

**An Introduction To Internet Programming
With SocketWrench .NET**

Introduction

The Transmission Control Protocol/Internet Protocol (TCP/IP) is the common language spoken by every device and application that communicates over the Internet. As a developer who is interested in integrating Internet functionality into your own software, it is important to understand the fundamentals of how TCP/IP works. This tutorial will explain the basic concepts behind network programming, and help get you started developing your first Internet application.

The examples included in this tutorial are geared primarily towards Visual Basic developers; however, the general concepts can be applied to any .NET programming language. To work with these examples, you should have Visual Studio 2005 and SocketWrench .NET installed. SocketWrench is the general purpose TCP/IP networking component that we use throughout our examples.

SocketWrench is also part of a larger collection of Internet components in our SocketTools family of products. With SocketTools, you can easily take advantage of features such as uploading and downloading files, sending e-mail messages, managing your Inbox on your mail server, submitting data to a web server, executing remote commands and much more. While SocketWrench makes it easy to create your own custom client and server applications, SocketTools further simplifies application development by enabling you to incorporate high-level Internet functionality in your software with just a few lines of code. More information and a free evaluation copy of both SocketWrench and SocketTools is available on the Catalyst Development website at www.catalyst.com

Windows Sockets API

The Windows Sockets specification was created by a group of companies, including Microsoft, in an effort to standardize the TCP/IP suite of protocols under Windows. Prior to Windows Sockets, each vendor developed their own proprietary libraries, and although they all had similar functionality, the differences were significant enough to cause problems for the software developers who used them. After choosing to use a specific vendor's library, the developer was locked into that particular implementation because a program written against one vendor's product would not work with another's. Windows Sockets was offered as a solution, leaving developers and their end-users free to choose any vendor's TCP/IP implementation with the assurance that the product would continue to work. Today, there are a few third-party TCP/IP products available for the Windows platform; however, most systems use the Microsoft TCP/IP libraries that are included as part of the base operating system and it has become the standard networking interface on the Windows platform.

SocketWrench is a component that uses the Windows Sockets API to simplify the development of Internet applications. It offers a higher level interface, enabling the developer to set properties, invoke methods and create event handlers to respond to network events. In programming languages like Visual Basic, it provides a more natural programming interface that avoids much of the error-prone drudgery commonly associated with sockets programming. By simply referencing SocketWrench in your project, setting some properties and responding to events, you can quickly and easily write an Internet-enabled application. However, before we get started with SocketWrench, we'll cover the basic terminology and concepts behind sockets programming in general.

Protocol Standards

In the context of Internet programming, a protocol establishes the "rules of the road" that each computer must follow so that all of the systems in the network can understand the data being exchanged between them. There are two general levels of protocols that are typically discussed. The first consists of the networking protocols that govern how two or more computer systems communicate with one another. Two examples of these lower level networking protocols would be TCP and UDP, which are discussed later in this tutorial.

The second type of protocol is one which determines how applications exchange information and perform certain tasks. An example of these higher level protocols would be HTTP and FTP, which are used to communicate with web and file servers. These higher level application protocols then use the lower level networking protocols such as TCP to communicate over the Internet.

Most Internet protocols, including the protocols that we discuss in this tutorial, are described in technical documents called Request for Comments (RFCs). These documents are published through the Internet Society, an international organization of computer scientists and engineers that promote Internet standards. Eventually some of these proposed Internet protocols are adopted by the Internet Engineering Task Force (IETF), which formalizes them as standard protocols.

A number is used to reference the standards document for each protocol. For example, the standards document for the File Transfer Protocol is number 959, and is commonly referred to as RFC 959. A list of the available RFC documents is available at the IETF website:

<http://www.ietf.org/>

It's important to keep in mind that RFCs are technical documents that describe the implementation details of a particular protocol, and are not meant to provide an overview of the protocol in layman's terms or explain how to use a particular protocol. For example, RFC 959 describes how FTP is implemented, but it will not explain how to use an application to download a file. The language used in RFCs also tends to be platform-neutral. In other words, you won't typically find specific implementation information for the Windows operating system. Those kinds of platform specific details are left to the programmer, and no assumptions are made about the underlying hardware or operating system.

The Transmission Control Protocol

When two computers wish to exchange information over a network, there are several components that must be in place before the data can actually be sent and received. Of course, the physical hardware must exist, which is typically either a network interface card (NIC) or a serial communications port for dial-up networking connections. In addition to the physical connection, computers also need to use a protocol that defines the parameters of the communication between them. One of the most popular protocols in use today is TCP/IP, which stands for Transmission Control Protocol/Internet Protocol.

By convention, TCP/IP is used to refer to a suite of protocols, all based on the Internet Protocol (IP). Unlike a single local network, where every system is directly connected to each other, an internet is a

collection of networks, combined into a single, virtual network. The Internet Protocol provides the means by which any system on any network can communicate with another as easily as if they were on the same physical network. Each system, commonly referred to as a host, is assigned a unique number that can be used to identify it over the network. The most common version of the Internet Protocol used today is IPv4 (version 4) which uses a 32-bit value for the address. Typically, this address is broken into four 8-bit numbers separated by periods. This is called dot-notation, and looks something like "192.43.19.64". Some parts of the address are used to identify the network that the system is connected to, and the remainder identifies the system itself. There are three classes of Internet addresses, generally referred to as class "A", "B" and "C". The rule of thumb is that class A addresses are assigned to very large networks, class "B" addresses are assigned to medium sized networks, and class "C" addresses are assigned to smaller networks with less than approximately 250 systems. It should be noted that the latest version of the Internet Protocol is IPv6 (version 6), which supports 128-bit addresses. However, at this point IPv4 is still the most widely used version of the protocol and the use of IPv6 remains largely limited to research and governmental networks. It is generally predicted that the widespread adoption of IPv6 will be phased in over the next five to ten years.

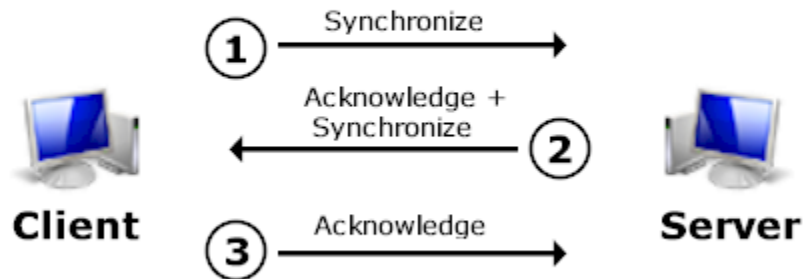
When a system sends data over the network using the Internet Protocol, it is sent in discrete units called datagrams, also commonly referred to as packets. A datagram consists of a header followed by application-defined data. The header contains the addressing information that is used to deliver the datagram to its destination, much like an envelope is used to address and contain postal mail. And like postal mail, there is no guarantee that a datagram will actually arrive at its destination. In fact, datagrams may be lost, duplicated or delivered out of order during their travels over the network.

Needless to say, this kind of unreliability can cause a lot of problems for software developers. What's really needed is a reliable, straightforward way to exchange data without having to worry about lost packets or jumbled data.

To fill this need, the Transmission Control Protocol (TCP) was developed. Built on top of IP, TCP offers a reliable, full-duplex byte stream that may be read and written to in a fashion similar to reading and writing a file. The advantages to this are obvious: the application programmer doesn't need to write code to handle dropped or out-of-order datagrams, and instead can focus on the application itself. And because the data is presented as a stream of bytes, existing code can be easily adopted and modified to use TCP.

