

Introduction to the Barracuda Embedded Web-Server

This paper covers fundamental concepts of HTTP and how the Barracuda Embedded Web Server can be used in an embedded device.

[Introduction to HTTP](#)
[Using a Web-Server in an embedded device](#)
[Using a Web-Server for regression tests](#)
[Using a browser to control an embedded device](#)
[Introduction to Server Side Scripting](#)
[Creating dynamic user interfaces using CSP](#)
[Sending data from a browser to the server](#)
[Hacking a Web Server using telnet](#)
[Rich Client Interface](#)
[The EventHandler](#)

Depending on your knowledge of the various technologies covered, it may not be necessary to read each section, although each section also presents some of Barracuda's unique features. This paper also assumes a basic understanding of HTML, C and C++.

This article is a PDF version of our online whitepaper at:

<http://barracudaserver.com/WP/intro/>

Please note that the PDF version is for printing a hardcopy. Please see the online version for the full whitepaper and the interactive examples.

Introduction to HTTP

The Hyper Text Transfer Protocol(HTTP) is a text-based Remote Procedure Call (RPC) protocol that can transfer any type of data between the client and server. An HTTP client opens a connection and sends a request message to an HTTP server. The server then returns a response message.

The HTTP header contains an initial line and a number of header lines followed by an optional body. The initial line of the request and the response are different. In a request, the initial line contains the HTTP method and the requested resource, while in a response, it contains the response protocol version, a status code, and a status message. The most common HTTP methods, the RPC type, are GET and POST, where GET usually means "Server, please give me this resource" and POST usually means "Server, here is my data", though this should be decided by the resource.

Example HTTP command, setting the refrigerator and freezer temp:

```
POST https://embeddedwebserver.net/refrigerator/temperature/ HTTP/1.1
User-Agent: I-typed-this-connecting-a-telnet-client-to-the-server
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

fridgeTemp=5&freezerTemp=-30
```

The Barracuda embedded Web Server treats all resources as executing units. A Barracuda executing unit can be a physical resource such as a page. A virtual resource can also be a container that can contain other executing units. An executing unit is just a plugin to the web-server. The Barracuda embedded web-server comes with a number of executing units such as a directory unit that can directly read from a ZIP file and present the content to a browser client. Other plugins, such as the EventHandler, can be installed as well. The above HTTP command example sends two parameters, fridgeTemp and freezerTemp to an executing unit in the Barracuda Virtual File System at location "/refrigerator/temperature/". A Directory Executing Unit is called HttpDir and a Page Executing Unit is called HttpPage. An HttpPage and HttpDir can be extended, or in Java terminology, you can extend the base classes and implement your own "live resource".

Using a Web Server

in an embedded device

A Web Server in one device is typically used for remote management of one or more devices. A client does not necessarily have to be a browser. Any client implementing the HTTP protocol stack can be used to control the device. The HTTP protocol is good for sending anything to the device, from control data to exchanging user data, and even uploading new software releases. A client HTTP library can be linked into a C or C++ client application or a scripting with native HTTP support, such as Python, can be used.

There are many reasons for using HTTP as a general protocol between a client and a server, the device. An application using the HTTP protocol benefits from services provided by HTTP such as:

- authentication and authorization support
- Encryption by using SSL
- Bypass firewall restrictions

A resource is addressed by its URL, or the path element of the URL. For example, we used the path "refrigerator/temperature/" above to address the temperature object in the server. One can think of a resource as an object with a number of methods. The methods can be the standard HTTP methods such as GET, POST, PUT etc, but HTTP is very flexible and you have no such constraints when designing a Barracuda executing unit. You are free to interpret the HTTP methods to anything you like as long as your client and server understand the data being exchanged.

Using a Web Server for regression tests

The HTTP protocol is becoming increasingly popular for exchanging data. You can even find Open Source HTTP client libraries that can be link with C or C++ host applications. Most scripting languages also come with native HTTP support. Python is one such scripting language. During development, internal objects and methods in your embedded system can be exposed using a Barracuda Executing Unit. Each Executing Unit is installed as a resource in the Barracuda Virtual File system, thus each object you want to expose can have its own URL.

A scripting language such as python simplifies writing advanced regression tests. Data sent to the server can be encoded as "application/x-www-form-urlencoded" data as shown above. URL encoded data, is so common in HTTP that the Barracuda Web Server automatically decodes all of the data in the URL encoded stream. URL encoded data in HTTP is much like the parameters passed to a C function.

