4 The Web Service Interface

This chapter describes the architecture of the AGS Web service and how the Java and Perl APIs access it. It also explains the message syntax for sending requests directly to the AGS Web service.

This chapter contains the following sections:

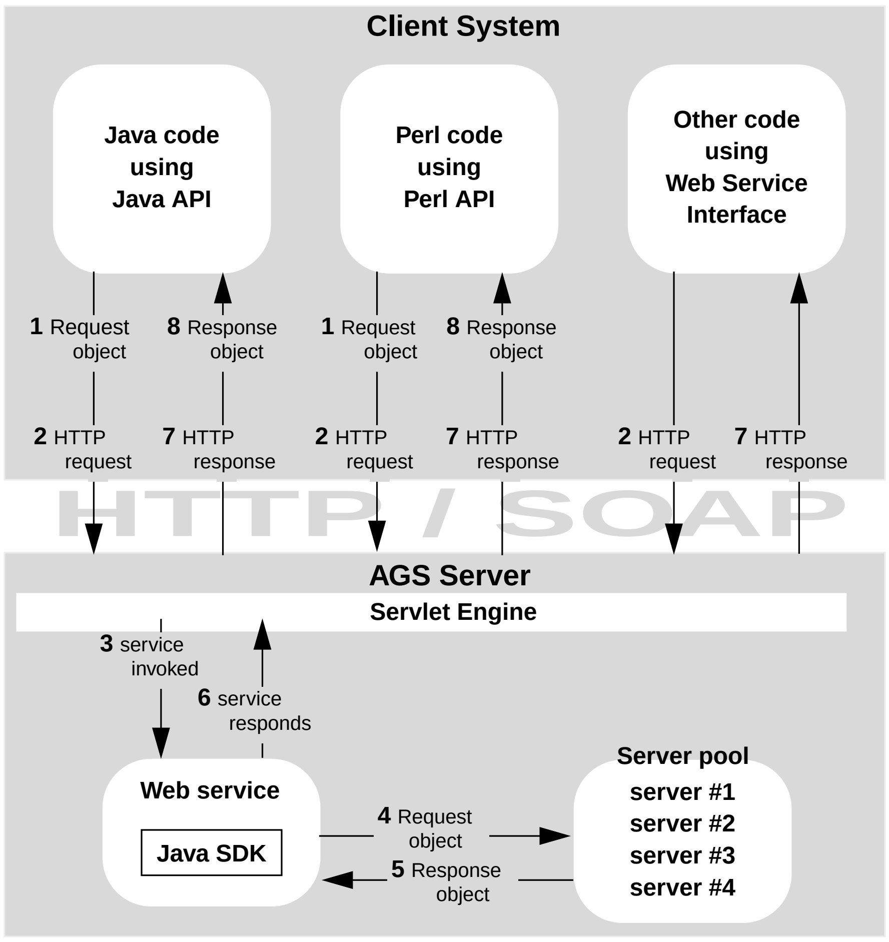
The Web Service Architecture

The Java and Perl APIs both provide access to the Web service with the `createWebServiceServer` method. When you send a request to such a Web service Server object, the details explained in this chapter are taken care of behind the scenes. The `execute` method in these cases deconstructs the information in the Request object, translates it into an HTTP message that conforms to the SOAP protocol, and sends it to the URL specified when the Server object was created. When the AGS server receives such a request, it reconstructs the Request object and sends it to a pooled AGS process for execution. When the request is processed, the Web service deconstructs the Response object and reformats the information into an HTTP response that also conforms to the SOAP protocol. This process is shown in Figure 4.1 on page 80.

instead of using the Java or Perl APIs to access the Web service, you may wish to use WSDL–capable tools to construct your own requests and send them directly to the Web Service. The information in this chapter enables you to do that.

The following figure shows how the Java and Perl APIs provide a connection to the AGS Web service, as well as showing a direct connection.

FIGURE 4.1 **Communicating with the AGS Web service**



The following steps explain Figure 4.1 in detail. The step numbers correspond to the numbers shown in the figure.

1. *Creating a Server and a Request object* — This step applies only when using the Java or Perl API to contact the Web service.

In your Java or Perl code, you create a Web service Server object by calling the `createWebServiceServer` method. This method takes one argument, the URL of the AGS Server. This URL is determined when you install AGS. You can create a secure connection to the AGS Server by supplying a URL that begins with https://. If you are using an LDAP server, you must also supply a user name and password in the request.