Before two programs can begin to exchange data using TCP, they must establish a "connection" with each other. This is done with a three-way handshake in which both sides exchange packets, establishing the initial packet sequence numbers. The client is responsible for initiating the connection, while the server's responsibility is to wait, listen and respond to incoming connections.

TCP Three-Way Handshake



The first step is for the client to send a synchronization (SYN) packet to the server that contains its sequence number. Next, the server responds with an acknowledgement and synchronization (SYN-ACK) packet that contains its sequence number in response. Finally, the client sends an acknowledgement (ACK) back to the server. Once the connection has been established, both sides may send and receive data until the connection is closed. This series of steps is important because it ensures that the bytes of data exchanged between the client and server will be received in the same order they were sent.

TCP is known as a stream-oriented protocol because data is exchanged between the client and server as a stream of bytes. While TCP will guarantee that the data will arrive intact, with the bytes received in the same order that they were written, there is no guarantee that the amount of data received in a single read operation on the socket will match the amount of data written by the remote host.

For example, consider a server that sends data to a client in four separate operations, each containing 1024 bytes of data. While it is convenient to think of these as discrete blocks of data, TCP considers it a stream of 4096 bytes. The client may receive that data in a single read on the socket, returning all 4096 bytes. Alternatively, it may read the socket, and only receive the first 1460 bytes; subsequent reads may return another 1460 bytes, followed by the remaining 1176 bytes. Applications which make assumptions about the amount of data they can read or write in a single operation may work in some environments, such as on a local network, but fail on slower connections.

A general rule to use when designing an application using TCP is to consider how the program would handle the situation where the read operation only returns a single byte. If the application can correctly handle this kind of extreme case, then it should function correctly even under adverse network conditions.

In some situations it may be desirable to design the application to exchange information as discrete messages or blocks of data. While this isn't directly supported by TCP, it can be implemented on top of the data stream. There are several methods that can be used to accomplish this, depending on the requirements of the application:

1. Exchange the data as fixed length structures. This is the simplest approach, and has very little or no overhead. The client and server can either use predefined values, or negotiate the size of the data structures when the connection is established.
2. Prefix variable-length data structures with the number of bytes being sent. The length value could be expressed either as a native integer value, or as a fixed-length string that is converted to a numeric value by the application. This allows the receiver to read this fixed length value, and then use that value to determine how many additional bytes must be read to obtain the complete message or data structure.
3. Prefix the data with a unique byte or byte sequence that would normally not be found in the data stream. This would be followed by the data itself, with a complete message received when another unique byte sequence is encountered. Alternatively, a unique byte sequence could be used to terminate a message. This is the approach that many Internet application protocols use, such as FTP, SMTP and POP3. Commands are sent as one or more printable characters, terminated with a carriage-return (CR) and linefeed (LF) byte sequence that tells the remote host that a complete command has been received.
4. A combination of the above methods, using unique byte sequences. The message length and even additional information such as a CRC-32 checksum or MD5 hash can be used to validate the integrity of the data. This would effectively create an "envelope" around the data being exchanged, and additional checks could be made to ensure that the data has been received and processed correctly.

Regardless of the method used, for best performance it is recommended that the application buffer the data received and then process the data out of that buffer. When using asynchronous (non-blocking) sockets, the application should read all of the data available on the socket, typically in a loop which adds the data to the buffer and exiting the loop when there is no more data available at that time.

The User Datagram Protocol

Unlike TCP, the User Datagram Protocol (UDP) does not present data as a stream of bytes, nor does it require that you establish a connection with another program in order to exchange information. Data is exchanged in discrete units called datagrams, which are similar to IP datagrams. In fact, the only features that UDP offers over raw IP datagrams are port numbers and an optional checksum.

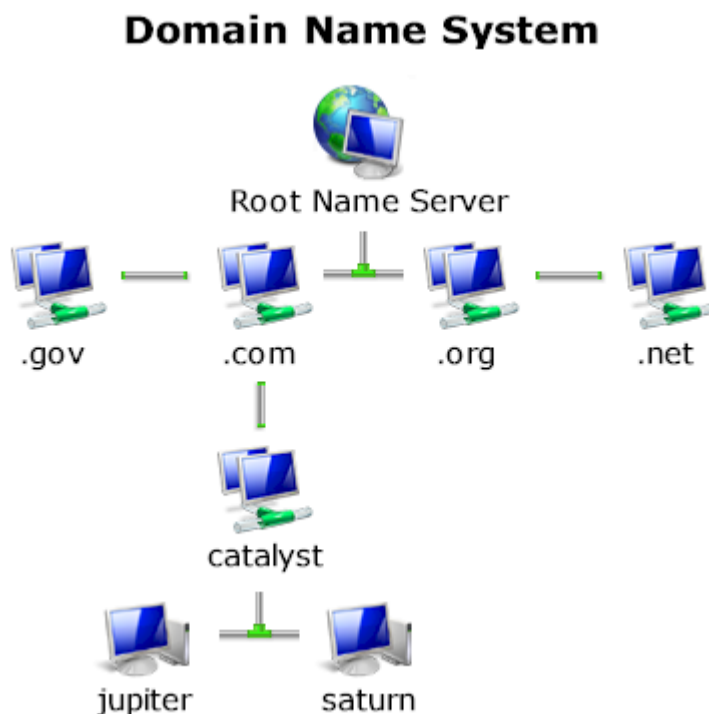
UDP is sometimes referred to as an unreliable protocol because when a program sends a UDP datagram over the network, there is no way for it to know that it actually arrived at its destination. This means that the sender and receiver must typically implement their own acknowledgement protocol on top of UDP. Much of the work that TCP does transparently (such as generating checksums, acknowledging the receipt of packets, retransmitting lost packets and so on) must be performed by the application itself.

With the limitations of UDP, you might wonder why it's used at all. UDP has the advantage over TCP in two critical areas: speed and packet overhead. Because TCP is a reliable protocol, it goes through great

lengths to insure data arrives at its destination intact, and as a result, it exchanges a fairly high number of packets over the network. UDP doesn't have this overhead and is considerably faster than TCP. In situations where speed is paramount, or the number of packets sent over the network must be kept to a minimum, UDP is often the preferred solution. A common use of UDP is with video and audio streaming, where the downside of potential packet loss is outweighed by the benefits of a faster transfer rate and reduced bandwidth.

Hostnames

Before an application can exchange data with another computer system, it must have several essential pieces of information. The first is the Internet Protocol address of the remote host. This address is typically expressed either as its numerical value in dot-notation, or by a logical name called a hostname. Like an address in dot-notation, hostnames are divided into several pieces separated by periods, called domains. Domains have a hierarchical structure, where the top-level domains define the type of organization to which that network belongs, and sub-domains further identify the specific network.



In this figure, the top-level domains are "gov" (government agencies), "com" (commercial organizations), "org" (organizations) and "net" (service providers). The fully qualified domain name is specified by naming the host and each parent sub-domain above it, separating them with periods. For example, the fully qualified domain name for the "jupiter" host would be "jupiter.catalyst.com". The system "jupiter" is part of the "catalyst" domain (a company's local network), which in turn is part of the "com" top-level domain (a domain used by all commercial enterprises).

In order to use a hostname instead of a dot-address to identify a specific system or network, there must be some correlation between the two. This is accomplished by one of two means: a local host table or a domain name server. A host table is a text file that lists the IP address of a host, followed by the names that it's known by. Typically this file is named **hosts** and is found in the C:\Windows\System32\Drivers\Etc folder.

A domain name server, on the other hand, is a system that can be presented with a hostname and will return that host's IP address. This approach is advantageous because the host information for the entire network is maintained in one centralized location, rather than being scattered about on every host on the network.

