time (several minutes), where the same mechanism can be relatively quick over HTTP. HTTP connections are much quicker to connect and automatically take advantage of caching mechanisms built into the Domain Naming Service. In addition, HTTP implementations provide the wrapping and security layer necessary to prevent unauthorized access to data or applications directly over the Internet. It might be okay to access a SQL server directly from the company LAN, but over the Internet the sheer volume and possible security issues call for tighter control than direct access allows through an intermediate application or system layer.

Many companies have also discovered that HTTP is a great mechanism for building custom solutions with intelligence both on the client and server without having to build complex, low-level servers and clients. HTTP is a text-based protocol and thus it's rather easy to work with both on the client and server—I'll show some examples of this in Chapter 7. HTTP-based applications can take advantage of the infrastructure that might already be in place from an HTML-based Web application by simulating a browser. Fat or medium clients can access the same Web server used for HTML apps to retrieve data directly or communicate with the server using the same implementation details as HTML-based applications, but without generating HTML and instead returning data (text or binary) or commands. You can transfer data or simply tell the server to retrieve information and return it as data to be used in the application. This can take the simple form of a command-based approach where the client tells the server to perform tasks, or to the other extreme of full-fledged, two-way communication that involves data transfers along the way.

From Web browser to Web server

Now let's take a closer look at how the active architecture binds together the client and the server in a typical Web application. **Figure 2.1** shows the relationship between the client and server sides. Note how they are separated by a dotted line representing the physical machine boundary that is crossed only by the HTTP protocol. HTTP is the primary mechanism for transporting requests and data across this boundary.

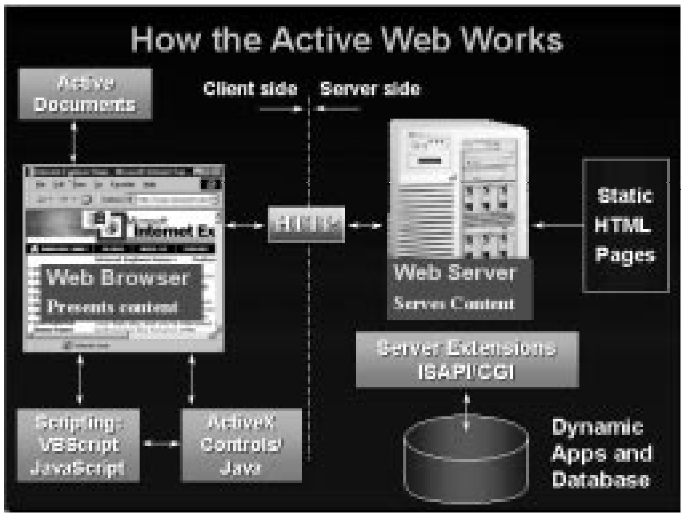

Figure 2.1*. The relationship between the client and server sides of a Web application.*

Let's take a look at how a request travels from client to server, and refer to Figure 2.1 for additional flow:

- **The browser provides the active interface.**
  The browser is essentially a terminal program on steroids, responsible for displaying content that the server provides.

- **The browser uses scripting and components to extend functionality.**
  Browsers support scripting for building logic to handle the user interface. Components can extend the browser's functionality and can consist of ActiveX controls, plug-ins and Java applets that are parameterized and embedded within an HTML document. Components can be controlled by using scripting languages.

- **Active Documents embed applications.**
  Active Documents are a special type of document available only for Internet Explorer that make it possible to run standard GUI applications inside the Web browser. It's possible to view Office apps directly in the browser. Visual FoxPro 6.0 also supports creation of

Active Document applications that allow a VFP app to run inside the browser virtually unchanged. Keep in mind that this technology is merely a packaging solution: The app runs on the local machine as a fat client without any support for data connectivity over the Web. It's no different than a local VFP application except it's *hosted* in the browser.

- **HTTP is the client-to-server transfer mechanism.**
  The HTTP protocol is used to send requests from client to server and result content back to the client. This diagram shows the most common implementation, where HTTP is the only avenue of getting to the back end. Rather than talking directly to the data, the browser always talks to the intermediary of the Web server over HTTP.

