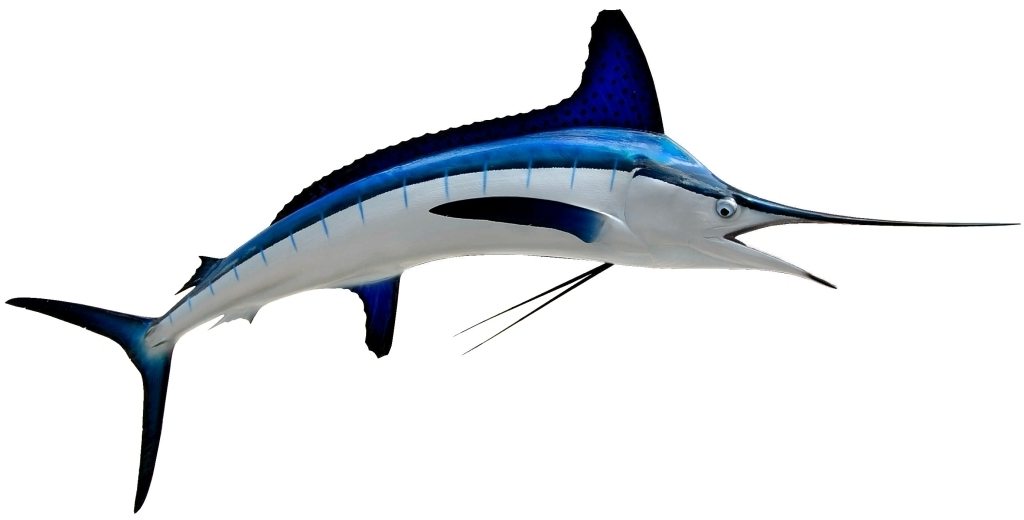
**MARLIN**

Webserver and webclient

3.0



Copyright © 2012 - 2016 ir. W. E. Huisman MSC

This library is released under the MIT license. You are free to use it in any way you like, provided that you leave a reference to the original software. See the file license.txt for further details.

**MANUAL AND DOCUMENTATION**

**Introduction and purpose**

This document is about the Marlin webserver library. It’s structure and how to make use of it in your programs.

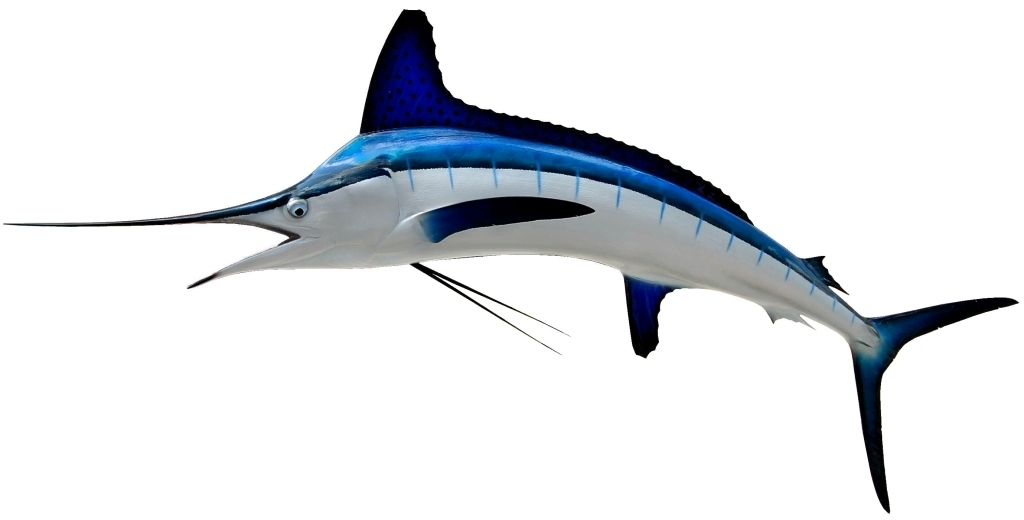
The ‘Marlin’ components – webserver and web client – are built in C++ around a number of general classes that take care of optimal performance and have the purpose that the server and client parts can be plugged-in in a C++ project.

The main reason to build this library was to expand an existing web server with basic HTTPS capabilities like SSL and TLS connections and advanced authentication possibilities like digest- and Kerberos authentication. Instead of building all those components again, I choose to use the general existing Microsoft components ‘HTTP-Server API 2.0’ and ‘WinHTTP API 5.1’

Reason to use C++ for these components – and no .NET technology – has to do with the performance of the components in comparison to the WCF services. Typically these compo­nents are a many times faster than a WCF implementation in Dot NET. Certainly if you configure all kinds of W3C standards like signing of messages, security encryption and reliable-messaging.