The `createWebServiceServer` method returns a `Server` object in Java and a `ServerWebService` object in Perl. You then create a `Request` object and put the desired information into it. You then call the `execute` method of the Server object, passing in the `Request` object as a parameter.

2. *Creating a Web service request* — The implementation of the Java and Perl APIs transparently converts the `Request` object you pass to the server's `execute` method into an appropriately constructed Web service request.

   If you are not using the Java or Perl APIs, but are using some other coding mechanism to access the AGS server, you must construct this Web service request yourself. The request must contain all the information necessary for the Web service to process your request. Chapter 4, "The Web Service Interface," describes the contents of such a request, and contains all the information you need to construct one. You can also discover this information by asking AGS for its WSDL description.

3. *Transmitting the Web service request* — If you are using the Java or Perl API as an entry point, the API implementation automatically routes the request to the appropriate host and port number on which AGS is listening, based on the URL you provided when you created the Server object.

4. *Executing the Web service request* — When the Web service receives the request, it checks the header for which server method to invoke (`execute` in this case) and passes the request to an AGS process to execute.

5. *Receiving a response from the* AGS *process* — When the AGS process is finished processing the request, it returns a `Response` object to the Web service. If you specified a result location, the result files reside on the AGS server's system; otherwise, result content is returned in the response.

6. *Creating a response for the client* — The Web service constructs an appropriate response.

7. *Transmitting the response* — The Web service sends the response back to the client. The error code in the response indicates whether the request succeeded or failed.

   If the request succeeded, the response contains log information and possibly result content. The structure and content of such a response is described in Chapter 4.

   If the request failed, error information is included in the response in place of results. The structure and content of such a response is described in Chapter 4.

   If your entry point was the Java or Perl API, you do not need to be concerned about the content of this response, as the API processes it for you. If you have used other code as your entry point, however, you are responsible for parsing this response yourself.

8. *Returning a Response object* — If your entry point was the Java or Perl API, the implementation of the Server object's `execute` method extracts the pertinent information from the Web service's response and puts it into an API `Response` object, which is returned from the method. This step does not apply if your entry point is from other code.

In summary, the basic mechanism of encoding AGS request information in an HTTP/SOAP message is identical no matter whether you invoke AGS with the Java API, the Perl API, or from other code. The main difference is that the Java and Perl APIs completely hide the fact that an HTTP/SOAP request/response cycle is involved.

Accessing the WSDL for the Web Service

If you choose to construct your own HTTP requests for the AGS Web service, you can access its WSDL with the following URL:

http://*localhost*:8019/altercast/AlterCast

where *localhost* is the name of the system on which the Web service is installed.

This link takes you to the Adobe Server Web Services page. From here, you can access the service description (the WDSL). There are also two other links that show you the request and response syntax for a get version and an execute request.

The AGS SDK includes C# and .NET examples that demonstrate how to call the AGS Web service directly, using the WSDL. See the directory `samples/api/NET` wherever you install the SDK.

A Brief Description of SOAP

The AGS request information in a request to the Web service is enclosed in a SOAP envelope. SOAP (Simple Object Access Protocol) defines a uniform way to pass XML–encoded data and to perform remote procedure calls using HTTP. The protocol defines three things:

- *The SOAP envelope* — A framework for describing where to deliver a message and how to process it. The SOAP envelope is the top–level element of any XML document that represents a SOAP message.
- *The SOAP encoding* — A set of encoding rules to express instances of application–defined data types. This encoding includes information about which parameters are allowed in the message, what each parameter’s type is, and how the parameters can be used together. AGS does not use this encoding.
- *SOAP transmission over HTTP* — A convention for representing remote procedure calls and responses. While the SOAP protocol can potentially be used in combination with a variety of other protocols, the implementation of the AGS Web service uses it with HTTP. A SOAP request can be transmitted over HTTP, typically within a POST request. The request is given the media type `text/xml` or multipart MIME and its body contains the SOAP envelope. Optionally, the request may contain a header called `SOAPAction` that indicates the intent of the request.

The AGS Web service fully supports the specifications for the SOAP envelope and SOAP transmission over HTTP.

For more information about SOAP, see http://www.w3.org and follow the link for “Web Services”.

Syntax for AGS Requests and Responses

The examples in this section show prototypical syntax for requests and responses. Values in red italics are placeholders for actual values in a real request or response. You can get this information from the WSDL, but it is presented here to provide context for some additional information.