However, exchanging complex data structures is difficult with URL-encoded data, so for these some serialization layer must be used. Fortunately, there are platform and language independent standards available that do this. SOAP is one method used to serialize an object into XML. Barracuda supports a lighter and more common XML protocol called XML-RPC.

Many scripting languages natively support XML-RPC. Python, for example, can utilize any object in the embedded device exposed as an XML-RPC object from a script running on a host computer. Another common regression test environment is Expect, based on TCL, which also has support for XML-RPC.

Using a browser to control an embedded device

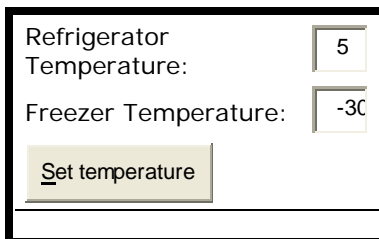
Most people associate a Web Server as a way of serving HTML content to a browser. An advanced Web Server such as Barracuda can do much more than serving HTML files. We briefly discussed above how one can use Barracuda as a general purpose HTTP stack and how one can do regression tests by exporting internal objects in a device to script programs running on a host computer.

A naive Web Server treats resources as files and either has no support for dynamically generated content or forces the Web-Designer to use certain predefined directories for generating dynamic content. For example, a server supporting CGI forces the web-designer to use the /cgi/ directory for dynamic content generation and data exchange.

Barracuda supports Executing Units, thus a resource is a "live resource" i.e. it is a mini program one can plug into the server. In Barracuda, a resource is typically derived from the HttpPage type. A more complex design can use the HttpDir type to design a collection of virtual resources. A Barracuda resource can dynamically generate anything from binary data such as streaming audio to plain text. A resource is typically generating HTML when the client is a browser.

A typical scenario when using a browser to control a device is for the browser to initially request a certain resource in the server. The resource might respond by sending back an HTML page containing information about that particular resource when receiving an HTTP GET command. For example, a resource at "refrigerator/temperature/" sends back information about the current temperature in the refrigerator and freezer. The HTML sent to the browser can for example contain an HTML form, which the user can use for setting a new temperature in the device.

HTML form example



The screenshot shows a web form with two text input fields. The first field is labeled "Refrigerator Temperature:" and contains the number "5". The second field is labeled "Freezer Temperature:" and contains "-30". Below these fields is a "Set temperature" button.

The HTML form source code

```
<form method="post">
  <table>
    <tr>
      <td>Refrigerator Temperature:</td>
      <td><input type="text" name="fridgeTemp"
value="5"/></td>
    </tr>
    <tr>
      <td>Freezer Temperature:</td>
      <td><input type="text" name="freezerTemp"
value="-30"/></td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="Submit" value="Set
temperature"/>
      </td>
    </tr>
  </table>
</form>
```

If the user wants to change the temperature, the user can use the form displayed in the browser to change the values and then press the submit button. When the user presses the submit button, the browser collects the data in the form and sends it as URL-encoded data to the server. The Barracuda server decodes the URL-encoded data and starts the resource specified in the POST command. In Barracuda, this will typically be the same resource as the resource that produced the initial HTML page. A naive web-server forces the web-developer to POST the data to a predefined fixed URL, for example, a CGI Web Server forces the web-developer to POST data to the "cgi/" directory.

Introduction to CSP

A Barracuda resource is typically derived from the `HttpPage` type. When the user requests a resource, the Barracuda Web Server locates the resource and delegates the request to the resource. A resource gets two objects from the server, a request object and a response object. The request object contains information about the client and any data sent from the client. The response object is used when dynamically creating the response data, such as an HTML response.

The following shows a code fragment from a resource object serving a HTTP GET request.

```
response->write("<html>");
response->write(" <body>");
response->write(" <p>Greetings</p>");
response->write(" </body>");
response->write("</html>");
```

As you can see from the above fragment, the resource object creates a simple HTML page containing a greeting message.

Most Web-Developers want to separate content from logic. The above `HttpPage` resource code example shows a tight integration of code and HTML content. Maintaining such a resource is difficult and time consuming. The `HttpPage` resource makes it possible to do advanced HTTP communication. A way of separating logic and content, while giving developers the flexibility of the services provided by a `HttpPage` resource is needed.