Another reason was the need to be able to run a webservice on remote desktops (Citrix !) environments. This was clearly not going to happen with the IIS ISAPI framework. Although IIS 7.0 and higher has done much good with the new integrated pipelining of the requests, enabling an IIS server on every desktop was not an option.

**The name**



This is a picture of a white marlin, a common type of marlin in the Atlantic ocean. This is about the fastest fish in that ocean and it’s a very secretive fish. If you manage to catch one on a fishing line, it will struggle on – unseen – in the deep. Like a fast webserver would ☺

**The components consist of**

a number of generic classes with which you can build the functionality of an application. These generic classes are grouped in a communal directory, so that they can be used in the server application, as well as in the client applications. Furthermore an example webserver and webclient program are created. These programs serve as a unit testing framework as well as coding examples for your convenience. This results in the following directories:

* Marlin Directory with the generic classes
* MarlinServer Directory with an example webserver
* MarlinClient Directory with an example client program
* HTTPManager Directory with the HTTP management application

To use these components and examples, u need only open the solution files in Microsoft Visual Studio (2015 version) in het MarlinServer and MarlinClient directory. Both solutions contain a test project and a directory with a sub collection of the generic classes in the Marlin directory.

For directions how to compile and set up a server on your machine, see the “Running the examples” section later on in this document.

**Overview of the architecture**

In this picture, you can see the overview of the total architecture of the Marlin webserver & web client components.



**Programming model: Clientside**

The central programming model is the HTTPClient object. Messages (HTTPMessage or SOAPMessage) are sent to this object through the overloaded “Send” method. After this method returns with the TRUE status, the message contains the answer of the server. You can then pull the answer of the server out of the object. Example 1 shows how this is done.

|  |
| --- |
| // Example 1  //  #include ”HTTPClient.h”  #include ”HTTPMessage.h”  void function()  {  HTTPClient client;  HTTPMessage msg(http\_get,”http://www.microsoft.com/nl-nl”);  if(client.Send(msg))  {  // Success. Get the result  msg.GetFileBuffer().SetFileName(“C:\tmp\MyLocalFile.html”);  msg.GetFileBuffer().WriteFile();  }  else  {  // Error  CString text;  client.GetError(text);  printf(“Error getting file: %s\n”,text);  }  } |

Of course: you can extend this example in many ways, by overriding the default settings of both the client and the message with a number of Set\* methods.

To be able to use webservices, you only need to replace the general HTTPMessage object by the more specialized SOAPMessage object. Add a few parameters to the SOAPMessage and send to your service interface by way of the client. If the “Send()” method returns TRUE, the object will contain the answers of the server. This is shown in example 2:

|  |
| --- |
| // Example 2  //  #include ”HTTPClient.h”  #include ”SOAPMessage.h”  HTTPClient client;  SOAPMessage msg(“http://mycompany.com/mynamespace/”  ,”TestMethodOne”  ,SOAP\_12  ,“https://mycompany.com/TestInterfaceOne/”);  // Fill the message with parameter info  msg.SetParameter(“ParamOne”,”Value1”);  msg.SetParameter(“ParamTwo”,”Value2”);  if(client.Send(msg))  {  // Success. Get the result from the object  CString result = msg.GetParameter(“ResultParameter”);  ...  }  else  {  // Error  CString text;  client.GetError(text);  printf(“Error getting file: %s\n”,text);  } |

Of course, in this manner we can only send 1 SOAP message at the time through our HTTP(S) channel. If we want to add the ability to add WS-Security, WS-ReliableMessaging or WSDL support we cannot do that right away.

But wait: by replacing the “HTTPClient” object with a “WebServiceClient” object these protocols work right out of the box! Just use the WebServiceClient’s “Send()” method will take care of this. We only need to construct a WebServiceClient object with the desired contract and URL for this purpose. See example 3:

|  |
| --- |
| // Example 3  //  #include ”WebServiceClient.h”  #include ”SOAPMessage.h”  WebServiceClient client(“http://mycompany.com/mynamespace/”  ,“https://mycompany.com/TestInterfaceOne/Reliable/”);  SOAPMessage msg(“https://mycompany.com/mynamespace/”,”TestMethodOne”)  client.SetReliable(true,RELIABLE\_ONCE);  ,  // Fill the message with parameter info  msg.SetParameter(“ParamOne”,”Value1”);  msg.SetParameter(“ParamTwo”,”Value2”);  if(client.Send(msg))  {  // Success after WS-Reliable protocol. Get the result from the object  CString result = msg.GetParameter(“ResultParameter”);  ...  }  else  {  // Error  CString text;  client.GetError(text);  printf(“Error getting file: %s\n”,text);  } |

