EDITOR'S REPORT

Human Interfaces for the Internet of Things

M2M Meets Web Applications Spawning the Internet of Things

The Internet of Things is growing rapidly. Managing the billions of devices is a challenge that must meet the expectations of users who have come to expect rich, graphical human interfaces via web browsers.

by Tom Williams, Editor-in-Chief

With all the talk, and with all the very real activity around the Internet of Things, we hear of numbers like 50 billion devices connected to the Internet. The possible applications are equally vast, targeting such things as efficient building control, industrial controls, military devices, medical instruments, smart consumer applications, transportation, environmental monitoring and more. A large portion of these 50 billion will be small and dedicated to a limited number of functions individually. Collectively, however, they will span huge applications such as those mentioned and generate vast amounts of data that are coming to be known as "Big Data." That Big Data eventually ends up on servers and server farms in the Cloud where it can be analyzed, combined and used for applications we may not have yet imagined.

What we are really getting with the Internet of Things is the foundation of what is coming to be known as Intelligent Systems, where devices communicate with each other mostly autonomously and yet their functions serve human ends, so human operators and consumers must interact with these systems in some manner. Since that interaction takes place via the Internet, it is natural that they are accessed through browsers. And increasingly, Internet access for things and people takes place with tablets and smartphones with their touch screen browsers. So how does all this work with a universe of small M2M devices that also must offer human access?

It should come as no surprise that humans require more resources to interact with devices and their applications than machines do when they simply communicate with one another. In other words, as Wilfred Nilsen, CEO of Real Time Logic, points out, to have meaningful interaction with an application, you need more than simple access to static pages, which is what you get with a simple web server. A simple web server, such as the well-known Apache, is really just an HTTP protocol stack that can access static, pre-defined web pages. But web servers like Apache can be enhanced with plug-ins and components to add functionality.

Getting to Rich Human-Machine Interfaces

Now of course most web sites provide more than just static pages, so there is some underlying application that dynamically creates pages in response to some user input. For embedded devices, that underlying application would mostly be some sort of control program that can execute input commands and return data about the status, etc. And of course, that added functionality requires resources in the form of processor power and memory.

Real Time Logic has concentrated its efforts at providing small, compact web servers for embedded devices with its Barracuda line. What it calls its Barracuda web server is actually more than just a simple HTTP protocol stack as described above, but it is aimed at requiring minimal resources. The Barracuda web server manages secure HTTP connections for Machine-to-Machine (M2M) communication and Human-to-Machine (H2M) interfaces. With C/C++ Server Pages (CSP) included, it delivers dynamic web applications, enabling live updating of secure data by authenticated connection (Figure 1).

The goal here is to provide access and interaction with a minimum footprint. The Barracuda web server is about 200 Kbytes and requires around another 60 Kbytes. The software development kit (SDK) provides a number of host tools that compile and link CSP files. The tools function similarly to a compiler or cross-compiler, and convert the C Server Pages files to either C or C++ code and to data files. A linker combines all the data files into one file, which is then embedded in the application. The produced C/C++

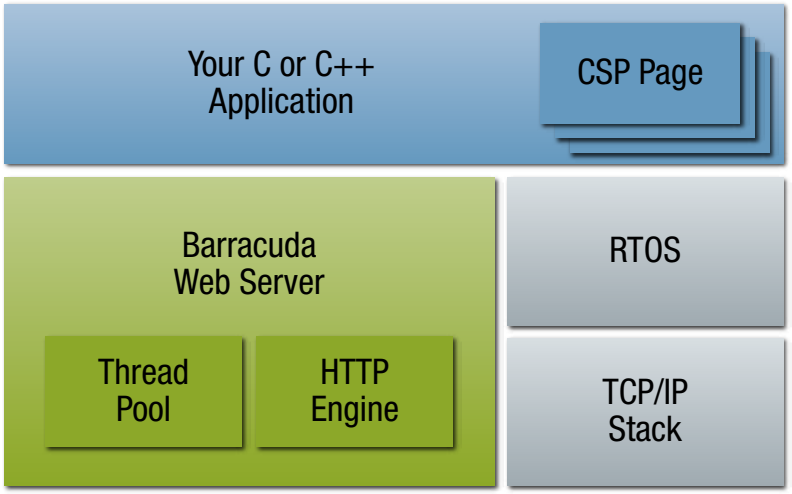

FIGURE 1

The Barracuda Web Server is an industrial-strength, small embeddable web server engine that is optimized for compact, deeply embedded devices.