Requests for AGS Processing

The HTTP request that expresses an AGS request consists of:

- A standard HTTP `POST` method.
- A special header called `SOAPAction` with a value of `"Execute"`. This causes the AGS Web service to invoke the `execute` method.
- A SOAP envelope that contains the AGS request information.

The following code shows a full HTTP request that expresses an AGS request for processing. All items in (red) italics in the following example are placeholders for actual values you supply. All string values in the request should be properly encoded using their Unicode codes, as required by XML.

Example 4.1 An HTTP/SOAP request for AGS processing

```
1  POST /altercast/AlterCast HTTP/1.1
2       Host: localhost
3       Content-Type: text/xml; charset=utf-8
4       Content-Length: length
5       SOAPAction: "http://ns.adobe.com/altercast/1.5/Execute"
6
7       <?xml version="1.0" encoding="utf-8"?>
8       <soap:Envelope
9             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
10            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
11            xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
12        <soap:Body>
13          <request xmlns="http://ns.adobe.com/altercast/1.5/">
14            <commands>string</commands>
15            <files>
16              <file>
17                <name>string</name>
18                <data>base64Binary</data>
19              </file>
20              <file>
21                <name>string</name>
22                <data>base64Binary</data>
23              </file>
24            </files>
25            <variables>
26              <variable>
27                <name>string</name>
28                <value>string</value>
29              </variable>
30              <variable>
31                <name>string</name>
32                <value>string</value>
33              </variable>
34            </variables>
35          </request>
36        </soap:Body>
37      </soap:Envelope>
```

- Lines 1-5 contain the HTTP header information. Line 5 specifies the special `SOAPAction` header.

The remaining lines define the SOAP envelope that contains the AGS request information:

- Lines 8-11 contain the standard SOAP attributes for the `soap:Envelope` element.
- Line 13 starts the AGS request portion of the body.
- Line 14 specifies the AGS commands for the request. The example shows how to specify the commands as a string within the request body. This single line will be replaced by multiple actual lines of text, depending on how many commands you need to specify in the request.

The basic requirement for a command set is that it be nested inside a `commands` element. When submitting requests directly to the Web service, you need an additional command element, nested inside the outer one and appropriately escaped for XML. All text within this internal `commands` element must also be escaped. The following code is an example of a short commands string that would replace line 14 in an actual request.

```
<commands>
    &lt;commands&gt;
       &lt;imageSize width=&quot;500&quot; height=&quot;300&quot;/ &gt;
    &lt;/commands&gt;
</commands>
```

- Line 15 starts the list of input files for this request. This element is followed by any number of `file` elements, each one specifying one input file. This example shows two such files. You specify the file data directly as a base-64 binary encoded string. For example, if the following command is specified in the command set:

  ```
  <loadContent source="flower">
  ```

  the `files` element should contains a corresponding element like this:

  ```
  <file>
     <name>flower</name>
     <data>...etc...</data>
  <file/>
  ```

  As an alternative to specifying the input as a string, you can specify the input with an attachment. The following URL describes the specification for including MIME attachments in SOAP messages: http://www.w3.org/TR/SOAP-attachments.
- Line 25 starts the list of variables for this request. This element is followed by any number of `variable` elements, each one specifying a single variable name/value pair. This data is read into a content holder called “data” and is used for automatic variable replacement and for explicit variable replacement with the `applyVariables` command, if you do not specify another source of input for the command.

Responses to Successful Processing

The HTTP request that expresses the response from an AGS request that was successfully processed consists of:

- *An error code* — The Web service returns a 200 (OK) when the request was successfully processed.
- *The results of the processing* — Results are enclosed in a SOAP envelope in the message body that contains the AGS response information. Log messages generated during request processing are always returned. Result files are returned in the response if you did not specify a result location in the AGS request or otherwise save the results to the file system.

The following code shows the syntax for a full HTTP response that AGS sends after it successfully processes a request. All items in (red) italics in the following example are placeholders for actual data values. Strings in the response are appropriately escaped for XML.