The Barracuda platform solves this problem by providing a special TAG compiler that takes an HTML file and translates the file to an `HttpPage`. The TAG language is called CSP. CSP, short for C Server Pages, is a modern reimplementaion of the CGI standard. CSP extends HTML with new tags that are only visible on the server side.

The following shows a very simple CSP page.

```
<html>
  <body>
    <p>Greetings</p>
  </body>
</html>
```

The above example shows that a CSP page looks like a regular HTML file. This page contains no special CSP tags and behaves like static HTML file.

CSP also makes it easy to create dynamic pages. A dynamic page is a resource that creates content either from user input or from some information in the device.

Let us now change the above simple CSP page to the following:

```
<html>
  <body>
    <p><%= "Greetings" %></p>
  </body>
</html>
```

CSP tags start with <% and end with %>. The <%= tag tells the CSP runtime to take the following string and send it to the client that requested the resource -- i.e. to the browser.

We have now added a CSP tag to the page, but the result produced is identical to our first CSP page. The string "Greetings" is constant and will never change.

Continuing with our CSP example page:

```
<html>
  <body>
    <p>Greetings user of <%=request->getHeaderValue("User-Agent")%></P>
  </body>
</html>
```

Please see online version for demo program.

Now we added some C++ code directly into the HTML page. The method `getHeaderValue` in the request object returns the value for the "User-Agent" HTTP request header value. The Barracuda server decodes all the header values and keeps them in an internal table, thus `getHeaderValue("User-Agent")` returns the identity of the client doing the request.

Barracuda is a C library written in ANSI C compatible code, but provides both a C++ and a C interface. The above C++ code can be written as the following C code

```
<%=HttpRequest.getHeaderValue(request, "User-Agent")%>
```

The Barracuda framework provides a rich API to the developer. A CSP resource can use all of the C or C++ methods provided by this API. A CSP resource has two implicit objects available when executing: the request and response object. The request object provides information sent from the client to the server, while the response object is used when sending data back to the client. The response object is implicitly used in the above code by the <%= tag.

Creating dynamic user interfaces using CSP.

The first phase of developing a CSP application is creating the HTML interface, and is usually handled by a professional web interface developer. The second phase involves a C programmer or web developer with a small amount of C experience to add the CSP tags to the HTML code.

The final step is testing the new interface in the Barracuda Embedded Web Server. This can be an iterative process, and will show places where adjustments should be made to the HTML or other interface components. HTML editors such as Dreamweaver will not corrupt or remove the CSP tags with the C/C++ code.

The following examples are simple so that they can easily be understood. In most web applications, the HTML would be much more extensive, and would also likely involve CSS.

First, we'll add an interface to display the temperature reported by a new controller unit on our refrigerator.

```
<html>
  <body>
    <h1>My refrigerator</h1>
    <table>
      <tr>
        <td>Refrigerator Temperature:</td>
        <td><%= "%d" rand()%10 %></td>
      </tr>
      <tr>
        <td>Freezer Temperature:</td>
        <td><%= "%d" - rand()%35 %></td>
      </tr>
    </table>
  </body>
</html>
```

Please see online version for demo program.

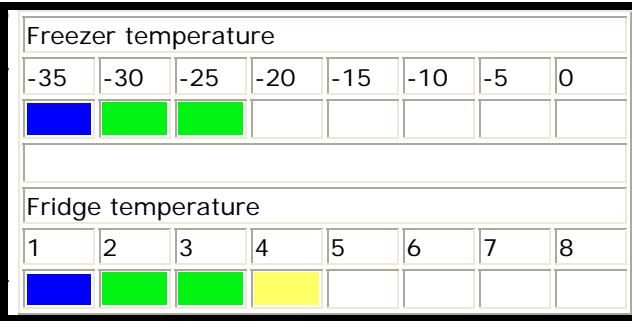
The <%= tag by default assumes that the data is a string. All other types must be explicitly declared. The <%= tag accepts the same format flags as printf, so <%= "%d" means "print an integer value". The above code uses the ANSI C function rand to generate a random temperature value. A real implementation would call a function that returns the actual temperature. The modulus operator is used such that we get a refrigerator temperature between 0 and 10 degrees Celsius, and a freezer temperature between 0 and -35 degrees Celsius.

Most of the content in this example page is static. This is typical when working with CSP. The HTML framework is always the same, but the temperature data is dynamically presented to the user.