- **The Web server provides connectivity to content, data and application services.**
  Web servers are responsible for handling the HTTP requests from browser clients. Based on the URL information provided by the client, the Web server decides what type of content to send back to the browser. Requests made to the server can ask for static content, which is simply pulled from disk and sent back to the client. Alternatively, the server can be asked for dynamic data, in which case the server passes on the request information to server extensions.

- **The Internet Server API (ISAPI) and Common Gateway Interface (CGI) are used for server-side extensions.**
  The Web server by itself is not terribly smart when it comes to providing dynamic data. Dynamic requests are instead passed on to server extensions that do the dirty work of actually creating the dynamic output that goes back to the browser. The server extensions, or "scripts" as they are often called, can then decide how to create the dynamic output. ISAPI and CGI are standard extension protocols that establish how the Web server makes requested information available to the applications that want to build dynamic data. The protocols consist of formatting for incoming request data and formatting for creating the output that is sent back to the Web server. Note that just about all server-side tools use either ISAPI or CGI, including scripting engines like ASP and Cold Fusion, which hide the ISAPI/CGI scripts by using script mappings to give the impression of "scripted pages." Behind those scripts sits an ISAPI/CGI engine that actually communicates with the Web server.

- **Visual FoxPro communicates with ISAPI.**
  Because ISAPI and CGI tend to be low-level interfaces that are coded in C++ or other low-level languages, these extensions are frequently used to build middleware software pieces. They call on external tools such as Visual FoxPro or Visual Basic, or internally implement a scripting engine like Active Server. FoxISAPI and Active Server Pages, both of which are discussed in this book, are implemented as ISAPI extensions. Tools like Web Connection, Cold Fusion, and some of the FrontPage extensions are also implemented as ISAPI and/or CGI extensions that abstract the low-level HTTP logistics.

- **COM is the messaging mechanism.**
  With Microsoft technologies, the mechanism for connecting back-end applications tends to

be COM. The connector scripts act as COM clients that call COM objects by passing parameters and retrieving return values. Using this mechanism it's possible to call a Visual FoxPro Automation server directly from a Web page and return Web content to the server.

Typical HTML-based Web applications follow these steps:

1. The user clicks a link or HTML form button in the Web browser.
2. The browser sends a request over the wire to the Web server via HTTP.
3. The Web server decodes the URL that the browser sent as part of its request, along with any additional information that the browser provides. Typically this will be HTML form variables or *Extra Headers* that tell the server to perform certain actions based on the request.
4. The server is now responsible for deciding what to do with the request. If it's a static document (HTML, graphic or other file content type) it simply pulls the file from disk and sends it back to the client. If it's a request for an extension, the server loads the extension, if not already loaded, and passes control to the extension.
5. The extension can process the request directly—it is possible to write ISAPI or CGI extensions in C if you have the need for speed or the patience to debug system level software. More commonly, the extension passes control to a back-end application, which could be a scripting engine like Active Server Pages or a back-end application called via COM, as is the case with FoxISAPI.
6. Once the back end application gets control, it is free to perform business logic, including providing access to databases and business objects. If you're using Visual FoxPro code directly with FoxISAPI, you're free to use any FoxPro code as long as no modal user interface code is run. When complete, the application returns a result (typically a string) to the ISAPI extension. Although Active Server Pages appears to be part of the IIS Web server, it really is also an ISAPI extension that implements a scripting engine. The ISAPI extension calls the engine, which calls a rich language library that allows access to both internal and external COM objects. The result is the same: The script essentially creates HTML output that's sent back to the Web server by the ISAPI extension.
7. Once the Web server receives the result from the extension, it passes that result over the HTTP link back to the browser.
8. The browser displays the content.
9. If the content contains scripting code, that code fires once the page has loaded on the client.
10. If using Dynamic HTML, code can continue to run once the page has finished loading, allowing access to all user interface elements of the HTML document.
11. If the HTML document also contains references to ActiveX controls or Java applets, these are either loaded from the local machine, if available, or automatically downloaded (with a prompt) from the Web via a CODEBASE tag in the document. The HTML document contains only an embedded *reference* to the object rather than the actual document. Based on the reference, the browser knows what to load or download.
12. Once loaded, the controls and applets can also be controlled by script code or events that fire in the control.
13. If the page loaded was an Active Document, the application in question is started and hosted inside the browser. The application partially takes over the browser's UI, with the

client area being the "desktop" for the application and the menu bar allowing access for extension by the client application.