EXAMPLE 4.2 An HTTP/SOAP response indicating successful AGS processing

```
1  HTTP/1.0 200 OK
2  Content-Type:text/xml
3
4  <?xml version="1.0" encoding="utf-8"?>
5  <soap:Envelope
6          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
8          xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
9       <soap:Body>
10         <response xmlns="http://ns.adobe.com/altercast/1.5/">
11             <files>
12                <file>
13                   <name>string</name>
14                   <width>int</width>
15                   <height>int</height>
16                   <type>string</type>
17                   <extension>string</extension>
18                   <data>base64Binary</data>
19                </file>
20                <file>
21                   <name>string</name>
22                   <width>int</width>
23                   <height>int</height>
24                   <type>string</type>
25                   <extension>string</extension>
26                   <data>base64Binary</data>
27                </file>
28             </files>
29             <log>
30                <entry>string</entry>
31                <entry>string</entry>
32             </log>
33         </response>
34      </soap:Body>
35 </soap:Envelope>
```

- Lines 1-2 contain standard HTTP header information.
- Lines 5-8 contain the standard SOAP attributes for the `soap:Envelope` element.
- Line 10 starts the AGS response portion of the body.
- Line 11 starts the list of results. The `files` element is followed by any number of `file` elements, each one specifying one result file. Besides the result contents, returned as a base-64 binary encoded string, the result content’s name, MIME type, file extension, and

width and height in pixels are returned. The exact meaning of this information is the same as is explained in the Java, Perl, and COM API chapters in this guide.

If you included the input files as MIME attachments in your request, AGS returns any results included in the response as MIME attachments also.

If you specified a result location in the request, result files are written to the Web service's local file system and the `files` element is omitted.

- Line 29 starts the list of log messages related to this request. The `log` element is followed by any number of `entry` elements, each one specifying a single log string that AGS generated while processing the request. If no log messages were generated, the `log` element is omitted.

Responses to Unsuccessful Processing

The HTTP response that expresses the response from an AGS request that was unsuccessfully processed consists of:

- *An error code* — The Web service returns a 500 (Internal Server Error) when the request was not successfully processed.
- *Error information* — The error information is enclosed in a SOAP envelope in the message body.

The following code shows the syntax for a full HTTP response that AGS sends when it fails to successfully process a request. All items in (red) italics in the following example are placeholders for actual data values. Strings in the response are appropriately escaped for XML.

Example 4.3 An HTTP/SOAP response indicating an AGS processing failure

```
HTTP/1.0 500 Internal Server Error
Content-Type:text/xml

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
     <soap:Body>
        <soap:Fault>
           <faultcode>soap:Server</faultcode>
           <faultstring>Could not handle the request because reason
           </faultstring>
           <detail>
              <log xmlns="http://ns.adobe.com/altercast/1.5/">
                 <entry>[NOTICE] Starting AlterCast request.</entry>
                 <entry>[ERROR] Unknown command &apos;commandName&apos;.
                 </entry>
              </log>
           </detail>
        </soap:Fault>
     </soap:Body>
</soap:Envelope>
```

- Lines 1-2 contain standard HTTP header information.
- Lines 5-21 contain the SOAP body:
  - On line 9, the value of the `faultcode` element is always either `soap:Client` or `soap:Server`. The former indicates that the request the client sent is invalid in some way. The latter means the server failed to execute the request for some other reason.
  - Lines 12-18 contain detailed information about the failure. The `detail` element is followed by a `log` element that contains an `entry` element for each log message that AGS generated while it was processing the request.

5 The COM API

This chapter explains the details of the COM (Component Object Model) API for AGS. General information about the underlying API model and other information that applies to all of the AGS APIs is contained in Chapter 1, "The Adobe Graphics Server Interfaces."

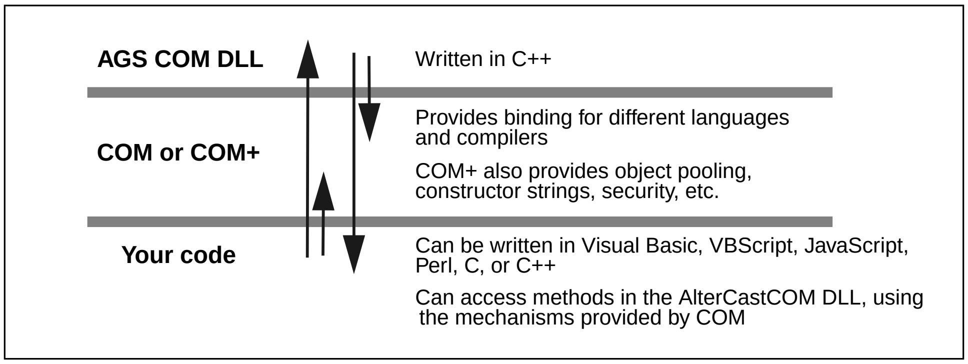
NOTE: The AGS COM API is supported only on Windows platforms.