Service Ports

In addition to the IP address of the remote system, an application also needs to know how to address the specific program with which it wishes to communicate. This is accomplished by specifying a service port, a 16-bit number that uniquely identifies an application running on the system. Instead of numbers, however, service names are usually used instead. Like hostnames, service names are usually matched to port numbers through a local file, commonly called **services**. This file lists the logical service name, followed by the port number and protocol used by the server. Like the **hosts** file, this file is also found in the C:\Windows\System32\Drivers\Etc folder.

A number of standard service names are used by Internet-based applications and these are referred to as well-known services. These services are defined by a standards document and include common application protocols such as FTP, POP3, SMTP and HTTP.

Remember that a service name or port number is a way to address an application running on a remote host. Because a particular service name is used, it doesn't guarantee that the service is available, just as dialing a telephone number doesn't guarantee that there is someone at home to answer the call.

Sockets

The previous sections described what information a program needs to communicate over a TCP/IP network. The next step is for the program to create what is called a socket, a communications end-point that can be likened to a telephone. However, creating a socket by itself doesn't let you exchange information, just like having a telephone in your house doesn't mean that you can talk to someone by simply taking it off the hook. You need to establish a connection with the other program, just as you need to dial a telephone number, and to do this you need the address of the application to which you want to connect. This address consists of three key parts: the protocol family, Internet Protocol (IP) address and the service port number.

We've already talked about the IP address and service port, but what's the protocol family? It's a number which is used to logically designate the group to which a given protocol belongs. Since the socket interface is general enough to be used with several different protocols, the protocol family tells

the underlying network software which protocol is being used by the socket. In our case, the Internet Protocol family will always be used when creating sockets. With the protocol family, IP address of the system and the service port number, you're ready to establish a connection.

Client-Server Applications

Programs written to use TCP are developed using the client-server model. As mentioned previously, when two programs wish to use TCP to exchange data, one of the programs must assume the role of the client, while the other must assume the role of the server. The client application initiates what is called an active open. It creates a socket and actively attempts to connect to a server. The server application creates a socket and passively listens for incoming connections from clients, performing what is called a passive open.

When the client initiates a connection, the server is notified that some process is attempting to connect with it. By accepting the connection, the server completes what is called a virtual circuit, a logical communications pathway between the two programs. It's important to note that the act of accepting a connection creates a new socket; the original socket remains unchanged so that it can continue to be used to listen for additional connections. When the server no longer wishes to listen for connections, it closes the original passive socket.

To review, there are five significant steps that a program which uses TCP must take to establish and complete a connection. The server side would follow these steps:

1. Create a socket.
2. Listen for incoming connections from clients.
3. Accept the client connection, creating a new socket.
4. Send and receive information.
5. Close the socket when finished, terminating the conversation.

In the case of the client, these steps are followed:

1. Create a socket.
2. Specify the address and service port of the server program.
3. Establish the connection with the server.
4. Send and receive information.
5. Close the socket when finished, terminating the conversation.

Only steps two and three are different, depending on if it's a client or server application. When creating a server application, it is generally desirable to use a multi-threaded design. After accepting the

connection, a thread is created to handle the socket I/O for that particular session. When the connection is closed, the thread can either be terminated or put to sleep until a new client connection has been accepted. This kind of implementation has a number of advantages, particularly on systems with multiple processors. A multi-threaded server is able to handle a larger number of client connections more efficiently, and generally it is easier to implement and debug because each client session operates independently from the others.

Asynchronous Sockets

One of the first issues you'll encounter when developing your Internet application is the difference between synchronous and asynchronous sockets. Whenever you perform some operation on a socket, it may not be able to complete immediately and return control back to your program. For example, reading data from a socket cannot complete until some data has been sent by the remote host. If there is no data waiting to be read, one of two things can happen: the method can wait until some data has been written on the socket, or it can return immediately with an error that indicates that there is no data to be read.

The first type of socket is called a synchronous or blocking socket. In other words, the program is "blocked" until the request for data has been satisfied. When the remote system does write some data on the socket, the read operation will complete and execution of the program will resume. The second type of socket is called an asynchronous or non-blocking socket, and requires that the application recognize the error condition and handles the situation appropriately.

Programs that use asynchronous sockets typically use one of two methods when sending and receiving data. One method is to have the program periodically attempt to read or write data from the socket, typically using a timer. However, this can result in higher CPU utilization and negatively impact the overall performance of the system. The preferred method is to use what is called asynchronous notification. This means that the program is notified whenever a socket event takes place, and in turn can respond to that event. For example, if the remote program writes some data to the socket, an event is generated so that program knows it can read the data from the socket at that point.

By default, socket I/O is synchronous and control is not returned to the program until the blocking socket operation has completed. However, this can introduce some problems on the Windows platform. When the socket blocks, it will either cause the current thread to sleep, or it will process Windows messages sent to the application. This can either result in the program appearing to become non-responsive or it may be re-entered at a different point with the blocked operation parked on the program's stack. For example, consider a program that attempts to read some data from the socket when a button is pressed. Because no data has been written yet, it blocks and the program goes into a message loop. The user then presses a different button, which causes code to be executed, which in turn attempts to read data from the socket, and so on.

To resolve this issue with blocking sockets, the Windows Sockets standard states that there may only be one outstanding blocked call per thread of execution. This means that applications that are re-entered

(as in the example above) will encounter errors whenever they try to take some action while a blocking function is already in progress. The creation of background worker threads to perform blocking socket operations is a common approach to address this issue. This allows the main user interface thread to continue to process Windows messages and remain responsive to the user.

It should be noted that there are advantages to using blocking sockets. In most cases, the application design and implementation is simpler, and overall throughput (the rate at which data is sent and received) is generally higher with blocking sockets because it does not have to go through an event mechanism to notify the application of a change in status.

The SocketWrench component facilitates the use of non-blocking sockets by firing events when appropriate. For example, an **OnRead** event is generated whenever the remote host writes on the socket, which tells your application that there is data waiting to be read.

In summary, there are three general approaches that can be taken when building an application:

- Use a synchronous (blocking) socket. In this mode, the program will not resume execution until the socket operation has completed. If multiple simultaneous connections must be established by the application, it is recommended that you use a multi-threaded design, where each session is managed by its own worker thread.
- Use an asynchronous (non-blocking) socket. This mode allows your application to respond to events. For example, when the remote system writes data to the socket, an **OnRead** event is generated for the control. Your application can respond by reading the data from the socket, and perhaps send some data back, depending on the context of the data received.
- Use a combination of blocking and non-blocking socket operations. The ability to switch between blocking and non-blocking modes "on the fly" provides a powerful and convenient way to perform socket operations.

If you decide to use asynchronous sockets in your application, it's important to keep in mind that you must check the return value from every read and write operation, since it is possible you may not be able to send or receive all the specified data.

Developers frequently encounter problems when they write a program that assumes a given number of bytes can always be written to, or read from, the socket. In many cases, the program works as expected when developed and tested on a local area network, but fails unpredictably when the program is released to a user that has a slower network connection (such as a serial dial-up connection to the Internet). By always checking the return values of these operations, you insure that your program will work correctly, regardless of the speed or configuration of the network.

Secure Communications

Security and privacy is a concern for everyone who uses the Internet, and the ability to provide secure transactions over the Internet has become one of the key requirements for many business applications.

SocketWrench has the ability to establish secure connections with remote servers, as well as function as a secure server itself. Although most of the technical issues such as data encryption are handled internally by the control and library, a general understanding of the standard security protocols is useful when designing your own applications.

