Chapter 16. Networking

.NET offers a variety of classes in the `System.Net.*` namespaces for communicating via standard network protocols, such as HTTP and TCP/IP. Here's a summary of the key components:

- `HttpClient` for consuming HTTP web APIs and RESTful services
- `HttpListener` for writing an HTTP server
- `SmtpClient` for constructing and sending mail messages via SMTP
- `Dns` for converting between domain names and addresses
- `TcpClient`, `UdpClient`, `TcpListener`, and `Socket` classes for direct access to the transport and network layers

The .NET types in this chapter are in the `System.Net.*` and `System.IO` namespaces.

> **NOTE**
>
> .NET also provides client-side support for FTP, but only through classes that have been marked as obsolete from .NET 6. If you need to use FTP, your best option is to reach for a NuGet library such as FluentFTP.

Network Architecture

Figure 16-1 illustrates the .NET networking types and the communication layers in which they reside. Most types reside in the *transport layer* or *application layer*. The transport layer defines basic protocols for sending and receiving bytes (TCP and UDP); the application layer defines higher-level protocols designed for specific applications such as retrieving web

pages (HTTP), sending mail (SMTP), and converting between domain names and IP addresses (DNS).

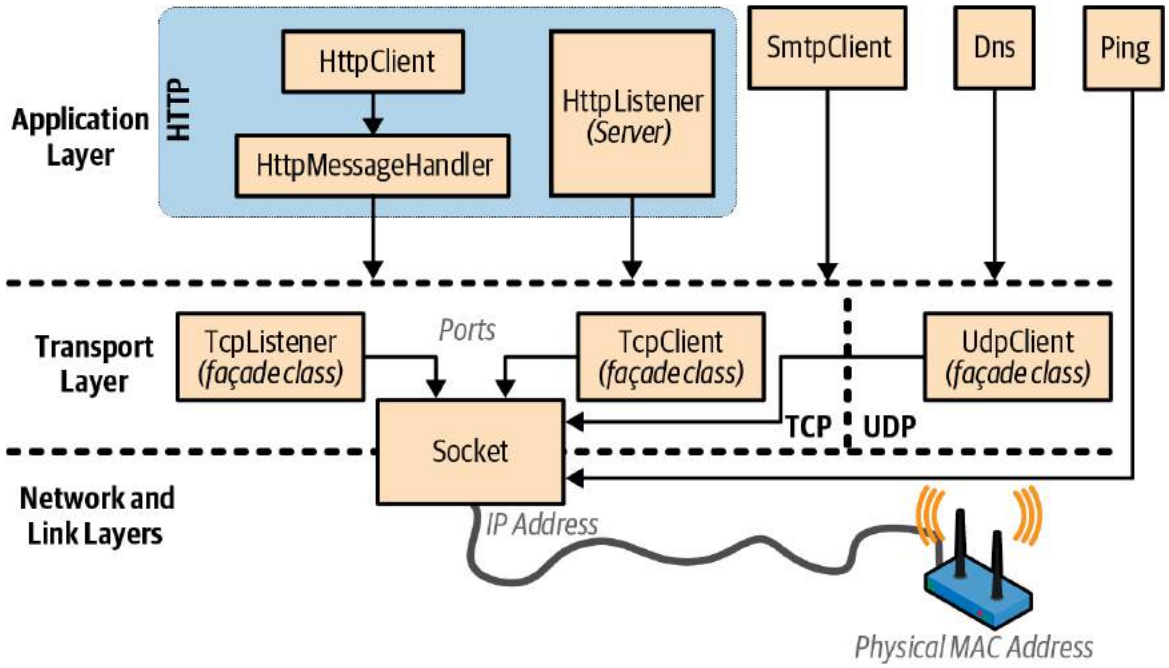

Figure 16-1. Network architecture

It's usually most convenient to program at the application layer; however, there are a couple of reasons why you might want to work directly at the transport layer. One is if you need an application protocol not provided in .NET, such as POP3 for retrieving mail. Another is if you want to invent a custom protocol for a special application such as a peer-to-peer client.

Of the application protocols, HTTP is special in its applicability to general-purpose communication. Its basic mode of operation—"give me the web page with this URL"—adapts nicely to "get me the result of calling this endpoint with these arguments." (In addition to the "get" verb, there is "put," "post," and "delete," allowing for REST-based services.)

HTTP also has a rich set of features that are useful in multitier business applications and service-oriented architectures, such as protocols for authentication and encryption, message chunking, extensible headers and cookies, and the ability to have many server applications share a single port and IP address. For these reasons, HTTP is well supported in .NET—both

directly, as described in this chapter, and at a higher level, through such technologies as Web API and ASP.NET Core.

As the preceding discussion makes clear, networking is a field that is awash in acronyms. We list the most common in Table 16-1.

Table 16-1. Network acronyms

Acronym	Expansion	Notes
DNS	Domain Name Service	Converts between domain names (e.g., *ebay.com*) and IP addresses (e.g., 199.54.213.2)
FTP	File Transfer Protocol	Internet-based protocol for sending and receiving files
HTTP	Hypertext Transfer Protocol	Retrieves web pages and runs web services
IIS	Internet Information Services	Microsoft's web server software
IP	Internet Protocol	Network-layer protocol below TCP and UDP
LAN	Local Area Network	Most LANs use internet-based protocols such as TCP/IP
POP	Post Office Protocol	Retrieves internet mail
REST	REpresentational State Transfer	A popular web service architecture that uses machine-followable links in responses and that can operate over basic HTTP
SMTP	Simple Mail Transfer Protocol	Sends internet mail

Acronym	Expansion	Notes
TCP	Transmission and Control Protocol	Transport-layer internet protocol on top of which most higher-layer services are built
UDP	Universal Datagram Protocol	Transport-layer internet protocol used for low-overhead services such as VoIP
UNC	Universal Naming Convention	*\\computer\sharename\filename*
URI	Uniform Resource Identifier	Ubiquitous resource naming system (e.g., *http://www.amazon.com* or *mailto:joe@bloggs.org*)
URL	Uniform Resource Locator	Technical meaning (fading from use): subset of URI; popular meaning: synonym of URI

Addresses and Ports

For communication to work, a computer or device requires an address. The internet uses two addressing systems:

IPv4

> Currently the dominant addressing system; IPv4 addresses are 32 bits wide. When string-formatted, IPv4 addresses are written as four dot-separated decimals (e.g., 101.102.103.104). An address can be unique in the world—or

unique within a particular *subnet* (such as on a corporate network).