This chapter contains the following sections. Within each section that details the contents of a COM interface, the methods are presented alphabetically:

An Overview of the COM API

NOTE: This section is not a general COM tutorial. We assume that if you are using this chapter, you already understand what COM is and have at least a basic working knowledge of how to use it.

Simply put, COM is a mechanism that allows you to build reusable binary components, and to exercise code written in one language from code written in another language. For example, the AGS COM library is written in C++, but you can access its methods from a variety of interpretive programming languages, such as Visual Basic, VBScript, JavaScript, and Perl, as well as some compiled languages, such as C and C++. COM provides a neutral language-independent binding between the code you write and the AGS code.


AGS COM DLL
Written in C++
COM or COM+
Provides binding for different languages and compilers
COM+ also provides object pooling, constructor strings, security, etc.
Your code
Can be written in Visual Basic, VBScript, JavaScript, Perl, C, or C++
Can access methods in the AlterCastCOM DLL, using the mechanisms provided by COM

The AGS COM API is supported only on Microsoft® Windows platforms: Windows NT® and Windows 2000. You can invoke the object constructors and API methods on either the local system or on a remote system, depending on how you have installed and configured AGS and on what platform(s) you are operating.

This chapter focuses on using the COM API from interpretive languages, but much of the information presented here is directly generalizable to the compiled languages as well. The differences lie in the actual method signatures presented by the AGS DLL.

Terms Used in This Chapter

Throughout this chapter, the following terms are used as follows:

COM	The version of COM that comes with Windows NT. This term is also sometimes used as a generic umbrella term for both COM and COM+. The context should make it clear which meaning is meant.
COM+	The version of COM that comes with Windows 2000. COM+ contains all the functionality of COM and some additional capabilities — object pooling, security, and constructor strings, to name a few. When you install AGS with its COM SDK option on a Windows 2000 system, the installer automatically creates a COM+ application configured for pooling. However, you are not bound to using the COM+ features on these platforms — you can simply use COM instead of COM+.
COM SDK	All the necessary files and executables that come with COM. This SDK is automatically installed with Windows NT.
COM+ SDK	All the necessary files and executables that come with COM+. This SDK is automatically installed with Windows 2000.
COM server	A generic term to cover a system on which either the COM or COM+ SDK has been installed.
AGS COM API	A collection of COM interfaces that define the API to AGS through both COM and COM+. When the immediate context makes it clear, this is sometimes shortened to the *COM API*.

The AGS COM library	The file called `AlterCastCOM.dll` which defines the interface and provides the implementation for the AGS COM API.
COM+ application	A group of related COM components that you can configure individually and as a whole, and that operate together in the COM+ environment. You can see the COM+ applications on a system from the control panel (**Start** > **Settings** > **Control Panel**) under Administrative Tools (**Component Services**> **Computers**>**My Computer** > **COM+ Applications**).

The API Interfaces

Conceptually, you can group the methods in the AGS COM API into three main groups:

- Servers — Methods you use to create a connection to an AGS server and to send requests via this connection.
- Requests — Methods you use to construct AGS requests.
- Responses — Methods you use to obtain the results of a processed AGS request that you have sent.

The AGS COM API consists of six interfaces and their corresponding objects, as shown in the following table:

TABLE 5.1 A summary of the COM API

Conceptual Area	COM Interface	COM Object	Methods

AGS Servers	`IACServer`	`ACServer`	`execute, getVersion`
			`addFontFolder`
			`setLogLevel, SetLogFile`

AGS Requests	`IACRequest`	`ACRequest`	`addData, addFile, addString`
			`addVariable`
			`setCommands, setCommandsFile`
			`setCommandsFileOnServer`
			`setResultLocation`
			`setResultOverwrite`

Results of Processed Requests	`IACResponse`	`ACResponse`	`getLog, getData`
	`IACLog`	`ACLog`	`count, item`
	`IACData`	`ACData`	`count, item`

	`IACRecord`	`ACRecord`	`getData, getName`
			`getExtension, getType`
			`getHeight, getWidth`

The only two objects in the above list that you need to create instances of are the `ACServer` and `ACRequest` objects — all the response–related objects are created by AGS when it finishes processing a request. The exact syntax for creating these objects varies, depending on the language you use to exercise the AGS COM API. All languages, however, give you the option of specifying a particular machine on which the object is created. There are some restrictions on where you can create the Server and Request objects, depending on the system configuration you are using. These restrictions are explained in Table 5.3 on page 94.