An improvement to this, which just prints out the temperature of the freezer and refrigerator in an HTML table, would be to present the temperature graphically as a bar chart.

There are a number of ways to create a bar chart using HTML. This example will dynamically create a HTML table containing small images.

The figure to the right shows a bar chart representation of a freezer temperature of -25 degrees Celsius and a refrigerator temperature of 4 degrees Celsius.



The colors blue, green, yellow, orange and red are used. Also, a transparent image is used when hiding the temperature element. The above freezer bar contains 5 transparent images for the temperature -20, -15, -5 and 0.

To keep the example as simple as possible, the refrigerator temperature is not included in our CSP example. The table in the CSP code below contains three table row elements (<tr>). The first row contains a table data element, which spans across 8 elements, the second row contains the temperatures, and the last table row contains the images. The getFreezerImg function returns the string for the current image to insert into the table element.

```

<html>
  <body>
    <h1>My refrigerator</h1>
    <table>
      <tr>
        <td colspan="8">Freezer temperature</td>
      </tr>
      <tr>
        <% /* print out the temperatures -35, -30 etc in the first table row */
          for(i = -35 ; i <= 0; i+=5)
            response->printf("<td>%d</td>", i);
        %>
      </tr>
      <tr>
        <% /* Print out the images in the second table row */
          for(i = -35 ; i <= 0; i+=5)
            response->printf("<td><img src=\"%s.gif\"/></td>",
              getFreezerImg(freezerTemp, i));
        %>
      </tr>
    </table>
  </body>
</html>
<%p
  int i;
  int freezerTemp = - rand()%35;
%>
<%g
#include <stdlib.h>
const char* getFreezerImg(int freezerTemp, int i)
{
  if(freezerTemp < i)
    return "transparent";
  switch(i)
  {
    case -35: return "blue";

    case -30:
    case -25: return "green";

    case -20:
    case -15: return "yellow";

    case -10:
    case -5: return "orange";

    case 0: return "red";

  }
  assert(0); /* Should not get here */
  return "";
}
%>

```

Please see online version for demo program.

The above example demonstrated several new tags.

The `<%g` tag means global. Anything declared here goes outside of the generated `HttpPage`. The `getFreezerImg` function is inserted into the global area by the CSP compiler.

The `<%p` is the prolog tag. Anything declared here is inserted at the beginning of the `HttpPage` service function. The `<%p` is a special form of a code fragment tag.

The `<%` is a code fragment tag. Anything in a code fragment tag is executed when the `HttpPage` service function runs. For example, the following code fragment tag from the example above prints out table data elements containing the images.

```
<%
  for(i = -35 ; i <= 0; i+=5)
    response->printf("<td><img src=\"%s.gif\"/></td>",
                    getFreezerImg(freezerTemp, i));
%>
```

The following will be printed out by the above code fragment if the freezer temperature is -25 degrees:

```
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
```

We use the `printf` function in the response object to print out the table data elements. Another possibility is shown below.

```
<% //Begin loop
  for(i = -35 ; i <= 0; i+=5) {
%>
  <td></td>
%>
  } //End loop
%>
```

This produces the same output as the original code fragment.

Sending data from a browser to the server

A client application does not necessarily have to be a web-browser. When a browser sends data to a server, the content type of the data normally is "application/x-www-form-urlencoded". Browsers don't normally send other content types without executing JavaScript code.

URL-encoded data is sent as key/value pairs as "key1=value1&key2=value2&key3=value3".

Some characters must also be "quoted" to prevent them from being handled as control characters. This "quoting" is transparent to the CSP designer when using the Barracuda framework.

Sending data from a browser to the server normally works like this:

- The browser sends a GET request to a resource in the web-server.
- The web-server delegates the request to the resource.
- The resource responds by sending an HTML page containing one or a number of HTML forms to the client.
- The user fills in the form and submits the form to the web-server.
- The browser collects the form data and sends the information as URL-encoded data to the resource.
- The web-server delegates the request to the resource, and this time the resource detects that this is a POST command. The resource fetches the pre-parsed URL-encoded data from the request object by using methods in the request object.