In this scenario, all data is accessed by going from the browser through HTTP to the Web server, through an ISAPI extension to a back-end application, which finally gets the data and sends back a result to the client in the form of HTML. This is the typical scenario that you see in today's dynamic, data-driven Web sites. While this process seems complex and potentially slow, it is a proven mechanism for providing access to data in online Web applications.

You can bypass this long way using custom client and server implementations or by using special data components, but it will sidestep standard protocols and typically involve getting your hands dirty in low-level server programming, which is exactly what the high-level HTTP protocol aims to eliminate. The key is that the "custom" part of this cycle is the code you create in the COM component that responds to a Web request. This code is not difficult to write, but it is different in that you retrieve your inputs from specialized objects and generate output as HTML or raw HTTP content. The rest of the process is wrapped up in the HTTP protocol and you don't have to worry about it how it works.

A closer look at the server side

When dealing with Visual FoxPro servers on the back end, the Web server needs to hand off control from its own internal processing to the Visual FoxPro back-end application. **Figure 2.2** illustrates how this process works on the server.

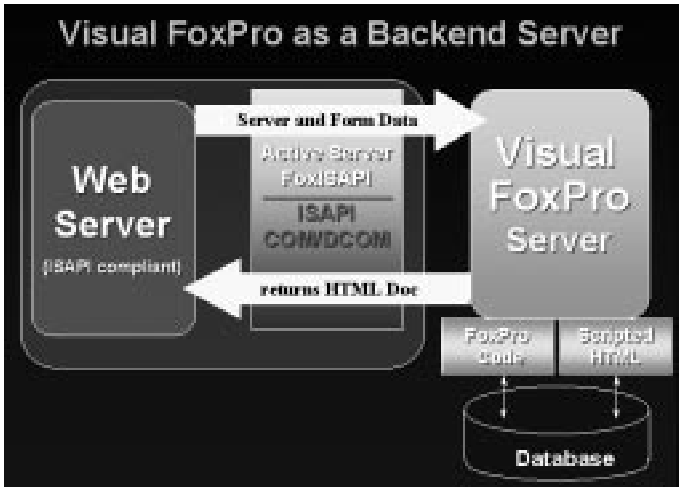

***Figure 2.2**. Internet Information Server communicates with a back-end application using ISAPI. When using VFP, the mechanism to reach VFP is COM. The VFP server is called and VFP responds by supplying either HTML/HTTP output directly or by providing business logic processing to a script page.*

ISAPI, the high-performance extension interface

The Web server receives the request from the browser. If the server is Internet Information Server (IIS) or another Microsoft or ISAPI-compliant server, it will pass the request to an ISAPI extension. ISAPI extensions are:

- **High-performance system-level libraries**
  ISAPI extensions are Win32 DLLs written in C++ or another low-level language. They tend to provide system-level code that handles the logistics of passing control to other back-end applications like Visual FoxPro or a script engine like Active Server Pages. However, ISAPI extensions can also be used to build application-level logic if you're willing to build applications in C++ and debug them in a hostile DLL environment hosted inside the Web server.

- **Hosted persistently inside the Web server's process**
  ISAPI extensions load into the Web server's address space and remain there. They essentially become part of the Web server itself. Once loaded, ISAPI extensions cannot be unloaded except by shutting down the Web server or an individual Virtual Application (more on that in the next chapter). This is the opposite of CGI, which creates a new process for each incoming request, runs the request and then unloads the process. This is much less efficient than a preloaded ISAPI extension. IIS also provides a configurable ceiling for how many ISAPI threads the process can create. This is to limit the amount of resources that are allocated to processing ISAPI requests to prevent overloading the Web server when traffic gets heavy.

- **Multithreaded**
  ISAPI extensions are multithreaded, which means they can process multiple requests at the same time. The same DLL can service multiple clients efficiently because the extension is already loaded in memory. But multithreading also makes programming ISAPI a tricky proposition because all code written must be thread-safe to avoid simultaneous access to common global data from different threads. One thing to keep in mind is that although the ISAPI extension is multithreaded, back-end applications that interact with ISAPI extensions may not be. For example, Visual FoxPro and Visual Basic are not multithreaded, so when servers built with these tools are called from ISAPI extensions directly, a simulation of multithreading must be implemented.