**Programming model: Serverside**

The central programming model is the sitehandler. Each handler is an object class that handles – in principal – one of the HTTP methods. So there is a handler for the GET and PUT commands of the HTTP protocol. Each handler has a central method “Handle”, with the current received message as a parameter. Information is pulled out of this object, after which the object is reset to a null state with the “Reset” method and filled with the answer. There is no need to call a Send() or Answer() method to send the result back to the caller. This is done by the framework automatically after the handler has ended. Here is a (very) simple example to service a GET request. Be advised: do not do this at home!!

|  |
| --- |
| // Example 4  //  #include ”SiteHandlerGet.h”  #define webroot “C:\\inetpub\\myapp\\”  bool SiteHandelerGet::Handle(HTTPMessage\* p\_message)  {  CString filename = webroot + p\_message->GetAbsolutePath();  p\_message.Reset();  p\_message.GetFileBuffer.SetFileName(filename);  } |

To service a basic SOAP request, you need only use the SOAP handler in this same manner. The SiteHandlerSoap has an override of the “Handle” method especially designed to service SOAP requests. Again there is no need to send the answer back. That is done after the handler has ended. Here is a handler to service the request from example 2.

|  |
| --- |
| // Example 5  //  #include ”SiteHandlerSoap”  bool SiteHandlerSoap::Handle(SOAPMessage\* p\_message)  {  // Getting the info  CString paramOne = p\_message.GetParameter(“ParamOne”);  CString paramTwo = p\_message.GetParameter(“ParamTwo”);  // Reseting the object  p\_message.Reset();  // Doing our logic!  if(paramOne == “Value1” and paramTwo == “Value2”)  {  P\_message.SetParameter(“ResultParameter”,”I declare this OK”);  }  } |

Now, how do we create an http handler on the server side? There are several ways which we show here. At first we need to declare an HTTPServer object. That’s the easy part. You then create an HTTPSite, set a few paramters of that site and ‘start’ it.

One of the parameters you set, is a SiteHandler which you declare first.

Here is a more elaborate example that extends the example number 5 just before.

|  |
| --- |
| // Example 6  //  #include ”HTTPServer.h”  #include ”HTTPSite.h”  #include ”SiteHandlerSoap”  class SiteHandlerSoapForMe : public SiteHandlerSoap  {  protected:  Bool Handle(HTTPMessage\* p\_message);  }  bool SiteHandlerSoapForMe::Handle(SOAPMessage\* p\_message)  {  // Getting the info  CString paramOne = p\_message.GetParameter(“ParamOne”);  CString paramTwo = p\_message.GetParameter(“ParamTwo”);  // Reseting the object  p\_message.Reset();  // Doing our logic!  if(paramOne == “Value1” and paramTwo == “Value2”)  {  P\_message.SetParameter(“ResultParameter”,”I declare this OK”);  }  }  void CreatingTheService()  {  CString url(“https://mycompany.com/TestInterfaceOne/”);  HTTPServer server(“TestInterfaceOne”);  HTTPSite\* mysite = server->CreateSite(URLPRE\_Strong,true,443,url);    // Setting the SOAP handler for this site  site->SetHandler(http\_post,new SiteHandlerSoapForMe());  site->AddContentType("application/soap+xml");  // Start the site explicitly  if(site->StartSite())  {  printf("Site started correctly: %s\n",url);  }  else  {  ++error;  printf("ERROR STARTING SITE: %s\n",url);  }  // Now start the server running (in it’s own thread!)  server.Run();  // Or start the server in this thread (commented out)  // Server.RunHTTPServer();  } |

As with the client programming model, we can again substitute the HTTPServer object for a WebServiceServer object. This enables us to handle WS-Security, WS-ReliableMessaging and the checking against a WSDL. The server side also will write the WSDL file on starting the object, unless we tell it not to with “SetGenerateWsdl(false)”.