The AGS SDK includes samples that demonstrate how to use the methods discussed in this chapter. There are samples that use the COM API from C++, JavaScript, and VBScript. See the directory `samples/api/com` wherever you install the SDK.

General Order of Method Use

The general order in which you use the methods in the COM API is as follows:

1. *You create an* `ACServer` *object*. Exactly how you do this is determined by the programming language you are using.
2. *You create a* `ACRequest` *object*. Exactly how you do this is determined by the programming language you are using. You then use the Request object's methods to populate the request.

NOTE: Steps 1 and 2 are interchangeable — it doesn't matter whether you create the server or the request first.

3. *You send the request to the server*. You pass your `ACRequest` object to the `ACServer` object's `execute` method.
4. *You get the results of the processing done in response to the request*. When the `execute` method terminates, it returns an `ACResponse` object. You extract the collection of results and the collection of log messages from this object, and then extract individual results or messages from these collections.

   Since the Server object persists after it handles a request, you can repeat steps 2 - 4 any number of times, using the same Server object.

Local and Remote Execution

The system on which you create the `ACServer` object determines the system on which AGS executes the requests you send to this server. The nature of COM allows you, under certain circumstances, to create the Server object on one system and your request objects on a different system, one or both of which may not be the same system on which your program that creates these objects is running. The following sections explain all of this.

Local Execution

Table 5.2 explains the system configuration needed to execute AGS requests on the same system on which your program runs.

TABLE 5.2 Using the AGS COM API for local execution of requests

Windows platform	**Requirements for using the AGS COM API**
NT	*Installation requirements*: • The AGS COM library is registered with Windows. The AGS installer does this for you. No further setup or configuration is necessary. *Execution requirements*: • Create all of your `ACServer` and `ACRequest` objects on the local system.
2000	*Installation requirements*: • AGS is installed with the COM SDK option on the local system. • The AGS COM library is registered with Windows. The AGS installer does this for you. • You can use either COM or COM+ on this platform. If you wish to use COM+, a COM+ application for AGS must exist and be appropriately configured. The AGS installer does this automatically for you when you install on Windows 2000. You can, however, change the configuration of the application once it is created. *Execution requirements*: • Create all of your `ACServer` and `ACRequest` objects on the local system.

Remote Execution

If you plan to run any part of AGS on a system other than the one on which your program runs, some restrictions apply. You may want to run your program on your local system, but create your request objects on another system, because that is where all the files you will add to your requests exist. Or you may want to run the program locally but run the AGS server on another system, for a variety of reasons. Your client system may not have sufficient hardware resources to run AGS. You may have many client systems, not all of which have AGS installed or licensed. The client and AGS may both be resource intensive, causing them to compete for resources unless run on separate systems.

Once you introduce remote execution, you introduce two sets of variables: the operating system on the local and remote machines, and whether you are using COM or COM+ on those machines. The restrictions that apply to remote execution can best be explained in the context of whether you are using COM or COM+. Table 5.3 is organized on this principal. When using this table, keep in mind the following restrictions:

- You can use COM on both Windows NT and Windows 2000.
- You can use COM+ only on Windows 2000.

In Table 5.3:

- The **Local** column identifies whether COM or COM+ is being used on the local machine. This is the machine on which your program that exercises the AGS COM API is running.
- The **Remote** column identifies whether COM or COM+ is being used on the remote machine(s).

TABLE 5.3 Using the AGS COM API for remote execution of requests

Local	Remote	Execution requirements and capabilities
COM ➡	COM	All `ACServer` and `ACRequest` objects must be created on a single remote machine. No objects can be created locally.
COM+ ➡	COM	
COM ➡	COM+	All `ACServer` and `ACRequest` objects must be created on one or more remote machines. This means you could create an `ACServer` object on one machine and an `ACRequest` object on a different machine. However, no objects can be created on the local machine.
COM+ ➡	COM+	You can create any combination of `ACServer` and `ACRequest` objects, all remotely on the same machine, all remotely on multiple machines, or some locally and some remotely on one or more machines. This scenario obviously gives you the most flexibility.

Method Signatures

The COM API can be used from a variety of languages, both interpretive and compiled. Because the terminology, syntax, and built–in types of these languages vary, we have chosen to use particular terms and types in a general sense.

We use the term *method* to cover C functions, C++ methods, Perl subroutines, etc. The method prototypes given in the synopsis sections in this chapter are presented in a generic form that looks much like Visual Basic, and uses generic names for common built–in types. The following table shows the type equivalency between this generic type and the actual types in the supported languages.