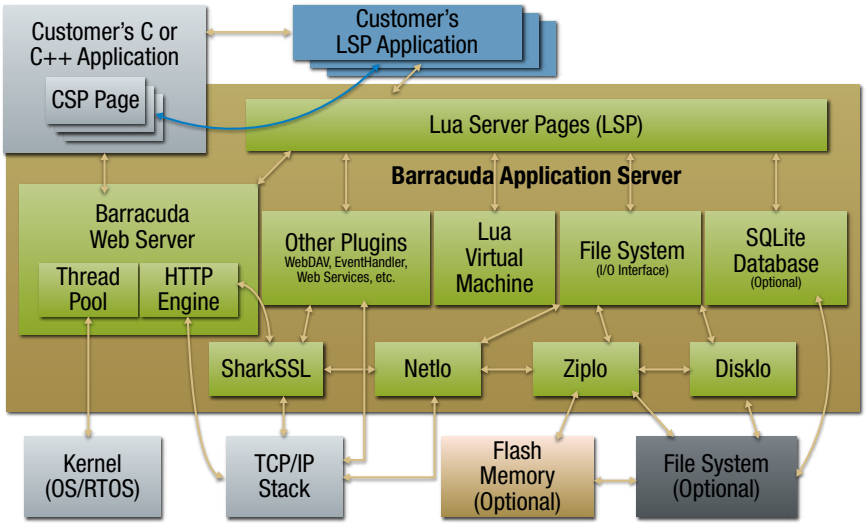

FIGURE 2

The full Barracuda Application Server is a C source library that lets designers access their own proprietary C functions from Lua code. It includes a rich selection of components and plug-ins.

code is compiled using your standard C/C++ (cross) compiler and the code is then linked with the application.

There are, of course, other ways to connect with embedded and networked systems from an IT environment. For example, there are dedicated programs that communicate with embedded devices using TCP/IP over the Internet because the developers may want interactive graphically rich user interfaces (e.g., dials, gauges, switches, displays) from a number of small remote devices. The trouble here is that such programs must be developed from scratch and cannot take advantage of many of the benefits of a web-based approach by piggybacking on the existing infrastructure. Of course, that in turn requires more resources on the embedded device to enable a rich human interface (Figure 2).

The Barracuda Application Server is an embeddable C source code library that builds on the web server to allow rich graphical applications and human-machine interfaces for interaction with embedded code. Since it is a C library, it can be compiled and linked to an embedded C application on the device. User interaction takes place via pages created with the Lua scripting language that interact with the C application via its API. These Lua server pages (LSPs) also interact with other components and plug-ins such as database, I/O and others.

According to Nilsen, the LSP pages are Lua code that are accessed via the HTTP engine, which will parse a request and then access that page. For example, after receiving "turn engine on" it will parse that page—the Lua code—on the fly and execute it using the Lua virtual machine, which interprets the page and can send a command to the C application and return the results to the user. The interfaces between the scripting code and the C side are "Lua bindings" that enable the scripting language to call functions in the C code.