Here is how we do this:

|  |
| --- |
| // Example 7  //  #include ”WebServiceServer.h”  void CreatingTheService()  {  CString url(“https://mycompany.com/TestInterfaceOne/”);  CString webroot(“C:\\inetpub\\”);  CString namespace(“http://mycompany.com/mynamespace/”);  WebServiceServer server(“TestOne”,webroot,url,URLPRE\_STRING,namespace,10);  // Create our site  HTTPSite\* mysite = server->CreateSite(URLPRE\_Strong,true,443,url);  // Setting the SOAP handler for this site  site->SetSoapHandler(http\_post,new SiteHandlerSoapForMe());    // Running the server  server.Run()  } |

**Bringing it all together**

But if we want to run a group of webservices? Like a service interface and describe it in a WSDL file for the world to know? Well: this can also be done in a WebServiceServer. To enable the WSDL functionality, you first have to register all services in the wsdl cache of that server. This can be done by submitting a copy of an incoming *and* an outgoing SOAP­Message to this cache. In the mean while you can do two things:

* Add extra info to each parameter for the WSDL for the ‘mandatory / optional’ status, and the optionality of the ordering of the elements;
* Provide each message with a service number, for use in the receiving dispatcher. Hold on to that thought and see what happens while programming.

Below is an excerpt from the testing examples, where I first derive a class form the web service server, and defining three services (one, two and three) for this class.

|  |
| --- |
| // Example 8a  //  #include "WebServiceServer.h"  #include "SiteHandlerSoap.h"  #define CONTRACT\_MF 1 // First  #define CONTRACT\_MS 2 // Second  #define CONTRACT\_MT 3 // Third  // DERIVED CLASS FROM WebServiceServer  class TestContract: public WebServiceServer  {  public:  TestContract(CString p\_name  ,CString p\_webroot  ,CString p\_url  ,PrefixType p\_channelType  ,CString p\_targetNamespace  ,unsigned p\_maxThreads);  protected:  WEBSERVICE\_MAP; // Using a WEBSERVICE mapping  // Declare all our webservice call names  // which will translate in the On.... methods  WEBSERVICE\_DECLARE(MarlinFirst)  WEBSERVICE\_DECLARE(MarlinSecond)  WEBSERVICE\_DECLARE(MarlinThird)  private:  // Our functionality  CString Translation(CString p\_language,CString p\_translation,CString p\_word);  // Set input/output languages  CString m\_language;  CString m\_translation;  }; |

After we have instantiated an object of this class we, can add the incoming and outgoing SOAP messages to the WSDL cache of the WebServiceServer through “AddOperation”.

The operations in this example here have numbers 1 (CONTRACT\_MF), 2 (CONTRACT\_MS) and 3 (CONTRACT\_MT).

|  |
| --- |
| // Example 8b  //  #include "WebServiceServer.h"  //////////////////////////////////////////////////////////////////////////  //  // PREPARING OUR WSDL, This is what will fill the WSDL file  //  void  AddOperations(TestContract& p\_server,CString p\_contract)  {  // Defining the names of the operations  CString first ("MarlinFirst");  CString second("MarlinSecond");  CString third ("MarlinThird");  CString respFirst ("ResponseFirst");  CString respSecond("ResponseSecond");  CString respThird ("ResponseThird");  // Defining input and output messages for the operations  SOAPMessage input1 (p\_contract,first);  SOAPMessage output1(p\_contract,respFirst);  SOAPMessage input2 (p\_contract,second);  SOAPMessage output2(p\_contract,respSecond);  SOAPMessage input3 (p\_contract,third);  SOAPMessage output3(p\_contract,respThird);  // Defining the parameters for all the operations  // First: Getting an accepted language  input1 .AddElement(NULL,"Language",WSDL\_Mandatory | XDT\_String, "string");  output1.AddElement(NULL,"Accepted",WSDL\_Mandatory | XDT\_Boolean,"bool");  // Second: Getting an accepted translation  input2 .AddElement(NULL,"Translation",WSDL\_Mandatory | XDT\_String, "string");  output2.AddElement(NULL,"CanDo", WSDL\_Mandatory | XDT\_Boolean,"bool");  // Third Getting the answer  input3 .AddElement(NULL,"WordToTranslate",WSDL\_Mandatory | XDT\_String,"string");  output3.AddElement(NULL,"TranslatedWord", WSDL\_Optional | XDT\_String,"string");  // Putting the operations in the WSDL Cache  p\_server.AddOperation(CONTRACT\_MF,first, &input1,&output1);  p\_server.AddOperation(CONTRACT\_MS,second,&input2,&output2);  p\_server.AddOperation(CONTRACT\_MT,third, &input3,&output3);  } |

Adding the operations to the WebServiceServer derived class, will automatically result in the writing of a WSDL file, unless of course we set “SetGenerateWsdl” to false.