IPv6

The newer 128-bit addressing system. Addresses are string-formatted in hexadecimal with a colon separator (e.g., [3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]). .NET requires that you add square brackets around the address.

The `IPAddress` class in the `System.Net` namespace represents an address in either protocol. It has a constructor accepting a byte array, and a static `Parse` method accepting a correctly formatted string:

```
IPAddress a1 = new IPAddress (new byte[] { 101, 102, 103, 104 });
IPAddress a2 = IPAddress.Parse ("101.102.103.104");
Console.WriteLine (a1.Equals (a2));                    // True
Console.WriteLine (a1.AddressFamily);                  // InterNetwork

IPAddress a3 = IPAddress.Parse
  ("[3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]");
Console.WriteLine (a3.AddressFamily);   // InterNetworkV6
```

The TCP and UDP protocols break out each IP address into 65,535 ports, allowing a computer on a single address to run multiple applications, each on its own port. Many applications have standard default port assignments; for instance, HTTP uses port 80; SMTP uses port 25.

NOTE

The TCP and UDP ports from 49152 to 65535 are officially unassigned, so they are good for testing and small-scale deployments.

An IP address and port combination is represented in .NET by the `IPEndPoint` class:

```
IPAddress a = IPAddress.Parse ("101.102.103.104");
IPEndPoint ep = new IPEndPoint (a, 222);          // Port 222
Console.WriteLine (ep.ToString());                // 101.102.103.104:222
```

> **NOTE**
>
> Firewalls block ports. In many corporate environments, only a few ports are open—typically, port 80 (for unencrypted HTTP) and port 443 (for secure HTTP).

URIs

A URI is a specially formatted string that describes a resource on the internet or a LAN, such as a web page, file, or email address. Examples include *http://www.ietf.org*, *ftp://myisp/doc.txt*, and *mailto:joe@bloggs.com*. The exact formatting is defined by the *Internet Engineering Task Force* (IETF).

A URI can be broken up into a series of elements—typically, *scheme*, *authority*, and *path*. The `Uri` class in the `System` namespace performs just this division, exposing a property for each element, as illustrated in Figure 16-2.

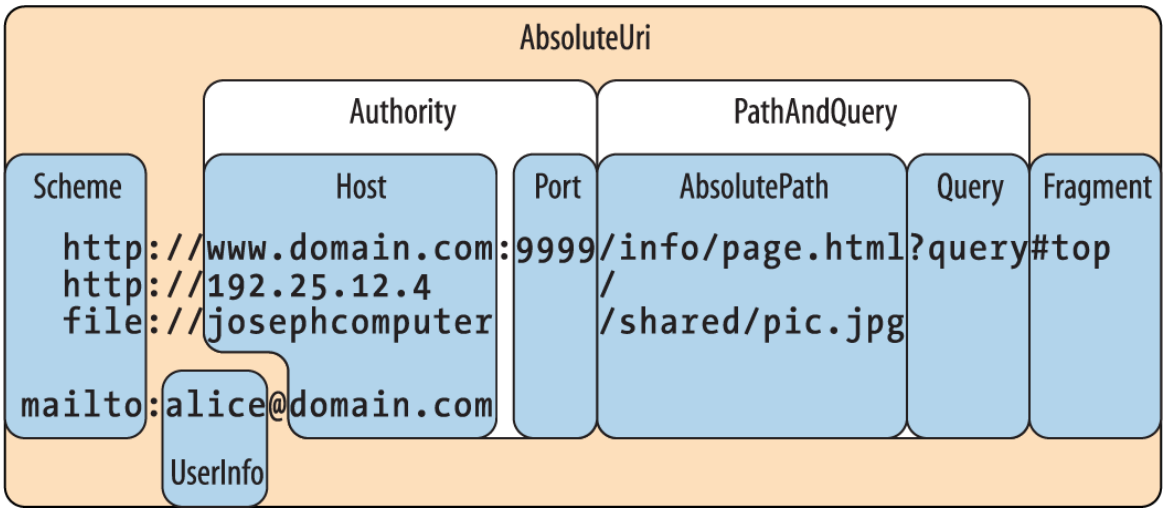

Figure 16-2. URI properties

> **NOTE**
>
> The `Uri` class is useful when you need to validate the format of a URI string or to split a URI into its component parts. Otherwise, you can treat a URI simply as a string—most networking methods are overloaded to accept either a `Uri` object or a string.

You can construct a `Uri` object by passing any of the following strings into its constructor:

- A URI string, such as *http://www.ebay.com* or *file://janespc/sharedpics/dolphin.jpg*
- An absolute path to a file on your hard disk, such as *c:\myfiles\data.xlsx* or, on Unix, */tmp/myfiles/data.xlsx*
- A UNC path to a file on the LAN, such as *\\janespc\sharedpics\dolphin.jpg*

File and UNC paths are automatically converted to URIs: the "file:" protocol is added, and backslashes are converted to forward slashes. The `Uri` constructors also perform some basic cleanup on your string before creating the `Uri`, including converting the scheme and hostname to lowercase and removing default and blank port numbers. If you supply a

URI string without the scheme, such as *www.test.com*, a `UriFormatException` is thrown.

`Uri` has an `IsLoopback` property, which indicates whether the `Uri` references the local host (IP address 127.0.0.1), and an `IsFile` property, which indicates whether the `Uri` references a local or UNC (`IsUnc`) path (`IsUnc` reports `false` for a *Samba* share mounted in a *Linux* filesystem). If `IsFile` returns `true`, the `LocalPath` property returns a version of `AbsolutePath` that is friendly to the local OS (with slashes or backslashes as appropriate to the OS), on which you can call `File.Open`.

Instances of `Uri` have read-only properties. To modify an existing `Uri`, instantiate a `UriBuilder` object—this has writable properties and can be converted back via its `Uri` property.