When you establish a connection to a server over the Internet (for example, a web server), the data that you exchange is typically routed over dozens of computer systems until it reaches its destination. Any one of these systems may monitor and log the data that it forwards, and there is no way for either the sender or receiver of that data to know if this has been done. Exchanging information over the Internet could be likened to talking with someone in a public restaurant. Anyone can choose to listen to what you're saying, and unless they introduce themselves, you have no idea who they are or if they've even heard what you said.

To ensure that private information can be securely exchanged over the Internet, two basic requirements must be met: there must be a way to send that information so that only the sender and the receiver can understand what is being exchanged, and there must be a way for them to determine that they each are in fact who they claim to be. The solution to the first problem is to use encryption, where a key is used to encrypt and decrypt the data using a mathematical formula. The second problem is addressed by using digital certificates. These certificates are issued by a Certification Authority (CA), which is a trusted third-party organization who verifies the identity of the individual or company who is issued a certificate. These two concepts, encryption and digital certificates, are combined to provide the means to send and receive information securely over the Internet.

The Secure Sockets Layer (SSL) protocol was originally developed by Netscape, and is still the most common protocol in use today. The latest improvements to SSL have resulted in the Transport Layer Security (TLS) protocol, and it is beginning to replace SSL as the standard for secure communications over the Internet. Microsoft also developed a protocol similar to SSL called the Private Communication Technology (PCT) protocol; however, it is not widely used. Each of these protocols were designed to provide essentially the same thing: a private exchange of encrypted data between the sender and receiver, making it unreadable by an intermediate system. Using the restaurant analogy, it would be as if two people were speaking in a language that only they could understand. Although someone sitting at the next table could listen in on the conversation, they wouldn't have any idea what was actually being said.

A secure connection, for example between a web browser and a server, begins with what is called the handshake phase where the client and server identify themselves. When the client first connects with the server it sends a message to the server and the server responds with its digital certificate, along with its public key and information about what type of encryption it would like to use. Next, the client generates a master key and sends this key to the server, which authenticates it.

Once the client and server have completed this exchange, keys are generated which are used to encrypt and decrypt the data that is exchanged. With the handshake completed, a secure connection between the client and server is established. SocketWrench handles the handshake phase of the secure

connection transparently and does not require any additional programming. If a secure connection cannot be established, an error is returned and the network connection is closed.

After the handshake phase has completed, the client may choose to examine the digital certificate returned by the server. The information contained in the certificate includes the date it was issued, the date it expires, information about the organization who issued the certificate (called the issuer) and to whom the certificate was issued (called the subject of the certificate). The client may also validate the status of the certificate, determining if it was issued by a trusted Certification Authority and was returned by the same company or individual it was issued to.

There may be certain cases where the client determines there's a problem with the certificate (for example, if the certificate's common name does not match the domain name of the server), but chooses to continue communicating with the server. Note the connection with the server will still be secure in this case. In other cases, for example if the certificate has expired, the client may choose to terminate the connection and warn the user.

Digital Certificates

With secure Internet connections, digital certificates are used to exchange public keys for data encryption and to provide identification information. This information typically includes the organization that was issued the certificate, its physical location and so on. The certificate itself is used to validate the public key actually belongs to the entity it was issued to. The certificate also includes information about the Certification Authority (CA) who issued the certificate.

The Certification Authority is responsible for validating the information provided by that organization, and then digitally signing the certificate, establishing a relationship between the two entities. When others validate the certificate, they know it has been issued by a trusted third-party.

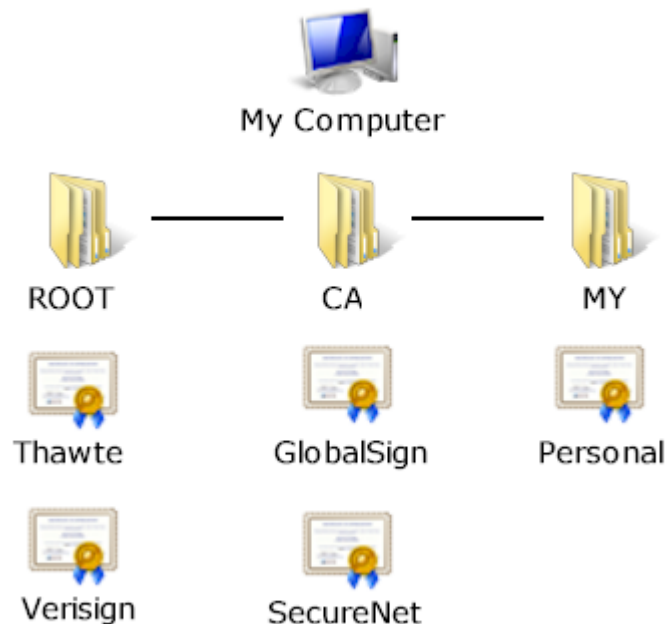
For example, let's say that a company wants to implement a secure site so that people can order products online. They would provide information about their company (organizational contacts, financial information and so on) to a trusted third party organization such as Verisign. Verisign would then verify that the information they provided was complete and correct, and then would issue a signed certificate to them, which they install on their server.

When a user (client system) connects to their server and checks the certificate, they see that it was issued by Verisign, a trusted Certification Authority. In essence, the user is saying that because they trust Verisign, and Verisign trusts the company the certificate was issued to, they will trust the company as well.

To establish this relationship between the Certification Authority and the organization the certificate is issued to, there needs to be a root certificate which has been signed by the same trusted organization. This serves as the beginning of the certification path that is used to validate signed certificates. Using the above example, on the user's system there is a root certificate for Verisign, signed by Verisign.

Root certificates are maintained in the local system's certificate store which is essentially a database of digital certificates. This database is structured so that different types of certificates can be organized in one central location on the system, and a standard interface is provided to enumerate and validate these certificates. Certificates are associated with a store name, allowing them to be easily categorized. For example, root certificates are stored under the name "Root", while a user's personal certificates (along with their private keys) are stored under the name "My".

Windows Certificate Store



When the Windows operating system is installed, there is a certificate store that contains the root certificates for the major Certification Authorities. However, there are situations where additional certificates may need to be added to the system. To facilitate this, the system management console can be used to install certificates, as well as export or remove certificates from the certificate store. When managing your system's certificate store, you should exercise the same caution as when you make changes to the system registry. Inadvertently removing a certificate could result in errors when attempting to access secure systems.

In general, the one situation where certificate management becomes important is when you want to develop your own secure server. This is because your server needs to have a signed certificate to send to the client in order to establish the secure connection. For general-purpose commercial applications, this generally means you would need to obtain a certificate signed by a Certification Authority such as Verisign. This certificate would then be installed in the certificate store on the server.

For development and testing purposes it may be inconvenient to purchase a certificate. There also may be situations in which an organization wishes to function as its own Certification Authority and issue certificates themselves. This allows the organization to control how certificates are managed and can be ideal for secure applications that are designed for the corporate intranet. SocketWrench includes CreatCert, a utility that enables you to create your own self-signed root certificates and server certificates. For more information, refer to the documentation included with SocketWrench.