After defining the interface, we can now define the operations themselves. This is the easy part where we define our handlers.

|  |
| --- |
| // Example 8c  //  #include ”WebServiceServer.h”  // Implementation of the TestContract class  TestContract::TestContract(CString p\_name  ,CString p\_webroot  ,CString p\_url  ,PrefixType p\_channelType  ,CString p\_targetNamespace  ,unsigned p\_maxThreads)  :WebServiceServer(p\_name,p\_webroot,p\_url,p\_channelType,p\_targetNamespace,p\_maxThreads)  {  }  // Mapping corresponding to the AddOperation of the WSDL  WEBSERVICE\_MAP\_BEGIN(TestContract)  WEBSERVICE(CONTRACT\_MF,MarlinFirst)  WEBSERVICE(CONTRACT\_MS,MarlinSecond)  WEBSERVICE(CONTRACT\_MT,MarlinThird)  WEBSERVICE\_MAP\_END  //////////////////////////////////////////////////////////////////////////  //  // HERE ARE THE SERVICE HANDLERS!!  // Derived from the definition above in the WEBSERVICE\_MAP  //  //////////////////////////////////////////////////////////////////////////  void  TestContract::OnMarlinFirst(int p\_code,SOAPMessage\* p\_message)  {  ASSERT(p\_code == CONTRACT\_MF);    m\_language = p\_message->GetParameter("Language");  printf("\n");  printf("Setting base language to: %s\n",(LPCTSTR)m\_language);  // Reset message and set answering parameters  p\_message->Reset();  p\_message->SetParameter("Accepted",m\_language == "Dutch");  }  ... more handlers... See “TestContract.cpp” |

After this bit of abstraction you need only to write a “OnXxxxxx” handler for each service call, using the input parameters, resetting the message and providing the answering parameters. All the rest is done by the Marlin framework.

**Overview of the programming model**

After this long explanation of the way of programming, let’s look at how that can be caught in a thinking model for programming. Ideally you construct a SOAP message through a call to the constructor (new SOAPMessage), and you send this message of to the HTTPClient. After the call returns, you receive either a null pointer, or an answer.

The framework takes care of constructing a HTTPMessage out of your SOAP message, carrying it over the HTTP(S) intra/internet to your server, where it gets processed, and returns a message to you.

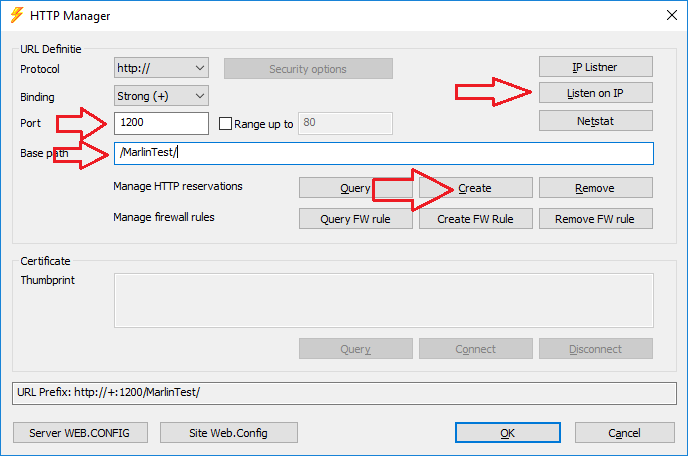
Note: Wherever you see the word ‘SOAP’ in this text and the figure below, you can also read ‘JSON’.



**Running the examples**

After downloading the sourcecode, the steps to get the examples running are the following:

1. Startup Visual Studio 2015 (minimum requirement);
2. Load the “HTTPManager” project and compile that one first, so we can set up a server on our machine. Recommended is to compile the “debug | x64” variant;
3. Start the HTTPManager and register the site “http://localhost:1200/MarlinTest/” (see the figure below how to do this);



After starting you should:

1. Press the “Listen on IP” button to make sure your machine will listen to IP traffic;
2. Fill in the number “1200” for a port number. All tests run on IP port 1200;
3. Fill in the base path “/MarlinTest”. All test run on subsites of this site;
4. Press the button “Create” to register the “http://+:1200/MarlinTest” prefix listener;
5. Close the HTTPManager program and go back to Visual Studio;
6. Load the “MarlinServer” project and compile it;
7. Start a Second Visual Studio and load and compile the “MarlinClient” project;
8. Now start the MarlinServer first and check that all sites have been registered;
9. Start the MarlinClient now and if all tests are satisfactory it will end with a “Yipee!!” ☺

**History of the Marlin server**

Long ago I needed a webserver that was capable of running on Citrix desktops. With the extra require­ment that no administrator should be bothered to grant me the rights to run a webserver like IIS or apache for instance. What I wanted was a webserver I could ‘smuggle’ on board in a corner of a larger application. So that the applications could communicate with each other on a relatively free and with zero administrative overhead.