`Uri` also provides methods for comparing and subtracting paths:

```
Uri info = new Uri ("http://www.domain.com:80/info/");
Uri page = new Uri ("http://www.domain.com/info/page.html");

Console.WriteLine (info.Host);     // www.domain.com
Console.WriteLine (info.Port);     // 80
Console.WriteLine (page.Port);     // 80  (Uri knows the default HTTP port)

Console.WriteLine (info.IsBaseOf (page));         // True
Uri relative = info.MakeRelativeUri (page);
Console.WriteLine (relative.IsAbsoluteUri);       // False
Console.WriteLine (relative.ToString());          // page.html
```

A relative `Uri`, such as *page.html* in this example, will throw an exception if you call almost any property or method other than `IsAbsoluteUri` and `ToString()`. You can directly instantiate a relative `Uri`, as follows:

```
Uri u = new Uri ("page.html", UriKind.Relative);
```

> **WARNING**
>
> A trailing slash is significant in a URI and makes a difference as to how a server processes a request if a path component is present.
>
> In a traditional web server, for instance, given the URI *http://www.albahari.com/nutshell/*, you can expect an HTTP web server to look in the *nutshell* subdirectory in the site's web folder and return the default document (usually *index.html*).
>
> Without the trailing slash, the web server will instead look for a file called *nutshell* (without an extension) directly in the site's root folder—which is usually not what you want. If no such file exists, most web servers will assume the user mistyped and will return a 301 *Permanent Redirect* error, suggesting the client retry with the trailing slash. A .NET HTTP client, by default, will respond transparently to a 301 in the same way as a web browser—by retrying with the suggested URI. This means that if you omit a trailing slash when it should have been included, your request will still work—but will suffer an unnecessary extra round trip.

The `Uri` class also provides static helper methods such as `EscapeUriString()`, which converts a string to a valid URL by converting all characters with an ASCII value greater than 127 to hexadecimal representation. The `CheckHostName()` and `CheckSchemeName()` methods accept a string and check whether it is syntactically valid for the given property (although they do not attempt to determine whether a host or URI exists).

HttpClient

The `HttpClient` class exposes a modern API for HTTP client operations, replacing the old `WebClient` and `WebRequest`/`WebResponse` types (which have since been marked as obsolete).

`HttpClient` was written in response to the growth of HTTP-based web APIs and REST services, and provides a good experience when dealing with protocols more elaborate than simply fetching a web page. In particular:

- A single `HttpClient` instance can handle concurrent requests and plays well with features such as custom headers, cookies, and authentication schemes.
- `HttpClient` lets you write and plug in custom message handlers. This enables mocking in unit tests, and the creation of custom pipelines (for logging, compression, encryption, and so on).
- `HttpClient` has a rich and extensible type system for headers and content.

> **NOTE**
>
> `HttpClient` does not support progress reporting. For a solution, see "HttpClient with Progress.linq" at *http://www.albahari.com/nutshell/code.aspx* or via LINQPad's interactive samples gallery.

The simplest way to use `HttpClient` is to instantiate it and then call one of its `Get*` methods, passing in a URI:

```
string html = await new HttpClient().GetStringAsync ("http://linqpad.net");
```

(There's also `GetByteArrayAsync` and `GetStreamAsync`.) All I/O-bound methods in `HttpClient` are asynchronous.

Unlike its `WebRequest`/`WebResponse` predecessors, to get the best performance with `HttpClient`, you *must* reuse the same instance (otherwise things such as DNS resolution can be unnecessarily repeated and sockets are held open longer than necessary). `HttpClient` permits concurrent operations, so the following is legal and downloads two web pages at once:

```
var client = new HttpClient();
var task1 = client.GetStringAsync ("http://www.linqpad.net");
var task2 = client.GetStringAsync ("http://www.albahari.com");
```

```
Console.WriteLine (await task1);
Console.WriteLine (await task2);
```

`HttpClient` has a `Timeout` property and a `BaseAddress` property, which prefixes a URI to every request. `HttpClient` is somewhat of a thin shell: most of the other properties that you might expect to find here are defined in another class called `HttpClientHandler`. To access this class, you instantiate it and then pass the instance into `HttpClient`'s constructor:

```
var handler = new HttpClientHandler { UseProxy = false };
var client = new HttpClient (handler);
...
```

In this example, we told the handler to disable proxy support, which can sometimes improve performance by avoiding the cost of automatic proxy detection. There are also properties to control cookies, automatic redirection, authentication, and so on (we describe these in the following sections).

GetAsync and Response Messages

The `GetStringAsync`, `GetByteArrayAsync`, and `GetStreamAsync` methods are convenient shortcuts for calling the more general `GetAsync` method, which returns a *response message*:

```
var client = new HttpClient();
// The GetAsync method also accepts a CancellationToken.
HttpResponseMessage response = await client.GetAsync ("http://...");
response.EnsureSuccessStatusCode();
string html = await response.Content.ReadAsStringAsync();
```

`HttpResponseMessage` exposes properties for accessing the headers (see "Headers") and the HTTP `StatusCode`. An unsuccessful status code such as 404 (not found) doesn't cause an exception to be thrown unless you explicitly call `EnsureSuccessStatusCode`. Communication or DNS errors, however, do throw exceptions.

`HttpContent` has a `CopyToAsync` method for writing to another stream, which is useful in writing the output to a file:

```
using (var fileStream = File.Create ("linqpad.html"))
  await response.Content.CopyToAsync (fileStream);
```

`GetAsync` is one of four methods corresponding to HTTP's four verbs (the others are `PostAsync`, `PutAsync`, and `DeleteAsync`). We demonstrate `PostAsync` later in "Uploading Form Data".

SendAsync and Request Messages

`GetAsync`, `PostAsync`, `PutAsync`, and `DeleteAsync` are all shortcuts for calling `SendAsync`, the single low-level method into which everything else feeds. To use this, you first construct an `HttpRequestMessage`:

```
var client = new HttpClient();
var request = new HttpRequestMessage (HttpMethod.Get, "http://...");
HttpResponseMessage response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
...
```

Instantiating a `HttpRequestMessage` object means that you can customize properties of the request, such as the headers (see "Headers") and the content itself, allowing you to upload data.