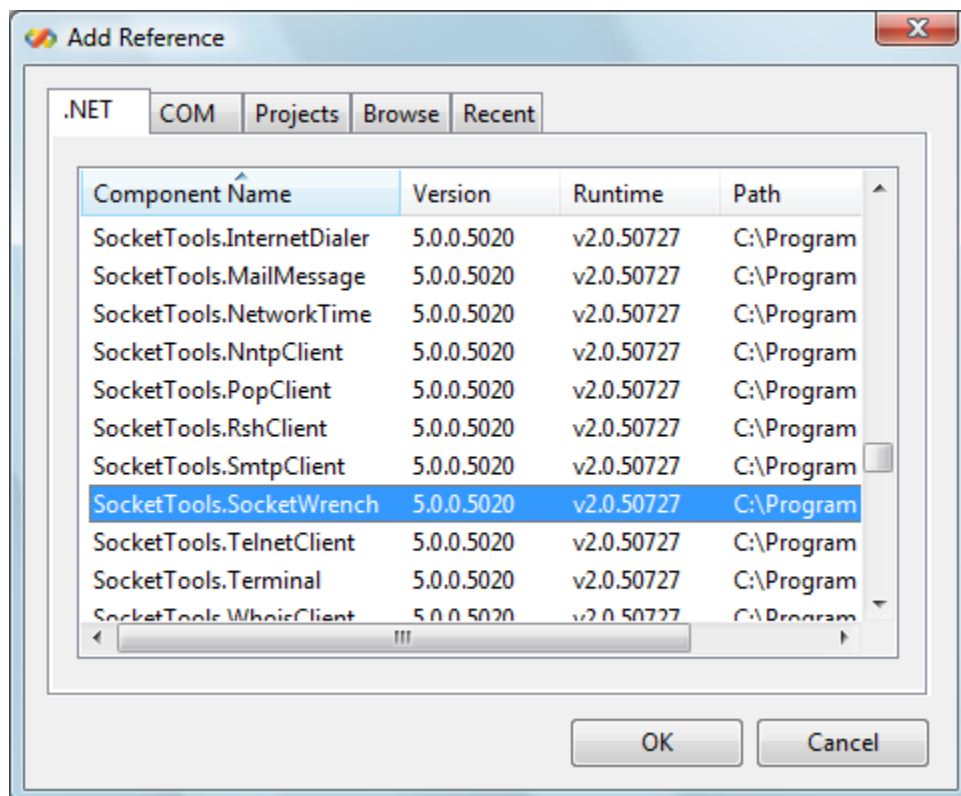
Web Client Example

To get started working with SocketWrench, we'll create a simple Visual Basic application that connects to a web server and retrieves the contents of an HTML page on that server. For this example, you'll need Microsoft Visual Studio 2005, SocketWrench .NET and a working Internet connection.

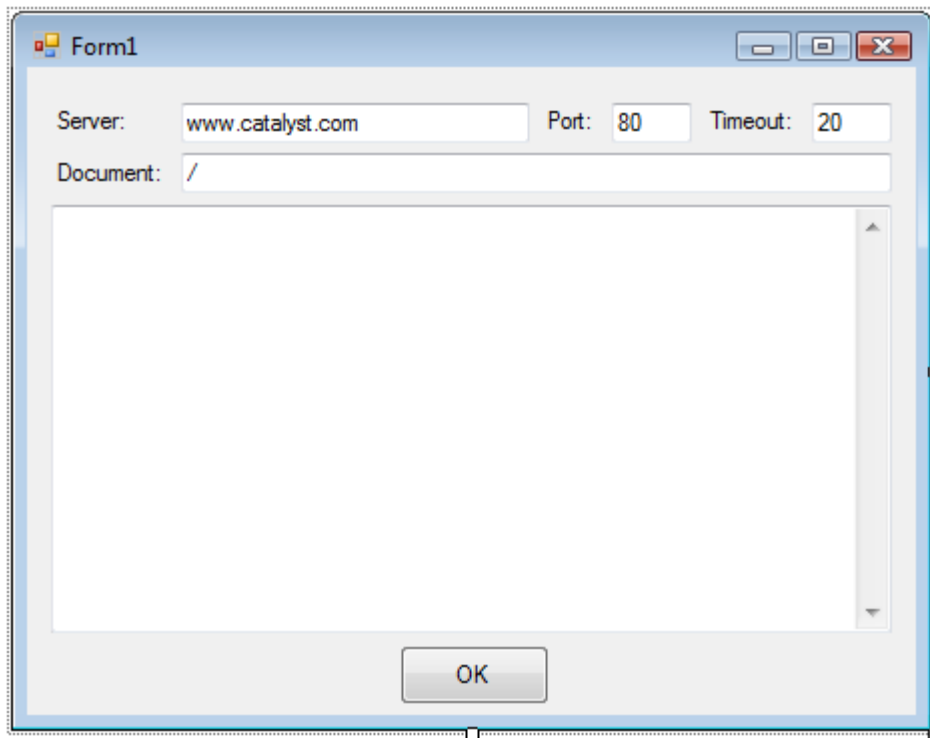
The first step after starting Visual Studio is to create a new Visual Basic project. Select **File | New | Project** from the menu, then select Visual Basic as the project type and Windows Application as the project template. A new project will be created with a single, empty form.

After the project has been created, the next step is to add SocketWrench .NET to your new project. Select **Project | Add Reference** from the menu and a dialog will be displayed which lists the available components. Scroll down to where SocketTools.SocketWrench is displayed, select it and then click on the OK button.

The User Interface



After the SocketWrench component has been referenced in your project, the next step is to create the user interface. We'll add several Label and TextBox controls, and a Button control.



In this example, the large TextBox control in the center of the form has the **Multiline** property set to True, the **WordWrap** property is set to False, and the **ScrollBars** property is set to Both. This will enable us to view the complete document that's retrieved from the web server and scroll through the contents. We have also initialized the TextBox controls with default values that we'll use with this example.

When the application is completed, you will be able to specify the name of the web server, the port number, a timeout period in seconds, and the document that you wish to view. Clicking the OK button will establish a connection with the server, request the document and then retrieve the contents of the document that the server returns.

With the user interface portion of the example complete, the next step is to write the code that will retrieve the document from the server. To do this, select **View | Code** from the menu, and an editor will display the empty form class.

Initializing SocketWrench

The first step is creating an instance of the SocketWrench component and then initializing that instance. To create an instance of SocketWrench, we'll create a private member variable in the form's class that references the component, and name it as "Socket":

```
Imports SocketTools.SocketWrench
```

```
Public Class Form1
    Private Socket As New SocketTools.SocketWrench
End Class
```

Next, we need to initialize that instance of the component when the form is created. This is done by calling the **Initialize** method, and it should be the first method that you call. SocketWrench uses explicit initialization like this because it dynamically loads the networking libraries that it requires. In turn, this allows your application to handle error conditions gracefully if the initialization process fails for some reason. A good place to call the **Initialize** method is in the form's Load event, so we'll create an event handler that does this:

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs)
    Handles Me.Load

    If Not Socket.Initialize() Then
        MessageBox.Show("Unable to initialize SocketWrench component")
    End If
End Sub
```

If the **Initialize** method fails, it will return a value of False. In that case, we'll simply display an error message and terminate the program. Failure can indicate that there was a problem loading the networking subsystem, or an internal problem with the SocketWrench component or how it was installed. If you continue to encounter this error, try reinstalling SocketWrench and see if that resolves the problem.

For each call to the **Initialize** method, there should be a corresponding call to the **Uninitialize** method when that instance of SocketWrench is no longer being used. Because sockets are a limited system resource, it's a good idea to explicitly release them when they're no longer needed. Although the .NET garbage collector will eventually do this for you, it's recommended that you do this explicitly. A good place to call the Uninitialize method is in the **FormClosed** event:

```
Private Sub Form1_FormClosed(ByVal sender As Object, ByVal e As
    System.Windows.Forms.FormClosedEventArgs) Handles Me.FormClosed

    Socket.Uninitialize()