After a long search I settled with the “Universal TCP Socket” program from Elmü that I found on Codeproject (<http://www.codeproject.com/Articles/34163/A-Universal-TCP-Socket-Class-for-Non-blocking-Serv>). It is a very nice product and along with other things has asyn­chro­nous handling of incoming TCP/IP traffic. The only thing needed was the implementation of the full HTTP protocol. Which I then did. And it worked out well for quite some time.

But times changed. Could I not take care of the wish to be able to also run on the internet, instead of the intranet alone? And please could you then see to it that the server understands regular SSL/TLS and authentication methods?

Well, what righteous programmer wouldn’t? I imagined myself already exploring the OpenSSL libraries and all sorts of cryptographic methods, but hey… wait.. Why bother if Microsoft hadn’t taken care of that already? After a thorough bing/google search I found and settled for the “HTTP Server API” from Microsoft. (<https://msdn.microsoft.com/en-us/library/windows/desktop/aa364510(v=vs.85).aspx)> It’s incorporated for free in any MS-Windows installation and after Windows-Vista / Server 2008 the version 2.0 of this library also takes care of SSL/TLS and authentication. So I ported the central classes of this library from the “Universal TCP Socket” to the “HTTP Server API 2.0”, and it became the HTTPServer in this program.

Well, to be exact: I hesitated for a while. Why not use an existing framework like gSOAP (<http://www.cs.fsu.edu/~engelen/soap.html> ) or the new and promising “Casablanca” framework from Microsoft (<https://casablanca.codeplex.com>)? Both are on a slightly higher abstraction level and provide all needed mechanisms. But after experimenting with both, I decided to stick with the current programming model. gSoap proved to be quite arcane and Casablanca a moving target before it moved to codeplex.

Communication with other programs – mainly in the .NET stack – were required over the years. Some of these programs used the ‘reliable-messaging-interface’. So that was added in an extra abstraction layer in “WebServiceServer” and “WebServiceClient”. Because the Microsoft .NET stack uses WSDL 1.1 and not 2.0 like the Java stack, I used that version (1.1) for the support of WSDL (Sorry Java guys).

From the beginning on I had written the HTTPMessage and the SOAPMessage to abstract the sending and receiving of messages. HTTPMessage took care of files (put and get) and the SOAPMessage took care of SOAP XML messages. All sites were sitting in a mapping of the server. After some time the server spread to more and more programs and it soon became clear that an extra layer of abstraction was needed. This became the HTTPSite object. Much of the settings and attributes of the server moved to the site objects, so that different sites could have different settings.

This web server is currently in its third incarnation. The version history is included in the sourcecode in the documentation directory. As is this document and the architecture overview in a Visio drawing.

**Why not using IIS and ISAPI?**

But why not using IIS for those applications where we can clearly administer them the correct way, without having to ‘smuggle’ them in? The ISAPI programming model for native appli­cations in C or C++ have some clear restrictions, stemming from the IIS administrative model. These restrictions are exposed below:

1. Originally the ISAPI model was very simple, with just one module handler, and the possibility to ‘read’ the http incoming traffic and to ‘write’ the answer back. This model is luckily expanded at the emergence of version 7, but still is very restricted. One of the better features of the HTTP Server API is the possibility to choose the way of writing back: in one go, per chunk or just by pointing to a filename to be written without even opening and reading / writing the file;
2. The ISAPI model does not guarantee that we end up in the same process or even on the same physical machine with our call. This makes it necessary to save the complete session state of a session to an external machine or a database, re-reading the session state upon each call. This slows down the server processing for session based appli­ca­tions. Certainly those applications that live on lots and lots of short calls.
3. Using ISAPI means that you have to scan each request. When combining multiple applications on a machine, the performance of the applications can serious degrade as all the applications will have to scan an average of ½ \* (sum of all requests of all applications). This leads to the installation instruction to install each and every one of the applications on it’s own machine. And thus heightening the total cost of ownership (TOC);
4. The ISAPI extension module interface was designed to be, well…, an interface for just *extensions*. Not the real thing. You can write extensions to interfaces or websites, but if you want it to be the thing itself, you’re in for a hard time.