Remember that you're not likely to write ISAPI extensions yourself, but rather you'll interact with them indirectly through a back-end application or scripting engine.

Calling all Visual FoxPro servers

So how do you actually get control to your Visual FoxPro code from the Web server? There are a number of ways, but in this book I'll look at two approaches:

- **Scripting engine**
  ISAPI extensions can be used to implement a scripting engine that makes it possible to use simple text-based documents to mix code and HTML in the same page. Microsoft Active

Server Pages and Cold Fusion are examples of engines that implement scripting engines on top of ISAPI extensions. (Cold Fusion is actually implemented in ISAPI, NSAPI and plain CGI.) The scripting engine makes it possible to access databases via ODBC directly from the scripting code, or allows extension of the scripting code by calling COM objects. You can access Visual FoxPro data directly with ODBC or choose to access business logic contained in Visual FoxPro COM objects that are loaded and called from a script page. Scripting engines work by assigning a script map to an ISAPI DLL that causes the specified extension (like .ASP) to be routed to an ISAPI DLL (like ASP.dll) that implements the scripting engine. Based on the URL, the ISAPI DLL can figure out where the page is located and then parse and evaluate the scripting code in the page.

- **ISAPI → back-end application communication**
  Another, more traditional mechanism is an ISAPI extension calling a back-end application directly. Examples of this mechanism are FoxISAPI and third-party tools like Web Connection, FoxWeb and X-Works. Here the Web server simply packages all the request information available and sends it to the back-end application. The actual mechanics of how this is accomplished differ from using COM to instantiate and call servers, to using DDE, or using a file-based polling mechanism to send and receive the request information and result content. Either way, the Web server passes request information (HTML Form variables, server status, browser information, and so on) to the back end, which returns a result in the form of an HTTP response or a specific result value. There are two advantages to using this implementation: Direct access means better performance and full control of input and output. If FoxPro is your back-end server you can take advantage of the FoxPro language and data access directly. You can also bring up forms for HTML rendering and even implement a scripting engine inside the Visual FoxPro code. I'll show some of these techniques in Chapter 5.

Whether you call your Visual FoxPro servers from scripts or using the direct back-end server approach, once your code gets control, all of Visual FoxPro's power is available to you to perform complex operations. You can access existing business objects and classes and use the VFP data engine, which continues to be the fastest, most flexible way to access data.

Microsoft's Component Object Model— The glue tying Windows to the Web

In case you haven't noticed, COM is Microsoft's religion. Everything coming out of Microsoft these days is implemented using COM interfaces. This is great for developers because the technologies are inherently accessible from any environment that can be a COM client. COM is a mechanism for abstracting objects and binary code with logical interfaces. Interfaces decouple the binary code from the rules of the interface—the calling conventions. This makes it possible for otherwise binary-incompatible applications to share functionality. For example, you can't call FoxPro code directly from a Visual Basic application. But if you create a COM object out of your Visual FoxPro class, that object is now available to be called by the VB application, or Visual Basic for Applications in Word or Excel, or Delphi or Visual C++.

COM is built around the concept of creating components and building applications in logical pieces that communicate with each other, rather than creating one big monolithic executable. This is important on the Web in particular because it allows you to build components in disparate development environments and then integrate them into a single application via COM. This pushes the concept of using the best tool for the job rather than doing everything with the same tool. COM makes the implementation phase easy by providing full-featured development environments like Visual FoxPro or Visual Basic to build COM objects, with minimal changes from traditional code models.

Microsoft is actively pushing COM by exposing most of its new technologies through COM to allow universal access from any COM-capable client, including support for invoking objects across the network. There are system services like Active Directory (for managing system resources such as the directory and security structures, IIS Web server administration, and so on) and ActiveX Data Objects (ADO), which is a system data access engine. Then there are ActiveX controls, which are simply a set of specialized COM interfaces. Even business productivity applications like Word and Excel are COM servers that can be accessed from other applications and scripted through them. Microsoft's Web browser also exposes a rich COM object model that allows browser functionality to be integrated directly into applications. Expect this trend to continue: Windows 98 and Windows 2000 (NT 5.0) will expose almost every aspect of the Windows operating system via COM interfaces that can be scripted with the Windows Scripting Host, a new batch language (using VBScript or JavaScript) that allows access to system objects.