End Sub
```

The **Uninitialize** method is typically invoked immediately before the application terminates. It will close any active network connection created by that instance of SocketWrench, and will release all of the system resources that were allocated for it. Once this is done, that instance of the component can no longer be used.

Establishing a Connection

After an instance of the SocketWrench component has been created and initialized, the next step is to establish a connection with the web server. We'll do this in the **Click** event handler for our Button control that was placed on the form.

To connect to the server, we'll use the **Connect** method. It's important to note that the **Connect** method can be called in several different ways, so it's recommended that you review the technical reference for more information. In this example, we'll use the server name, port number and timeout value the user provides. The TextBox1 control specifies the server name or IP address, the TextBox2 control specifies the port number, and the TextBox3 control specifies the timeout period in seconds. Here's what our **Click** event handler looks like so far:

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Button1.Click

    ' Initialize the SocketWrench properties, setting the server
    ' host name, remote port number and timeout period.

    Socket.HostName = TextBox1.Text
    Socket.RemotePort = Integer.Parse(TextBox2.Text)
    Socket.Timeout = Integer.Parse(TextBox3.Text)

    ' Establish a connection with the web server, and display
    ' an error message if the connection attempt fails.

    If Not Socket.Connect() Then
        MessageBox.Show(Socket.LastErrorString)
        Exit Sub
    End If

End Sub
```

First, we initialize a few properties to the values provided by the user. The **HostName** property specifies the host name or IP address of the server that we want to connect to. The **RemotePort** property specifies the port number that will be used to establish the connection. The **Timeout** property specifies the number of seconds to wait for the connection to be established.

Next, we call the **Connect** method and check the return value. If the method returns True, the connection has been established successfully. If it returns False, an error has occurred. In that case, we display a message box with a description of the last error using the **LastErrorString** property. There is also a property named **LastError** which returns a numeric error code that identifies the last error that has occurred.

One important thing to keep in mind using the **LastError** and **LastErrorString** properties is that their values are only meaningful if the previous method indicates an error has actually occurred, either by returning a value that indicates failure or throwing an exception. This means you should never check the

value of the **LastError** property to determine if the previous method has failed; instead, you should always check the method's return value.

Requesting the Document

Once the connection has been established, the next step is to request the document from the web server. To do this, we'll send the GET command to the server, along with the name of the document we would like to view. To send this command, we'll use the **WriteLine** method which sends a line of text to the remote host, terminating it with a carriage return and linefeed character sequence. This is very similar to how a line of text is written to a text file. Here's our updated code in the **Click** event handler to send the request to the server:

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Button1.Click

    ' Initialize the SocketWrench properties, setting the server
    ' host name, remote port number and timeout period.

    Socket.HostName = TextBox1.Text
    Socket.RemotePort = Integer.Parse(TextBox2.Text)
    Socket.Timeout = Integer.Parse(TextBox3.Text)

    ' Establish a connection with the web server, and display
    ' an error message if the connection attempt fails.

    If Not Socket.Connect() Then
        MessageBox.Show(Socket.LastErrorString)
        Exit Sub
    End If

    ' Send the GET command to the server using the WriteLine
    ' method, requesting the document that the user has specified.

    If Not Socket.WriteLine("GET " + TextBox4.Text) Then
        MessageBox.Show(Socket.LastErrorString)
        Socket.Disconnect()
        Exit Sub
    End If

End Sub
```

The TextBox4 control specifies the name of the document we want to retrieve from the server. If we use "/" as the document name, it tells the server we want the default index page. The GET command is combined with the name of the document and sent to the server using the **WriteLine** method. It is important to note this is the simplest form of the GET command, and is used for demonstration purposes here because of its simplicity. For an example of how a more complex GET request can be performed, refer to the ViewPage example included with SocketWrench .NET.

If the **WriteLine** method returns True, then that indicates the command has been sent successfully to the server. If it returns False, then the operation has failed and a message box is displayed to the user. If that occurs, we also terminate the connection to the server by calling the **Disconnect** method.

It's important to note that **WriteLine** is not the only way that you can send data to a server. For example, SocketWrench also provides a more general purpose **Write** method which can be used to send data stored in a byte array, and can be used when binary (non-textual) data must be exchanged with the server. There's also the higher level **WriteStream** method which can be used to send very large amounts of data in a single method call. For more information, refer to the Technical Reference that is included with SocketWrench.

Reading the Document

After the command requesting the document has been sent, the next step is to read the contents of the document returned by the server. This is done using the **ReadLine** method, which reads a line of text from the server and returns it in a string. We call the **ReadLine** method in a loop, reading each line of text returned by the server, until there is no more data available to be read. Here is the complete code for the **Click** event handler:

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Button1.Click

    ' Initialize the SocketWrench properties, setting the server
    ' host name, remote port number and timeout period.

    Socket.HostName = TextBox1.Text
    Socket.RemotePort = Integer.Parse(TextBox2.Text)
    Socket.Timeout = Integer.Parse(TextBox3.Text)

    ' Establish a connection with the web server, and display
    ' an error message if the connection attempt fails.

    If Not Socket.Connect() Then
        MessageBox.Show(Socket.LastErrorString)
        Exit Sub
    End If

    ' Send the GET command to the server using the WriteLine
    ' method, requesting the document that the user has specified.

    If Not Socket.WriteLine("GET " + TextBox4.Text) Then
        MessageBox.Show(Socket.LastErrorString)
        Socket.Disconnect()
        Exit Sub
    End If

    ' Declare the string buffer that will contain the text
    ' returned by the server, and a Boolean variable.

    Dim strBuffer As String = String.Empty
    Dim bContinue As Boolean = True
```

```
' Call the ReadLine method in a loop until it returns a
' value of False. For each line of text that is read,
' append it to the multiline TextBox control.

Do
    bContinue = Socket.ReadLine(strBuffer)
    If bContinue Then
        TextBox5.AppendText(strBuffer + vbCrLf)
    End If
Loop Until bContinue = False

' Disconnect from the server, releasing the socket handle
' that was allocated for this session.

Socket.Disconnect()

End Sub
```

When there is no more data available to be read, the **ReadLine** method will return False, and the loop will be exited. At that point, the only remaining step is to terminate the connection by calling the **Disconnect** method. This gracefully closes the connection and releases the socket handle that was allocated for the connection.

Keep in mind that there is also more than one way you can read data from the server. While the **ReadLine** method is designed to read lines of text, there is also a general purpose **Read** method that can be used to receive binary (non-textual) data and store it in a byte array. There's also the higher level **ReadStream** method which can receive an arbitrarily large amount of data in a single method call and return it in a buffer that you provide. For more information about these methods, please review the Technical Reference included with SocketWrench.

Now that we have completed the code for the Button control's **Click** event, the final step is to run the example. Pressing F5 will execute the program, and clicking the OK button will retrieve the index page from the Catalyst Development web server. Although this was a very simple example, it demonstrated the important aspects of creating an Internet application using SocketWrench:

1. Using the **Initialize** method after creating an instance of the component. This should be done shortly after the new instance is created, and before any other methods are invoked.
2. Using the **Connect** method to establish a connection using a specific host name and port number. The return value from the **Connect** method will tell you if the connection was successful, or if an error has occurred.
3. Sending data to the server using the **WriteLine** method, and receiving data from the server using the **ReadLine** method. The return value will indicate if the method was successful, if there was an error or there is no more data available to read.
4. Terminating the connection by calling the **Disconnect** method. This method should be invoked when the session has completed, releasing the socket handle that was allocated by the previous call to the **Connect** method.

These steps are common to virtually every application you'll create with SocketWrench, and are similar to the steps discussed in the previous section about client-server applications.

Creating a Secure Connection

To complete our example, we'll make a few modifications to also support secure connections to the web server using the standard SSL and TLS protocols. One of the benefits of SocketWrench is that this is a simple change, typically requiring that you set and check the value of a few properties. The initial client-server handshake, certificate validation and the data encryption and decryption are all handled for you automatically.