The following code will make the refrigerator resource also able to change the temperature settings.

```

<html>
  <body>
    <h1>My refrigerator</h1>
    <form method="post">
      <table>
        <tr>
          <td>Refrigerator Temperature:</td>
          <td><input type="text" name="fridgeTemp" value="<%= "%d"
fridgeTemp%>" /></td>
        </tr>
        <tr>
          <td>Freezer Temperature:</td>
          <td><input type="text" name="freezerTemp" value="<%= "%d"
freezerTemp%>" /></td>
        </tr>
        <tr>
          <td colspan="2"><input type="Submit" value="Set temperature" /></td>
        </tr>
      </table>
    </form>
  </body>
</html>
<%!
  // Declare two variables in the HttpPage object
  int fridgeTemp;
  int freezerTemp;
%>
<%!!
  // Initialize the two variables at system startup.
  fridgeTemp = 5; // Default refrigerator temp
  freezerTemp = -30; // Default freezer temp
%>
<%p
  if(request->getMethodType() == HttpMethod_Post)
  { // The following code is executed if the user press the submit button
    int newFridgeTemp = (int)U32_atoi(request->getParameter("fridgeTemp"));
    int newFreezerTemp = (int)U32_atoi(request->getParameter("freezerTemp"));
    // Are the new values within the tolerated limits.
    if(newFridgeTemp < 1 || newFridgeTemp > 7 ||
       newFreezerTemp < -35 || newFreezerTemp > -18)
    { /* No, the user obviously does not know enough about food
       safety. Send user to a page that can educate the user
       on food safety.
       */
      response->sendRedirect( /* Send HTTP code 302 to browser */
        "http://lancaster.unl.edu/food/ciqxx.htm");
      return; // Abort
    }
    //Set the new temperature
    fridgeTemp = newFridgeTemp;
    freezerTemp = newFreezerTemp;
  }
%>

```

Please see online version for demo program.

If you look at the HTML section, you will see that we have now added a HTML form element to the page. The original table data elements printing out the temperature are now changed to a table data element containing a HTML form input type.

```
<td><input type="text" name="fridgeTemp" value="<%= "%d" fridgeTemp%"/></td>
```

The default value for fridgeTemp is 5 degrees, so what is initially sent to the browser is:

```
<td><input type="text" name="fridgeTemp" value="5"/></td>
```

When the user clicks the submit button, the browser sends the data to the web-server. The initial HTTP POST command in the HTTP section above shows what this form data looks like during transmission. The URL-encoded data "fridgeTemp=5&freezerTemp=-30" corresponds to the two field names in the CSP example.

The CSP code extracts the fridgeTemp and freezerTemp in the prolog section, i.e., the tag starting with <%p.

```
request->getParameter("fridgeTemp")  
request->getParameter("freezerTemp")
```

The code example above uses the keys "fridgeTemp" and "freezerTemp" in order to fetch the value part of the data.

The server will perform a sanity check on the data. This is an important step and is typically where web applications fail. If the sanity check fails, the user is forwarded to a page explaining food safety. This is something we added for fun. A web application would normally respond with an error page.

Another implicit sanity check is performed by the U32_atoi function. The U32 is an unsigned 32 bit type in the Barracuda web-server. The U32_atoi function works like the standard atoi function, but it also can tolerate a NULL string without crashing. Even though the U32 function returns an unsigned integer, the function can decode and negate negative numbers.

```
//The following returns NULL if "fridgeTemp" is not in the URL-encoded data.  
request->getParameter("fridgeTemp")
```

Hackers are constantly trying to penetrate and crash web servers by using a number of methods, and they aren't connecting to web servers with browsers for these attacks. They won't always be sending proper form data, so the server must check that all required fields are present.

"Hacking" a Web Server using telnet

As a simple example, open a command window (DOS window). In windows 2000 and XP a command window can be started by pressing the start button and thereafter selecting run. Type cmd in the pop-up window and press return. Copy the text below by marking it and press control-C.

```
telnet embeddedwebserver.net 80
POST /refrigerator/temperature/ Http/1.0
User-Agent: I-typed-this-connecting-a-telnet-client-to-the-server
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

fridgeTemp=2&freezerTemp=-25
```

Now paste the text into the DOS window. Text can be pasted into a DOS window by pressing the top left icon in the DOS window, the small "C:\\" icon. Scroll down to edit and select paste. The resource should respond with a message telling you that you successfully set the refrigerator temperature to 2 degrees and the freezer temperature to -25 degrees Celsius. You can also [download](#) two DOS batch files, which automatically run the telnet commands.

Now do the same thing, but without the URL-encoded data.

```
telnet embeddedwebserver.net 80
POST /refrigerator/temperature/ Http/1.0
User-Agent: I-typed-this-connecting-a-telnet-client-to-the-server
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
```

Copy the above text and paste it into the same DOS window, and observe the different response message.