Uploading Data and HttpContent

After instantiating a `HttpRequestMessage` object, you can upload content by assigning its `Content` property. The type for this property is an abstract class called `HttpContent`. .NET includes the following concrete subclasses for different kinds of content (you can also write your own):

- `ByteArrayContent`
- `StringContent`

- `FormUrlEncodedContent` (see "Uploading Form Data")
- `StreamContent`

For example:

```
var client = new HttpClient (new HttpClientHandler { UseProxy = false });
var request = new HttpRequestMessage (
  HttpMethod.Post, "http://www.albahari.com/EchoPost.aspx");
request.Content = new StringContent ("This is a test");
HttpResponseMessage response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```

HttpMessageHandler

We said previously that most of the properties for customizing requests are defined not in `HttpClient` but in `HttpClientHandler`. The latter is actually a subclass of the abstract `HttpMessageHandler` class, defined as follows:

```
public abstract class HttpMessageHandler : IDisposable
{
  protected internal abstract Task<HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken);

  public void Dispose();
  protected virtual void Dispose (bool disposing);
}
```

The `SendAsync` method is called from `HttpClient`'s `SendAsync` method.

`HttpMessageHandler` is simple enough to subclass easily and offers an extensibility point into `HttpClient`.

Unit testing and mocking

We can subclass `HttpMessageHandler` to create a *mocking* handler to assist with unit testing:

```
class MockHandler : HttpMessageHandler
{
  Func <HttpRequestMessage, HttpResponseMessage> _responseGenerator;

  public MockHandler
    (Func <HttpRequestMessage, HttpResponseMessage> responseGenerator)
  {
    _responseGenerator = responseGenerator;
  }

  protected override Task <HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken)
  {
    cancellationToken.ThrowIfCancellationRequested();
    var response = _responseGenerator (request);
    response.RequestMessage = request;
    return Task.FromResult (response);
  }
}
```

Its constructor accepts a function that tells the mocker how to generate a response from a request. This is the most versatile approach because the same handler can test multiple requests.

`SendAsync` is synchronous by virtue of `Task.FromResult`. We could have maintained asynchrony by having our response generator return a `Task<HttpResponseMessage>`, but this is pointless given that we can expect a mocking function to be short running. Here's how to use our mocking handler:

```
var mocker = new MockHandler (request =>
  new HttpResponseMessage (HttpStatusCode.OK)
  {
    Content = new StringContent ("You asked for " + request.RequestUri)
  });

var client = new HttpClient (mocker);
var response = await client.GetAsync ("http://www.linqpad.net");
string result = await response.Content.ReadAsStringAsync();
Assert.AreEqual ("You asked for http://www.linqpad.net/", result);
```

(`Assert.AreEqual` is a method you'd expect to find in a unit-testing framework such as NUnit.)

Chaining handlers with DelegatingHandler

You can create a message handler that calls another (resulting in a chain of handlers) by subclassing `DelegatingHandler`. You can use this to implement custom authentication, compression, and encryption protocols. The following demonstrates a simple logging handler:

```
class LoggingHandler : DelegatingHandler
{
  public LoggingHandler (HttpMessageHandler nextHandler)
  {
     InnerHandler = nextHandler;
  }

  protected async override Task <HttpResponseMessage> SendAsync
    (HttpRequestMessage request, CancellationToken cancellationToken)
  {
    Console.WriteLine ("Requesting: " + request.RequestUri);
    var response = await base.SendAsync (request, cancellationToken);
    Console.WriteLine ("Got response: " + response.StatusCode);
    return response;
  }
}
```

Notice that we've maintained asynchrony in overriding `SendAsync`. Introducing the `async` modifier when overriding a task-returning method is perfectly legal—and desirable in this case.

A better solution than writing to the `Console` would be to have the constructor accept some kind of logging object. Better still would be to accept a couple of `Action<T>` delegates that tell it how to log the request and response objects.

Proxies

A *proxy server* is an intermediary through which HTTP requests can be routed. Organizations sometimes set up a proxy server as the only means by which employees can access the internet—primarily because it simplifies security. A proxy has an address of its own and can demand authentication so that only selected users on the LAN can access the internet.

To use a proxy with `HttpClient`, first create an `HttpClientHandler` and assign its `Proxy` property and then feed that into `HttpClient`'s constructor:

```
WebProxy p = new WebProxy ("192.178.10.49", 808);
p.Credentials = new NetworkCredential ("username", "password", "domain");

var handler = new HttpClientHandler { Proxy = p };
var client = new HttpClient (handler);
...
```

`HttpClientHandler` also has a `UseProxy` property that you can assign to false instead of nulling out the `Proxy` property to defeat autodetection.

If you supply a domain when constructing the `NetworkCredential`, Windows-based authentication protocols are used. To use the currently authenticated Windows user, assign the static `CredentialCache.DefaultNetworkCredentials` value to the proxy's `Credentials` property.

As an alternative to repeatedly setting the `Proxy`, you can set the global default as follows:

```
HttpClient.DefaultWebProxy = myWebProxy;
```

Authentication

You can supply a username and password to an `HttpClient` as follows:

```
string username = "myuser";
string password = "mypassword";

var handler = new HttpClientHandler();
handler.Credentials = new NetworkCredential (username, password);
var client = new HttpClient (handler);
...
```

This works with dialog-based authentication protocols, such as Basic and Digest, and is extensible through the `AuthenticationManager` class. It also supports Windows NTLM and Kerberos (if you include a domain name

when constructing the `NetworkCredential` object). If you want to use the currently authenticated Windows user, you can leave the `Credentials` property `null` and instead set `UseDefaultCredentials` to `true`.

When you provide credentials, `HttpClient` automatically negotiates a compatible protocol. In some cases, there can be a choice: if you examine the initial response from a Microsoft Exchange server web mail page, for instance, it might contain the following headers:

```
HTTP/1.1 401 Unauthorized
Content-Length: 83
Content-Type: text/html
Server: Microsoft-IIS/6.0
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
WWW-Authenticate: Basic realm="exchange.somedomain.com"
X-Powered-By: ASP.NET
Date: Sat, 05 Aug 2006 12:37:23 GMT
```

The 401 code signals that authorization is required; the "WWW-Authenticate" headers indicate what authentication protocols are understood. If you configure the `HttpClientHandler` with the correct username and password, however, this message will be hidden from you because the runtime responds automatically by choosing a compatible authentication protocol, and then resubmitting the original request with an extra header. Here's an example:

```
Authorization: Negotiate TlRMTVNTUAAABAAAt5II2gjACDArAAACAwACACgAAAAQ
ATmKAAAAD0lVDRdPUksHUq9VUA==
```

This mechanism provides transparency, but generates an extra round trip with each request. You can avoid the extra round trips on subsequent requests to the same URI by setting the `PreAuthenticate` property on the `HttpClientHandler` to `true`.

CredentialCache

You can force a particular authentication protocol with a `CredentialCache` object. A credential cache contains one or more `NetworkCredential` objects, each keyed to a particular protocol and URI prefix. For example, you might want to avoid the Basic protocol when logging into an Exchange Server because it transmits passwords in plain text:

```
CredentialCache cache = new CredentialCache();
Uri prefix = new Uri ("http://exchange.somedomain.com");
cache.Add (prefix, "Digest",  new NetworkCredential ("joe", "passwd"));
cache.Add (prefix, "Negotiate", new NetworkCredential ("joe", "passwd"));

var handler = new HttpClientHandler();
handler.Credentials = cache;
...
```

An authentication protocol is specified as a string. The valid values include:

```
Basic, Digest, NTLM, Kerberos, Negotiate
```

In this particular situation it will choose `Negotiate`, because the server didn't indicate that it supported `Digest` in its authentication headers. `Negotiate` is a Windows protocol that currently boils down to either Kerberos or NTLM, depending on the capabilities of the server, but ensures forward compatibility of your application when future security standards are deployed.

The static `CredentialCache.DefaultNetworkCredentials` property allows you to add the currently authenticated Windows user to the credential cache without having to specify a password:

```
cache.Add (prefix, "Negotiate", CredentialCache.DefaultNetworkCredentials);
```

Authenticating via headers

Another way to authenticate is to set the authentication header directly:

```
var client = new HttpClient();
client.DefaultRequestHeaders.Authorization =
```

```
    new AuthenticationHeaderValue ("Basic",
      Convert.ToBase64String (Encoding.UTF8.GetBytes ("username:password")));
...
```

This strategy also works with custom authentication systems such as OAuth.

Headers

`HttpClient` lets you add custom HTTP headers to a request, as well as enumerate the headers in a response. A header is simply a key/value pair containing metadata, such as the message content type or server software. `HttpClient` exposes strongly typed collections with properties for standard HTTP headers. The `DefaultRequestHeaders` property is for headers that apply to every request:

```
var client = new HttpClient (handler);

client.DefaultRequestHeaders.UserAgent.Add (
  new ProductInfoHeaderValue ("VisualStudio", "2022"));

client.DefaultRequestHeaders.Add ("CustomHeader", "VisualStudio/2022");
```

The `Headers` property on the `HttpRequestMessage` class, however, is for headers specific to a request.

Query Strings

A query string is simply a string appended to a URI with a question mark, used to send simple data to the server. You can specify multiple key/value pairs in a query string with the following syntax:

```
?key1=value1&key2=value2&key3=value3...
```

Here's a URI with a query string:

```
string requestURI = "http://www.google.com/search?q=HttpClient&hl=fr";
```

If there's a possibility of your query including symbols or spaces, you can use `Uri`'s `EscapeDataString` method to create a legal URI:

```
string search = Uri.EscapeDataString ("(HttpClient or HttpRequestMessage)");
string language = Uri.EscapeDataString ("fr");
string requestURI = "http://www.google.com/search?q=" + search +
                    "&hl=" + language;
```

This resultant URI is:

```
http://www.google.com/search?q=(HttpClient%20OR%20HttpRequestMessage)&hl=fr
```

(`EscapeDataString` is similar to `EscapeUriString` except that it also encodes characters such as `&` and `=`, which would otherwise mess up the query string.)

Uploading Form Data

To upload HTML form data, create and populate the `FormUrlEncodedContent` object. You can then either pass it into the `PostAsync` method or assign it to a request's `Content` property:

```
string uri = "http://www.albahari.com/EchoPost.aspx";
var client = new HttpClient();
var dict = new Dictionary<string,string>
{
    { "Name", "Joe Albahari" },
    { "Company", "O'Reilly" }
};
var values = new FormUrlEncodedContent (dict);
var response = await client.PostAsync (uri, values);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```

Cookies

A cookie is a name/value string pair that an HTTP server sends to a client in a response header. A web browser client typically remembers cookies and replays them to the server in each subsequent request (to the same address)

until their expiry. A cookie allows a server to know whether it's talking to the same client it was a minute ago—or yesterday—without needing a messy query string in the URI.

By default, `HttpClient` ignores any cookies received from the server. To accept cookies, create a `CookieContainer` object and assign it an `HttpClientHandler`:

```
var cc = new CookieContainer();
var handler = new HttpClientHandler();
handler.CookieContainer = cc;
var client = new HttpClient (handler);
...
```

To replay the received cookies in future requests, simply use the same `CookieContainer` object again. Alternatively, you can start with a fresh `CookieContainer` and then add cookies manually, as follows:

```
Cookie c = new Cookie ("PREF",
                       "ID=6b10df1da493a9c4:TM=1179...",
                       "/",
                       ".google.com");
freshCookieContainer.Add (c);
```

The third and fourth arguments indicate the path and domain of the originator. A `CookieContainer` on the client can house cookies from many different places; `HttpClient` sends only those cookies whose path and domain match those of the server.