According to the Hypertext Transfer Protocol (HTTP), the standard protocol for web servers, the default port number for standard (non-secure) connections is port 80. For secure connections using SSL/TLS, the default port number is 443. To accommodate this, we'll modify the example to check if the user has specified port 443, and if they have, establish a secure connection to the server. After setting the **HostName**, **RemotePort** and **Timeout** properties, add this code:

```
' If the user has specified port 443, then this should be a
' secure connection to the web server

If Socket.RemotePort = 443 Then
    Socket.Secure = True
End If
```

The **Secure** property tells SocketWrench that you wish to establish a secure connection with the remote host. In most cases, this is all you will have to do to enable the security features in your application. SocketWrench will automatically select the appropriate security protocol for you, check the certificate returned by the remote host and negotiate for the strongest encryption algorithm supported by both the client and server.

Once a secure connection has been established, a number of other security related properties become available to the control. These properties fall into two general groups, returning information either about the secure connection itself, or about the server's digital certificate. The properties which provide information about the connection are:

- | | |
|-----------------------|--|
| CipherStrength | This property returns information about the relative strength of the encryption that is being used to secure the data. The value returned is actually the length of the key (in bits) used by the encryption algorithm, and will typically be 40, 56 or 128. A key length of 40 bits is considered weak, while a key length of 56 bits is considered to be moderate and 128 bit or higher keys are considered very secure. |
| HashStrength | This property returns information about the strength of the message digest (hash) that was selected. Common values returned by this property are 128 and 160. |

SecureCipher	This property identifies the encryption algorithm that was selected. The algorithms supported are RC2, RC4, DES, and Triple DES. The most commonly used algorithms are AES and RC4.
SecureHash	This property identifies the message digest (hash) algorithm that was selected. The algorithms supported are SHA and MD5. The most commonly used message digest is MD5. This algorithm is used during the handshake phase between the client and server, and is made available to the client for informational purposes.
SecureKeyExchange	This property identifies the key exchange algorithm that was selected. The algorithms supported are RSA, KEA and Diffie-Hellman. The most commonly used key exchange algorithm is RSA.
SecureProtocol	This property identifies the protocol used to establish the secure connection. The protocols supported are SSL 2.0, SSL 3.0, PCT 1.0 and TLS 1.0.

In addition to information about the secure connection, there are several properties which return information about the remote server's digital certificate. These properties are:

CertificateExpires	This property returns the date the server's certificate expires. If this value is earlier than the current date, the certificate has expired. In that case, it is recommended that you alert the user that the certificate is no longer valid.
CertificateIssued	This property returns the date the server's certificate was issued by the certificate authority. If this date is later than the current date, or later than the date the certificate was issued, the certificate is invalid.
CertificateIssuer	This property returns information about the organization that issued the certificate. The data is returned as a string which contains one or more tagged name and value pairs.
CertificateStatus	This property returns information about the status of the certificate. The client is responsible for checking this value, and based on the value returned, decide if the connection should be terminated or not.
CertificateSubject	This property returns information about the organization to which the certificate was issued. Like the CertificateIssuer property, this property returns a string which contains one or more tagged name and value pairs.

It is recommended your application immediately check the value of the **CertificateStatus** property after the secure connection has been established. This allows your application to make the decision as to whether or not it is safe to communicate with the server based on the status of the digital certificate it returns. For example, using the above code the **CertificateStatus** property would return a value of

SecurityCertificate.certificateValid, which indicates the certificate is valid. After the call to the Connect method, add this code:

```
' If this is a secure connection, then check the status
' of the server's certificate. If the certificate is not
' valid, warn the user and terminate the connection.

If Socket.Secure And _
    Socket.CertificateStatus <> SecurityCertificate.certificateValid Then
    MessageBox.Show("There is a problem with the site certificate")
    Socket.Disconnect()
    Exit Sub
End If
```

Although this example terminates the connection if the certificate cannot be validated, keep in mind this is not required. For example, you could modify the program to display a message box indicating there is a problem with the certificate and asking if they want to continue. Even if the certificate cannot be validated, the connection is still encrypted. However, an invalid certificate can indicate the site has been compromised, so it is always recommended you strongly advise your users of the potential risks.

If you wanted to display specific information about a certificate, for example, the name of the organization that issued the certificate or the name of the company that it was issued to, you would need to use the **CertificateIssuer** and **CertificateSubject** properties. These are string properties that return one or more comma-separated values. Each value provides information about the certificate. For example, if the **CertificateSubject** property returns this string:

```
C=US, O="RSA Data Security, Inc."
```

That would specify two pieces of information that are available for this certificate:

1. C=US
2. O="RSA Data Security, Inc."

Each of these values consists of an identifier called an RDN (Relative Distinguished Name) and its data. Since the second value contains a comma, it is enclosed in quotes, and this needs to be accounted for when parsing the string. There are a predefined set of RDNs defined by the X.500 standard which are used in certificates. The most commonly used RDNs in X.509 certificates are:

RDN	Description
C	The ISO standard two character country code.
S	The name of the state or province.
L	The name of the city or locality.
O	The name of the company or organization.
OU	The name of the department or organizational unit.
CN	The common name; with X.509 certificates, this is the domain name of the site the certificate was issued to.

So, for example, if you wanted to determine the domain name that a certificate was issued to, you would need to read the value of the **CertificateSubject** property and parse the resulting string for the "CN" (Common Name) RDN.

With the changes we've made to the example to support secure connections, here's what our complete Button **Click** event handler looks like:

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Button1.Click

    ' Initialize the SocketWrench properties, setting the server
    ' host name, remote port number and timeout period.

    Socket.HostName = TextBox1.Text
    Socket.RemotePort = Integer.Parse(TextBox2.Text)
    Socket.Timeout = Integer.Parse(TextBox3.Text)

    ' If the user has specified port 443, then this should be a
    ' secure connection to the web server

    If Socket.RemotePort = 443 Then
        Socket.Secure = True
    End If

    ' Establish a connection with the web server, and display
    ' an error message if the connection attempt fails.

    If Not Socket.Connect() Then
        MessageBox.Show(Socket.LastErrorString)
        Exit Sub
    End If

    ' If this is a secure connection, then check the status
    ' of the server's certificate. If the certificate is not
    ' valid, warn the user and terminate the connection.

    If Socket.Secure And _
        Socket.CertificateStatus <> SecurityCertificate.certificateValid Then
        MessageBox.Show("There is a problem with the site certificate")
        Socket.Disconnect()
        Exit Sub
    End If

    ' Send the GET command to the server using the WriteLine
    ' method, requesting the document that the user has specified.

    If Not Socket.WriteLine("GET " + TextBox4.Text) Then
        MessageBox.Show(Socket.LastErrorString)
        Socket.Disconnect()
        Exit Sub
    End If

    ' Declare the string buffer that will contain the text
```

```
' returned by the server, and a Boolean variable.

Dim strBuffer As String = String.Empty
Dim bContinue As Boolean = True

' Call the ReadLine method in a loop until it returns a
' value of False. For each line of text that is read,
' append it to the multiline TextBox control.

Do
    bContinue = Socket.ReadLine(strBuffer)
    If bContinue Then
        TextBox5.AppendText(strBuffer + vbCrLf)
    End If
Loop Until bContinue = False

' Disconnect from the server, releasing the socket handle
' that was allocated for this session.