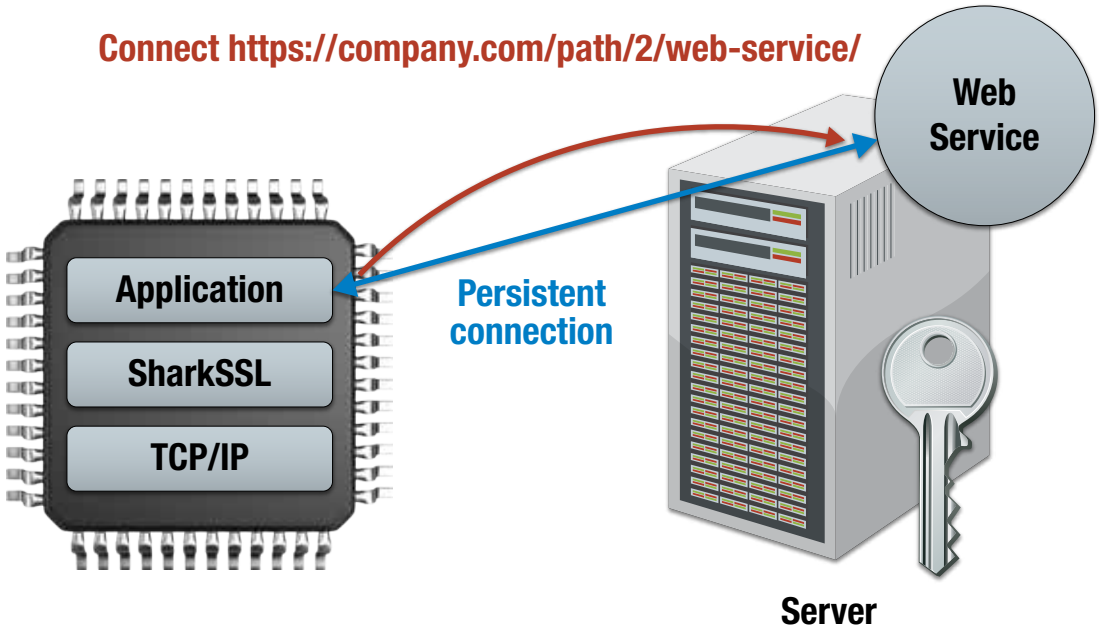

FIGURE 3

The RTL M2M approach requires minimum resources for a microcontroller to exchange data with other devices and with an application server.

Using a scripting language like Lua at this level, Nilsen notes, greatly simplifies development. Of course, the underlying application that will often sit on top of an RTOS such as Green Hills Integrity or Wind River's VxWorks will be written in C because of the need for detailed control of the underlying hardware. Once that is done, the use of a scripting language like Lua lets developers work at a higher level to make the application on the remote device easily available via a browser. And browsers themselves are relatively simple, so the device should be accessible from PCs, Macs and Android or iOS-based tablets and smartphones. That is what users increasingly expect and demand.

Rich Interaction with Tiny Devices

The options for developers, however, should not be a choice between a simple web interface on a resource-limited device or a rich interface on a larger, more powerful device. There are, after all, these millions of small devices that are collectively doing all this important stuff. We want rich interaction with them as well. For interaction with a microcontroller such as an Intel Atom or an ARM Cortex-A4, you certainly can't embed an application server, but you can rely on a combination of M2M communication among small devices and a small, dedicated server with the resources to run the Barracuda Application Server.

The classic M2M design uses standard SOAP/XML web services. But a SOAP stack with its XML parser is often too big for a microcontroller. Even the HTTP engine required by the web server may be too big for a microcontroller's internal memory. A microcontroller can communicate with a specialized online web service by using secure communication managed with a TCP/IP stack and a secure socket layer (SSL) client stack, in this case Real Time Logic's SharkSSL. The added benefit of the approach used by RTL is that the need for an HTTP protocol stack is eliminated because the device connects to the specialized web service by sending an initial HTTP header that is then morphed into a persistent socket connection as soon as the connection is established with the server. Any data sent over the persistent connection is encrypted by the SSL stack (Figure 3).

With this approach, small microcontroller-based devices can communicate with each other by simply exchanging data once the connection is established. They can also communicate with the application server running on a small, low-cost but resource richer platform. On the one side they don't even need to be on the Internet, but simply on a local Ethernet connected to a port on the server device. That server device then is connected to the Internet where its pages can be accessed via browsers from anywhere. Such a server can connect to potentially hundreds or thousands of small distributed devices. There are many single board computers on the market that can easily fulfill these requirements.

How those devices with their embedded applications are managed is then entirely a matter of the application on the server, which is accessed from a normal browser. They can, for example, be configured or updated collectively, be selected from lists or by defined groups, or even individually since all will have their own IPv6 IP address. The approach of using an intermediate server is also cost-effective because only one platform needs to run the full application server. Still, all the options are available.

Real Time Logic
Monarch Beach, CA.
(949) 388-1314.
[www.realtimelogic.com]