Writing an HTTP Server

NOTE

If you need to write an HTTP server, an alternative higher-level approach (from .NET 6) is to use the ASP.NET minimal API. Here's all it takes to get started:

```
var app = WebApplication.CreateBuilder().Build();
app.MapGet ("/", () => "Hello, world!");
app.Run();
```

You can write your own .NET HTTP server with the `HttpListener` class. The following is a simple server that listens on port 51111, waits for a single client request, and then returns a one-line reply:

```
using var server = new SimpleHttpServer();

// Make a client request:
Console.WriteLine (await new HttpClient().GetStringAsync
  ("http://localhost:51111/MyApp/Request.txt"));

class SimpleHttpServer : IDisposable
{
  readonly HttpListener listener = new HttpListener();

  public SimpleHttpServer() => ListenAsync();
  async void ListenAsync()
  {
    listener.Prefixes.Add ("http://localhost:51111/MyApp/");  // Listen on
    listener.Start();                                         // port 51111

    // Await a client request:
    HttpListenerContext context = await listener.GetContextAsync();

    // Respond to the request:
    string msg = "You asked for: " + context.Request.RawUrl;
    context.Response.ContentLength64 = Encoding.UTF8.GetByteCount (msg);
    context.Response.StatusCode = (int)HttpStatusCode.OK;

    using (Stream s = context.Response.OutputStream)
    using (StreamWriter writer = new StreamWriter (s))
      await writer.WriteAsync (msg);
  }
```

```
  public void Dispose() => listener.Close();
}

OUTPUT: You asked for: /MyApp/Request.txt
```

On Windows, `HttpListener` does not internally use .NET `Socket` objects; it instead calls the Windows HTTP Server API. This allows many applications on a computer to listen on the same IP address and port—as long as each registers different address prefixes. In our example, we registered the prefix *http://localhost/myapp*, so another application would be free to listen on the same IP and port on another prefix such as *http://localhost/anotherapp*. This is of value because opening new ports on corporate firewalls can be politically arduous.

`HttpListener` waits for the next client request when you call `GetContext`, returning an object with `Request` and `Response` properties. Each is analogous to client request or response, but from the server's perspective. You can read and write headers and cookies, for instance, to the request and response objects, much as you would at the client end.

You can choose how fully to support features of the HTTP protocol, based on your anticipated client audience. At a bare minimum, you should set the content length and status code on each request.

Here's a very simple web page server, written *asynchronously*:

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading.Tasks;

class WebServer
{
  HttpListener _listener;
  string _baseFolder;      // Your web page folder.

  public WebServer (string uriPrefix, string baseFolder)
  {
    _listener = new HttpListener();
```

```
    _listener.Prefixes.Add (uriPrefix);
    _baseFolder = baseFolder;
  }

  public async void Start()
  {
    _listener.Start();
    while (true)
      try
      {
        var context = await _listener.GetContextAsync();
        Task.Run (() => ProcessRequestAsync (context));
      }
      catch (HttpListenerException)     { break; }   // Listener stopped.
      catch (InvalidOperationException) { break; }   // Listener stopped.
  }

  public void Stop() => _listener.Stop();

  async void ProcessRequestAsync (HttpListenerContext context)
  {
    try
    {
      string filename = Path.GetFileName (context.Request.RawUrl);
      string path = Path.Combine (_baseFolder, filename);
      byte[] msg;
      if (!File.Exists (path))
      {
        Console.WriteLine ("Resource not found: " + path);
        context.Response.StatusCode = (int) HttpStatusCode.NotFound;
        msg = Encoding.UTF8.GetBytes ("Sorry, that page does not exist");
      }
      else
      {
        context.Response.StatusCode = (int) HttpStatusCode.OK;
        msg = File.ReadAllBytes (path);
      }
      context.Response.ContentLength64 = msg.Length;
      using (Stream s = context.Response.OutputStream)
        await s.WriteAsync (msg, 0, msg.Length);
    }
    catch (Exception ex) { Console.WriteLine ("Request error: " + ex); }
  }
}
```

The following code sets things in motion:

```
// Listen on port 51111, serving files in d:\webroot:
var server = new WebServer ("http://localhost:51111/", @"d:\webroot");
try
{
  server.Start();
  Console.WriteLine ("Server running... press Enter to stop");
  Console.ReadLine();
}
finally { server.Stop(); }
```

You can test this at the client end with any web browser; the URI in this case will be *http://localhost:51111/* plus the name of the web page.

> **WARNING**
>
> `HttpListener` will not start if other software is competing for the same port (unless that software also uses the Windows HTTP Server API). Examples of applications that might listen on the default port 80 include a web server or a peer-to-peer program such as Skype.

Our use of asynchronous functions makes this server scalable and efficient. Starting this from a user interface (UI) thread, however, would hinder scalability because for each *request*, execution would bounce back to the UI thread after each `await`. Incurring such overhead is particularly pointless given that we don't have shared state, so in a UI scenario we'd get off the UI thread, either like this:

```
Task.Run (Start);
```

or by calling `ConfigureAwait(false)` after calling `GetContextAsync`.

Note that we used `Task.Run` to call `ProcessRequestAsync` even though the method was already asynchronous. This allows the caller to process another request *immediately* rather than having to first wait out the synchronous phase of the method (up until the first `await`).

Using DNS

The static `Dns` class encapsulates the DNS, which converts between a raw IP address, such as 66.135.192.87, and a human-friendly domain name, such as *ebay.com*.

The `GetHostAddresses` method converts from domain name to IP address (or addresses):

```
foreach (IPAddress a in Dns.GetHostAddresses ("albahari.com"))
  Console.WriteLine (a.ToString());     // 205.210.42.167
```

The `GetHostEntry` method goes the other way around, converting from address to domain name:

```
IPHostEntry entry = Dns.GetHostEntry ("205.210.42.167");
Console.WriteLine (entry.HostName);                   // albahari.com
```

`GetHostEntry` also accepts an `IPAddress` object, so you can specify an IP address as a byte array:

```
IPAddress address = new IPAddress (new byte[] { 205, 210, 42, 167 });
IPHostEntry entry = Dns.GetHostEntry (address);
Console.WriteLine (entry.HostName);                   // albahari.com
```

Domain names are automatically resolved to IP addresses when you use a class such as `WebRequest` or `TcpClient`. However, if you plan to make many network requests to the same address over the life of an application, you can sometimes improve performance by first using `Dns` to explicitly convert the domain name into an IP address, and then communicating directly with the IP address from that point on. This avoids repeated round-tripping to resolve the same domain name, and it can be of benefit when dealing at the transport layer (via `TcpClient`, `UdpClient`, or `Socket`).

The DNS class also provides awaitable task-based asynchronous methods:

```
foreach (IPAddress a in await Dns.GetHostAddressesAsync ("albahari.com"))
  Console.WriteLine (a.ToString());
```

Sending Mail with SmtpClient

The `SmtpClient` class in the `System.Net.Mail` namespace allows you to send mail messages through the ubiquitous Simple Mail Transfer Protocol, or SMTP. To send a simple text message, instantiate `SmtpClient`, set its `Host` property to your SMTP server address, and then call `Send`:

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myserver.com";
client.Send ("from@adomain.com", "to@adomain.com", "subject", "body");
```

Constructing a `MailMessage` object exposes further options, including the ability to add attachments:

```
SmtpClient client = new SmtpClient();
client.Host = "mail.myisp.net";
MailMessage mm = new MailMessage();

mm.Sender = new MailAddress ("kay@domain.com", "Kay");
mm.From   = new MailAddress ("kay@domain.com", "Kay");
mm.To.Add  (new MailAddress ("bob@domain.com", "Bob"));
mm.CC.Add  (new MailAddress ("dan@domain.com", "Dan"));
mm.Subject = "Hello!";
mm.Body = "Hi there. Here's the photo!";
mm.IsBodyHtml = false;
mm.Priority = MailPriority.High;

Attachment a = new Attachment ("photo.jpg",
                               System.Net.Mime.MediaTypeNames.Image.Jpeg);
mm.Attachments.Add (a);
client.Send (mm);
```

To frustrate spammers, most SMTP servers on the internet will accept connections only from authenticated connections and require communication over SSL:

```
var client = new SmtpClient ("smtp.myisp.com", 587)
{
  Credentials = new NetworkCredential ("me@myisp.com", "MySecurePass"),
  EnableSsl = true
};
```

```
client.Send ("me@myisp.com", "someone@somewhere.com", "Subject", "Body");
Console.WriteLine ("Sent");
```

By changing the `DeliveryMethod` property, you can instruct the `SmtpClient` to instead use IIS to send mail messages or simply to write each message to an *.eml* file in a specified directory. This can be useful during development:

```
SmtpClient client = new SmtpClient();
client.DeliveryMethod = SmtpDeliveryMethod.SpecifiedPickupDirectory;
client.PickupDirectoryLocation = @"c:\mail";
```

Using TCP

TCP and UDP constitute the transport layer protocols on top of which most internet—and LAN—services are built. HTTP (version 2 and below), FTP, and SMTP use TCP; DNS and HTTP version 3 use UDP. TCP is connection-oriented and includes reliability mechanisms; UDP is connectionless, has a lower overhead, and supports broadcasting. *BitTorrent* uses UDP, as does Voice over IP (VoIP).

The transport layer offers greater flexibility—and potentially improved performance—over the higher layers, but it requires that you handle such tasks as authentication and encryption yourself.

With TCP in .NET, you have a choice of either the easier-to-use `TcpClient` and `TcpListener` façade classes, or the feature-rich `Socket` class. (In fact, you can mix and match, because `TcpClient` exposes the underlying `Socket` object through the `Client` property.) The `Socket` class exposes more configuration options and allows direct access to the network layer (IP) and non-internet-based protocols such as Novell's SPX/IPX.

As with other protocols, TCP differentiates a client and server: the client initiates a request, while the server waits for a request. Here's the basic structure for a synchronous TCP client request:

```
using (TcpClient client = new TcpClient())
{
  client.Connect ("address", port);
  using (NetworkStream n = client.GetStream())
  {
    // Read and write to the network stream...
  }
}
```

`TcpClient`'s `Connect` method blocks until a connection is established (`ConnectAsync` is the asynchronous equivalent). The `NetworkStream` then provides a means of two-way communication, for both transmitting and receiving bytes of data from a server.

A simple TCP server looks like this:

```
TcpListener listener = new TcpListener (<ip address>, port);
listener.Start();

while (keepProcessingRequests)
  using (TcpClient c = listener.AcceptTcpClient())
  using (NetworkStream n = c.GetStream())
  {
    // Read and write to the network stream...
  }

listener.Stop();
```

`TcpListener` requires the local IP address on which to listen (a computer with two network cards, for instance, can have two addresses). You can use `IPAddress.Any` to instruct it to listen on all (or the only) local IP addresses. `AcceptTcpClient` blocks until a client request is received (again, there's also an asynchronous version), at which point we call `GetStream`, just as on the client side.

When working at the transport layer, you need to decide on a protocol for who talks when and for how long—rather like with a walkie-talkie. If both parties talk or listen at the same time, communication breaks down!

Let's invent a protocol in which the client speaks first, saying "Hello," and then the server responds by saying "Hello right back!" Here's the code:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

new Thread (Server).Start();       // Run server method concurrently.
Thread.Sleep (500);                // Give server time to start.
Client();

void Client()
{
  using (TcpClient client = new TcpClient ("localhost", 51111))
  using (NetworkStream n = client.GetStream())
  {
    BinaryWriter w = new BinaryWriter (n);
    w.Write ("Hello");
    w.Flush();
    Console.WriteLine (new BinaryReader (n).ReadString());
  }
}

void Server()     // Handles a single client request, then exits.
{
  TcpListener listener = new TcpListener (IPAddress.Any, 51111);
  listener.Start();
  using (TcpClient c = listener.AcceptTcpClient())
  using (NetworkStream n = c.GetStream())
  {
    string msg = new BinaryReader (n).ReadString();
    BinaryWriter w = new BinaryWriter (n);
    w.Write (msg + " right back!");
    w.Flush();                      // Must call Flush because we're not
  }                                 // disposing the writer.
  listener.Stop();
}