**APPENDIX 1: BASE CONFIGURATION**

**The base configuration**

of the Marlin client/server is done by using the standard MS-Windows tools. This is done because the configuration takes place in the MS-Windows kernel. At a minimum you must take the following steps:

1. Register a machine to be listened to. You must login ‘as administrator’ and give the following command:

netsh http add iplisten ipaddress=0.0.0.0

Instead of the 0.0.0.0 address, you can also use the ‘real IP address of your machine (or that of another machine in a cluster farm)

1. Register an URL to be listened to, and add the specific rights for users who can listen to this URL. You must login ‘as administrator’ and give the following command:

netsh http add urlacl url=http://+:1200/MarlinTest/ user=\Everyone

This configures the URL for specific users. This is an equivalent of the command that works on multilingual versions of the MS-Windows kernel:

netsh http add urlacl url=http://+:1200/MarlinTest/ user=D:(A;;GX;;;WD)

The nasty thing here is that changing one of the elements of the URL like:

- http to https

- strong to weak address binding

- servicename

You will have to remove the registration with “delete urlacl” and register the changed URL anew.

1. If the registered URL is a secure service (HTTPS) you will need to install a certificate in your certificate store (with “mmc.exe”) and configure this certificate to be used with the URL. These are the steps.

1) Start "mmc" as administrator

2) Add the module "Certificates" and start the “Machine” configuration!!

3) Add your certificate in the “Trusted Publisher” store

4) Find the thumbprint of the certificate and remove the spaces

5) Use the thumbprint of the certificate to add to the HTTP service with:

netsh http add sslcert ipport=123.321.12.12:443 certhash=0102030405060708090A0B0C0D0E0F1011121314

appid={00112233-4455-6677-8899-AABBCCDDEEFF}

The parameter certhash is the thumbprint of the certificate just added to the store. The app-id is irrelevant fort his solution (only used for WCF services of .NET), but the app-id is a minimum requirement of the command.

1. Create a "web.config" file after the example of the "web.config.txt". See for documentation in the "WebConfig" class file.

These configuration steps are gathered in the “HTTPManager” program. This program automates all the steps above AND writes a “web.config” file for you.

**APPENDIX 2: SUPPORTED FEATURES**

The following features are being supported:

The server side

The Marlin HTTPServer uses the "WinHTTP API 2.0 protocol". Features are:

- SOAP webservices message traffic

- JSON dataservices message traffic

- HTTP message traffic

- REST services

- XML support for SOAP webservice messages

- Full cookie support

- Synchronous as well as a-synchronous message traffic

- SSL/TLS connections with server certificates

- Full support for SSE (Server Sent Events)

- Full IPv4 and IPv6 support for HTTP sessions

- Support for UTF-8 / UTF-16 in combination with MBCS (multi-byte character-sets) apps

- Support for Terminal-services and remote-desktop protocols (e.g. for Citrix/PowerFuse)

- Configuration files (web.config, supported through HTTPManager.exe)

- Logging (configurable + caching in asynchronous overlapping I/O)

- Multi-codepage support for character sets. All (!!) code pages including UTF-16.

- Thread pooling + throttling

- Error support (OS errors, SOAP fault stacks and Safe exception handlers)

- Support for ANSI & multi-byte-character set compilation

- Full use of the Kernel HTTP, to support connections to IIS and .NET WCF services

- Reliable-messaging according to the W3C 2005/02 protocol

- Reliable-messaging is WCF aware. Detecting of the Microsoft "netrm" namespace optimizations supported

- Reliable-messaging is compatible with WCF, it generates he nonce's ("urn:uuid:<GUID>")

- WS-Security standard for the signing of the body, encryption of the body and of the complete soap message

- Not dependent on an active/not active IIS. Can be used in a Terminal Services client environment!

- Authentication caching + token/ACL support for impersonation

- Authentication by the following methods: basic, NTLM, negotiate, digest and Kerberos

- Kernel support for URL authentication

- Extra performance by using the configurable MS-Windows kernel parameters

- Kernel configuration through the "netsh / netconfig" tools

- All sessions are made unique as the "user/ip-address/desktop" combination

- Protected by Structured Exception Handlers (SEH) against crashes of individual threads

- Crash reporting with full error stack listing and loaded external modules (DLL’s)

- Application parameter overrides through general web.config and web.config per site

WebServiceServer uses the HTTPServer to support webservices at the server side. Extra features are:

- Data contracts through registering of all SOAP messages.

- Generation of a WSDL 1.1 file for other development environments to read and base there services on.

- Optional checking of incoming and outgoing messages against the data contract.

- Very simple implementations are supported by deriving a class from the WebService­Server and implementing the service dispatcher through the WEBSERVICE\_MAP structure à la MFC.