The path in the HTTP POST example requests a directory since the path ends with "/". The web server automatically adds "index.html" to the end of all directory requests. If you change the path to "/refrigerator/temperature/index.html" in the above examples, the results will be the same.

Barracuda supports request delegation, which is typically used in Model View Controller design. We use request delegation to construct a simple filter in index.html. The actual temperature resource is in "ex4.html". The index.html CSP resource is just a filter that validates a POST request and responds by sending a different message to the client if the Content-Length is 0.

The index.html resource delegates (forwards) the request to the ex4.html resource if the Content-Length is valid. This can be seen by changing the path in the first (the valid) telnet example above from "/refrigerator/temperature/" to "/refrigerator/temperature/ex4.html". You should get the same response message from the server.

Rich Client Interface

There are a lot of buzzwords on the internet these days about Rich Client applications, like "Rich Client Interface", "DHTML client", "Web Services", "Ajax", etc. A Rich Client application uses the browser as an environment to create a fully interactive graphical user interface rich and interactive user interface, and lies between a standard lightweight HTML form and a full native client-side OS-based application. A Rich Client can also be a standard Windows application that communicates with the server using HTTP, although such an application must be installed on the host computer before interacting with the server.

Modern browsers natively support a programming language called JavaScript. They also have a standard API, the Document Object Model, that JavaScript code can use for creating the user interface. This API allows the JavaScript code to create interactive GUI applications that resemble native applications. This basically means that you can design Dynamic Web-Applications using DHTML without the need of refreshing or reloading pages when you send data to the server.

A DHTML client communicates with the server using an object called XMLHttpRequest which, despite the name, has nothing to do with XML. The XMLHttpRequest allows JavaScript code to send any type of HTTP request to the server.

Below we have created a Rich Client Interface for the simple fridge controller we discussed above.

Please see online version for demo program.

The popular "Web Services" is basically a method of serializing and de-serializing objects over HTTP. Barracuda supports XML-RPC Web Services, even a standard Web Services client can be used when communicating with Barracuda. Typically, Web Services won't be used in a DHTML client when URL encoded data is sufficient. The original resource already accepts URL encoded data which is much easier to manage for most applications.

The resource "/refrigerator/temperature/ex4.html" is designed to accept URL encoded data. Our original resource wasn't configured to handle Web Services requests, so a new temperature object should be constructed specifically for Web-Services.

The response from the XMLHttpRequest object is sent asynchronously to a JavaScript callback function. The original service function responds by sending a HTML page to the client. A DHTML client could potentially parse the HTML to get the new temperature settings, but this requires a lot of work. A better solution is to tell the server resource that the client accepts JavaScript code as a response message by setting the correct "Accept" header in the request.

```
xmlhttp.setRequestHeader('Accept', 'JavaScript');
```

The server resource can simply check for this header.

```
if( strcmp("JavaScript", request->getHeaderValue("Accept")) == 0)
    Send JavaScript object
else
    Send HTML page
```

If the client sent the header, then a JavaScript object is sent as response message.

Ajax is an example of this, and can be found by searching on the internet. You can also take a look at Google Suggest, which is using an XMLHttpRequest object in the background as you type your search string.

The EventHandler

The above examples show how to design basic user interfaces using CSP technology and Rich Client Interfaces that can communicate with the server without refreshing the page.

Both the server generated CSP interface and the Rich Client interface must explicitly ask the server for a new temperature reading. A CSP-generated page could be sent to the browser that automatically refreshes the page after a number of seconds, and the Rich Client interface can have a JavaScript timer periodically call the XMLHttpRequest.

Polling the server for data adds overhead to the network traffic, and can even make the client interface behave strangely. It would be better if the server could inform the client asynchronously of changes to the temperatures without waiting for the client to ask again.

The Barracuda EventHandler plugin makes this possible as described in our EventHandler White Paper. This White Paper shows Rich Client Interface designs that aren't possible to implement using traditional RPC mechanisms.

Conclusion

We have covered a lot of ground in this paper, but we have barely scratched the surface of the endless possibilities with the Barracuda platform. The following White Papers are for users interested in a deeper understanding of the Barracuda platform.

<http://barracudaserver.com/WP/>

References

Please see online version at <http://barracudaserver.com/WP/intro/>