Socket.Disconnect()

End Sub
```

In summary, to establish a secure connection you need to set the **Secure** property to a value of True, and then you should check the value of the **CertificateStatus** property after the Connect method has returned. If the certificate is valid, you can proceed normally. If there is a problem with the certificate, you should either terminate the connection or alert the user there is a problem and let them make that decision.

Debugging Applications

One of the issues that every developer has to contend with is problems that arise in an application after it's been distributed to end-users. And errors related to Windows Sockets programming can be even more difficult to track down because there are so many variables involved (such as the platform, operating system version, system configuration, and so on). To address these difficult problems, the SocketWrench control has the built-in ability to log the Windows Sockets API function calls that are made. There are three properties related to creating a log file: **Trace**, **TraceFile** and **TraceFlags**. Setting these properties enables your application to dynamically manage function tracing features available to the control. The **Trace** property is a Boolean flag which simply enables or disables the function tracing feature.

The **TraceFile** property specifies the name of a trace log file in which each function and its parameters will be written. If this property is not explicitly set, then a file named **SocketTools.log** will be created in the system's temporary directory (the directory specified by the TEMP environment variable). The **TraceFlags** property specifies what type of logging will be performed by the control, and may be set to one of four values: 0 (TRACE_ALL) in which all functions will be logged, 1 (TRACE_ERROR) in which only errors will be logged, 2 (TRACE_WARNING) in which case both warnings and errors will be written to the

log file, and 4 (TRACE_HEXDUMP in which all functions will be logged, together with ASCII and hexadecimal displays of all data that is sent or received on sockets. By default, all function calls are logged by the control (TRACE_ALL).

The trace file has the following format:

```
EXAMPLE1 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
EXAMPLE1 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
EXAMPLE1 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process being traced. The second column identifies if the record is reporting information, a warning, or an error. What follows is the name of the Windows Sockets function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record within brackets.

When reading a trace log, there are two common things that you will see:

1. The error code 10035, which corresponds to the Windows Sockets error WSAEWOULDBLOCK is a normal occurrence on connect calls, and should not be taken as a cause for concern by itself.
2. The normal return value for the **select** function is greater than zero, typically a value of one. A select call that returns zero usually indicates a timeout. A return value of -1 indicates an error condition.

If parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value prefixed with "0x". A special type of pointer, called a null pointer, is recorded as NULL.

Those functions which expect socket addresses are displayed in the format "aa.bb.cc.dd:nnnn". The first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. Note that in the second line of the above example, the application is attempting to connect to a system with the IP address 192.0.0.1 on port 1234.

To enable logging in your application, you also need to redistribute an extra file called **SocketTools.TraceLog.dll** and install it in the same directory as your application. If you have set the **Trace** property to True, but this library cannot be loaded, then SocketWrench will silently reset the **Trace** property to False. Note that this library will only provide logging capability to the SocketWrench component; it is not a general purpose library for logging Windows Sockets functions and will not log the function calls made by any other application or component.

There are several ways you could incorporate trace logging in your software. The simplest would be a menu item or a command line switch (like /Debug) in which the **Trace** property would be set to True. A more complex approach would be to include a dialog or property sheet which allows the user to specify the log file name and options. When an end-user calls for technical support and is encountering a problem you think may be network related, you can instruct them to enable the logging feature and email or fax you a copy of the log file. In turn, if it is a problem you don't understand, you can send the

log file to a Catalyst Development support technician who can analyze the log and provide you with additional information about the problem.

Remember that if you do not use the logging features at any time during the execution of your program, there is no additional performance penalty. If you do enable logging at some point, the library will be loaded and memory will be allocated by the logging functions. These functions open, append to the trace log file, flush and then close the file for each Windows Sockets function call that is made. This insures that the last function called is logged in case of a general protection fault or other abnormal termination of the program. However, because of the file I/O overhead, it's recommended that your program rename or remove the log file before beginning a new trace.

Advanced Development Using SocketTools

SocketWrench is part of a package developed by Catalyst called SocketTools. In addition to the comprehensive, but fairly low-level, access that SocketWrench provides, SocketTools includes components and libraries for many of the popular Internet application protocols. There are several different editions of SocketTools available, and all editions provide royalty-free redistribution licensing and a thirty day money-back guarantee. Evaluation copies of all of the SocketTools Editions are available for downloading from our website.

SocketTools .NET Edition

The SocketTools .NET Edition is a collection of managed code classes, designed to simplify the integration of Internet functionality into applications built using the Visual Studio development platform. SocketTools .NET is ideal for the Visual Basic or C# developer who requires the ease of use and rapid development features of a component, without the complexities of the native socket classes and without requiring in-depth knowledge of how the various Internet protocols are implemented.

The SocketTools Secure .NET Edition has the additional feature of integrated support for secure, encrypted connections using the industry standard Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. Your data is protected by industrial strength 128-bit encryption, and SocketTools includes support for client certificates and other advanced features. You can also create your own, custom secure client and server applications and there's no need for you to understand the details of certificate management, data encryption or how the security protocols work. All it takes is a few lines of code to enable the security features with very minimal changes to any existing code.

SocketTools Library Edition

The SocketTools Library Edition includes standard Windows dynamic link libraries (DLLs) which can be used in a wide variety of programming languages such as Visual Studio .NET, Visual C++, Visual Basic and Delphi. The Library Edition is ideal for the developer who requires the high performance, minimum resource utilization and flexibility of a lower level interface, without the inherent overhead of ActiveX controls or COM libraries.

The SocketTools Library Edition API has over 800 functions which can be used to develop applications that meet a wide range of needs. SocketTools covers it all, including uploading and downloading files, sending and retrieving email, remote command execution, terminal emulation, and much more.

In addition to the above, the SocketTools Secure Library Edition has integrated support for secure, encrypted connections using the industry standard Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. It provides an interface for all of the major Internet protocols including HTTPS, FTPS, SMTPS, POP3S and IMAPS. Creating a secure connection only requires a few additional lines of code, and doesn't require that you understand the complex details of data encryption or certificate validation. The Secure Library Edition handles all of the details, enabling you to add security features to your application quickly and easily.

SocketTools Scripting Edition

The SocketTools Scripting Edition includes COM components which can be used in a wide variety of scripting languages such as VBScript, JScript and PHP. In addition, the Scripting Edition components can also be used with Visual Studio.NET, Visual C++ and Visual Basic. The Scripting Edition is ideal for the developer who requires the flexibility, ease of use and rapid development features of a component designed specifically for client and server-side scripting.

In addition to the above, the SocketTools Secure Scripting Edition has integrated support for secure, encrypted connections using the industry standard Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. The Secure Editions implement the major secure protocols such as HTTPS, FTPS, SMTPS, POP3S, IMAPS and more.

SocketTools Visual Edition

The SocketTools Visual Edition includes ActiveX controls (OCXs) which can be used in a wide variety of programming languages such as Visual Studio.NET, Visual C++ and Visual Basic. The Visual Edition is ideal for the developer who requires the flexibility, ease of use and rapid development features of a component without the complexities of working with the Windows Sockets API or in-depth knowledge of how the various Internet protocols are implemented.

The SocketTools Visual Edition consists of eighteen ActiveX controls which can be used to develop applications that meet a wide range of needs. SocketTools covers it all, including uploading and downloading files, sending and retrieving email, remote command execution and terminal emulation.

If you're developing commercial applications which require the ability to establish secure, encrypted connections using the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols, the SocketTools Secure Visual Edition includes support for all of the major secure protocols including HTTPS, FTP, SMTPS, POP3S and IMAPS. Your data is protected by industrial strength 128-bit encryption, and SocketTools includes support for client certificates and other advanced features. All it takes is setting a few properties to enable the security features, with very minimal changes to any existing code.

For more information about SocketTools, please visit the Catalyst Development website at <http://www.catalyst.com/products/sockettools/>