- Generates the service pages for SOAP 1.1 and SOAP 1.2 examples and distributes the WSDL contract file to external parties through HTTP-GET services

- Checking of in- and outgoing services against the WSDL

HTTPManager to manage URL’s and rights.

- This program simplifies the management and authentication of URL’s ports and certificates. The use of kernel scripts through “netsh” is no longer needed at the bare metal level, but you can still do that of course.

- This program contains an editor for the "web.config" files with overrides of the installation parameters.

The client side

HTTPClient uses the "WinHTTP 5.1 Client protocol". Features are:

- SOAP message traffic (SOAP 1.0 / 1.1 / 1.2)

- HTTP message traffic and REST service support

- XML support for SOAP

- SSL/TLS connections with certificates

- Support for UTF-8 / UTF-16 in combination with MBCS (multi-byte character-sets) apps

- Full cookie support

- Full SOAP 1.0 HTTP header support

- Synchronous and asynchronous messages

- Agent configuration against client 'spoofing'

- EventSource component to support SSE (Server-Sent-Events)

- Proxy configuration including proxy-authentication

- Proxy bypass lists to rule out sets of proxies

- Full authentication support including "Windows LanManager single-sign-on"

- Multi-codepage support. All (!!) code pages including UTF-16.

- Configuration through web.config and through the program

- Test configuration for self-signed certificates (the so-called ‘relax options’)

- WS-ReliableMessaging support (2005/02)

- WS-Security support (2004/02)

- TerminalServices support for Citrix

- Retry support for weak TCP/IP environments

WebServiceClient implements the use of webservices at the client side. Features are:

- Simple creation and calling of the URL channel

- Full HTTP handling including SOAP faults

- Processes all WSDL 1.1 calls

- Processes all WS-Reliable and WS-Security calls

- Checking of in- and outgoing services against the WSDL

General features of all components

- Small footprint (less than 500 K)

- High performance through direct compilation in C++ against the MS-Windows kernel

- Standard protocols (W3C / IETF)

- Fully REST compatible

- Fully transparent use of SSL/TLS and http-authentication

External requirements for the use of the components

- Compilation minimal on Visual Studio 2013.

- HTTPServer only runs on MS-Windows versions "Vista" and higher. Full support only from Windows 2008 R2 and up.

**APPENDIX 3: RELEVANT DOCUMENTATION AND OPEN STANDARDS**

This is a list of links to the used and relevant open standards, and relevant documentation on the Microsoft MSDN site, the W3C site and the IETF site:

HTTP Server API Version 2.0 Reference

<https://msdn.microsoft.com/en-us/library/windows/desktop/aa364634(v=vs.85).aspx>

Using the HTTP Server Version 2.0 API

<https://msdn.microsoft.com/en-us/library/windows/desktop/aa364703(v=vs.85).aspx>

HTTP Client API

<https://msdn.microsoft.com/en-us/library/windows/desktop/aa384252(v=vs.85).aspx>

HTTP Protocol

<https://www.ietf.org/rfc/rfc2616.txt> (the original RFC)

<https://tools.ietf.org/html/rfc7230> (Message syntax and routing)

<https://tools.ietf.org/html/rfc7231> (Symantics and content)

<https://tools.ietf.org/html/rfc7232> (Conditional requests)

<https://tools.ietf.org/html/rfc7233> (Range requests)

<https://tools.ietf.org/html/rfc7234> (Caching)

<https://tools.ietf.org/html/rfc7235> (Authentication)

<https://tools.ietf.org/html/rfc7236> (Authentication schemes)

<https://tools.ietf.org/html/rfc7237> (Method registrations)

HTTP Basic and digest authentication

<https://www.ietf.org/rfc/rfc2617.txt>

HTTP State management (cookies)

<https://www.ietf.org/rfc/rfc2965.txt>

Server-Sent-Events

<http://www.w3.org/TR/eventsource/>

WS-Addressing

<http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509>

WS-MessageSecurity

<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0>

WS-ReliableMessaging

<http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf>

XML-Standards (Primers only)

<http://www.w3.org/TR/2004/REC-xml-20040204> (version 1.0)

<http://www.w3.org/TR/2004/REC-xml11-20040204/> (version 1.1)

SOAP Standards (Primers only)

<http://www.w3.org/TR/2000/NOTE-SOAP-20000508> (version 1.1)

<http://www.w3.org/TR/2007/REC-soap12-part0-20070427> (version 1.2)

JSON Standard

<http://json.org>

<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>