COM is extremely important in Microsoft's Internet strategy because the extensibility of Microsoft tools is built around it. This includes server engines like Active Server Pages and FoxISAPI, which can be extended by calling COM objects. As you've seen in the previous section, COM can be used to hook Web back-end applications directly out of the ISAPI layer. COM is also used for extending Active Server Pages with system functionality. COM is used as the binding mechanism for RDS, which consists of a client-hosted ActiveX control and a server-hosted COM object marshaled over HTTP. The two objects communicate using COM over HTTP. In short, COM is everywhere when you work with Microsoft Internet technology.

COM and Visual FoxPro

The good news is that COM integration with Visual FoxPro is easy and almost transparent if you're already familiar with Visual FoxPro. Visual FoxPro can be both a COM client and server. COM client support has been in Visual FoxPro since version 3.0 and includes both the ability to call COM servers like Word or Excel from an application as well as the ability to host ActiveX controls. Creating COM objects makes it possible for Visual FoxPro classes to expose any PUBLIC methods and properties to COM client applications.

To create a COM object in VFP, you simply need to know how to create a class and add a special OLEPUBLIC keyword to the class definition (or check the OLEPUBLIC flag in the Class Info dialog):

```
DEFINE CLASS SimpleServer as Custom OLEPUBLIC

FUNCTION HelloWorld
LPARAMETER lcName
RETURN "Hello " + lcName + "! The time is: " + TIME()

ENDDEFINE
```

Compiling this little class definition into an EXE or COM DLL will cause Visual FoxPro to create a COM object that can be called from any COM client. To call your COM object in VFP you can do this from the command window:

```
loServer = CreateObject("SimpleServer.Simple")
? loServer.HelloWorld("Rick")
```

Not so useful, since you could also call the class directly. But you can also call your COM object from a Visual Basic application:

```
SET loServer = CreateObject("SimpleServer.Simple")
lcHello = loServer.HelloWorld("Rick")
```

If you're ready to take the object onto the Web, you can call it from an Active Server Page (if you plan to use ASP make sure to compile the server to a DLL):

```
<HTML>
<% SET loServer = CreateObject("SimpleServer.Simple") %>
<%= loServer.HelloWorld("Rick") %>
</HTML>
```

As you can see, COM servers are simple to implement from the Visual FoxPro perspective. Calling these servers from other applications is also very straightforward and not all that different from the way you call the same object in Visual FoxPro natively.

The basics of COM are easy to deal with in Visual FoxPro. Complexities come into play when you start building applications that call your servers in high-volume environments. Any server application that expects high volumes will have to deal with load balancing and issues of making sure that the resources on the server are used efficiently. Technologies like Microsoft Transaction Server and Microsoft Message Queue become important in balancing resource use and handling server availability for creating applications that are scalable and provide reliable service. Because Visual FoxPro can build only single-threaded servers, there are some special issues involved in scaling Visual FoxPro COM objects. I'll introduce some of these issues in Chapter 4, and discuss them in more detail in Chapters 9 and 10.

What else do you need to know?

This chapter has gone over the terminology and reviewed the data flow between client and server. I've touched on what I think are the most important aspects that you should be familiar with before diving into the individual technologies and getting your hands dirty with code. The terminology can be a little overwhelming at first, but after reading through this chapter you should feel confident when you see certain terms come up again in the context of the discussion.

There is much more to discuss in terms of technology. I consciously deferred discussion of specific technologies that are more implementation-specific to the appropriate chapters. You will find out about ADO in the next chapter, and about RDS in Chapter 8. These technologies are important, but they are not essential to the infrastructure and can be better covered in the context of actually being used. I'll also expand on using COM with Visual FoxPro as we get into building COM objects.

This chapter has dealt with the infrastructure. The next chapter will take you through the configuration issues of setting up your server and development environment.