// OUTPUT: Hello right back!
```

In this example, we're using the `localhost` loopback to run the client and server on the same machine. We've arbitrarily chosen a port in the unallocated range (above 49152) and used a `BinaryWriter` and `BinaryReader` to encode the text messages. We've avoided closing or

disposing the readers and writers in order to keep the underlying `NetworkStream` open until our conversation completes.

`BinaryReader` and `BinaryWriter` might seem like odd choices for reading and writing strings. However, they have a major advantage over `StreamReader` and `StreamWriter`: they prefix strings with an integer indicating the length, so a `BinaryReader` always knows exactly how many bytes to read. If you call `StreamReader.ReadToEnd`, you might block indefinitely—because a `NetworkStream` doesn't have an end! As long as the connection is open, the network stream can never be sure that the client isn't going to send more data.

> **NOTE**
>
> `StreamReader` is in fact completely out of bounds with `NetworkStream`, even if you plan only to call `ReadLine`. This is because `StreamReader` has a read-ahead buffer, which can result in it reading more bytes than are currently available, blocking indefinitely (or until the socket times out). Other streams such as `FileStream` don't suffer this incompatibility with `StreamReader` because they have a definite *end*—at which point `Read` returns immediately with a value of `0`.

Concurrency with TCP

`TcpClient` and `TcpListener` offer task-based asynchronous methods for scalable concurrency. Using these is simply a question of replacing blocking method calls with their `*Async` versions and awaiting the task that's returned.

In the following example, we write an asynchronous TCP server that accepts requests of 5,000 bytes in length, reverses the bytes, and then sends them back to the client:

```
async void RunServerAsync ()
{
  var listener = new TcpListener (IPAddress.Any, 51111);
  listener.Start ();
  try
  {
```

```
    while (true)
      Accept (await listener.AcceptTcpClientAsync ());
  }
  finally { listener.Stop(); }
}

async Task Accept (TcpClient client)
{
  await Task.Yield ();
  try
  {
    using (client)
    using (NetworkStream n = client.GetStream ())
    {
      byte[] data = new byte [5000];

      int bytesRead = 0; int chunkSize = 1;
      while (bytesRead < data.Length && chunkSize > 0)
        bytesRead += chunkSize =
          await n.ReadAsync (data, bytesRead, data.Length - bytesRead);

      Array.Reverse (data);   // Reverse the byte sequence
      await n.WriteAsync (data, 0, data.Length);
    }
  }
  catch (Exception ex) { Console.WriteLine (ex.Message); }
}
```

Such a program is scalable in that it does not block a thread for the duration of a request. So, if 1,000 clients were to connect at once over a slow network connection (so that each request took several seconds from start to finish, for example), this program would not require 1,000 threads for that time (unlike with a synchronous solution). Instead, it leases threads only for the small periods of time required to execute code before and after the `await` expressions.

Receiving POP3 Mail with TCP

.NET provides no application-layer support for POP3, so you need to write at the TCP layer in order to receive mail from a POP3 server. Fortunately, this is a simple protocol; a POP3 conversation goes like this:

Client	Mail server	Notes
Client connects...	`+OK Hello there.`	Welcome message
`USER joe`	`+OK Password requ ired.`	
`PASS password`	`+OK Logged in.`	

`LIST`	`+OK`
	`1 1876`
	`2 5412`
	`3 845`
	`.`
		Lists the ID and file size of each message on the server

`RETR 1`	`+OK 1876 octets`
	Content of message #1...
	`.`
		Retrieves the message with the specified ID

`DELE 1`	`+OK Deleted.`	Deletes a message from the server
`QUIT`	`+OK Bye-bye.`	

Each command and response is terminated by a new line (CR + LF) except for the multiline `LIST` and `RETR` commands, which are terminated by a single dot on a separate line. Because we can't use `StreamReader` with `NetworkStream`, we can start by writing a helper method to read a line of text in a nonbuffered fashion:

```
string ReadLine (Stream s)
{
  List<byte> lineBuffer = new List<byte>();
  while (true)
  {
```

```
    int b = s.ReadByte();
    if (b == 10 || b < 0) break;
    if (b != 13) lineBuffer.Add ((byte)b);
  }
  return Encoding.UTF8.GetString (lineBuffer.ToArray());
}
```

We also need a helper method to send a command. Because we always expect to receive a response starting with `+OK`, we can read and validate the response at the same time:

```
void SendCommand (Stream stream, string line)
{
  byte[] data = Encoding.UTF8.GetBytes (line + "\r\n");
  stream.Write (data, 0, data.Length);
  string response = ReadLine (stream);
  if (!response.StartsWith ("+OK"))
    throw new Exception ("POP Error: " + response);
}
```

With these methods written, the job of retrieving mail is easy. We establish a TCP connection on port 110 (the default POP3 port) and then start talking to the server. In this example, we write each mail message to a randomly named file with an *.eml* extension, before deleting the message off the server:

```
using (TcpClient client = new TcpClient ("mail.isp.com", 110))
using (NetworkStream n = client.GetStream())
{
  ReadLine (n);                               // Read the welcome message.
  SendCommand (n, "USER username");
  SendCommand (n, "PASS password");
  SendCommand (n, "LIST");                    // Retrieve message IDs
  List<int> messageIDs = new List<int>();
  while (true)
  {
    string line = ReadLine (n);               // e.g.,  "1 1876"
    if (line == ".") break;
    messageIDs.Add (int.Parse (line.Split (' ')[0] ));   // Message ID
  }

  foreach (int id in messageIDs)         // Retrieve each message.
  {
```

```
    SendCommand (n, "RETR " + id);
    string randomFile = Guid.NewGuid().ToString() + ".eml";
    using (StreamWriter writer = File.CreateText (randomFile))
      while (true)
      {
        string line = ReadLine (n);      // Read next line of message.
        if (line == ".") break;          // Single dot = end of message.
        if (line == "..") line = ".";    // "Escape out" double dot.
        writer.WriteLine (line);         // Write to output file.
      }
    SendCommand (n, "DELE " + id);       // Delete message off server.
  }
  SendCommand (n, "QUIT");
}
```

NOTE

You can find open source POP3 libraries on NuGet that provide support for protocol aspects such as authentication TLS/SSL connections, MIME parsing, and more.