App Servers and Lua Scripting Speed Rich Web Applications for Small Devices

With ever more smart devices connecting to the web, even small embedded devices must be able to serve up rich graphical presentations of the data to satisfy user expectations. With time and space at a premium, a scripting approach can be invaluable.

by Wilfred Nilsen, Real Time Logic

Running a business used to be straightforward. You had development and production and marketing and sales, little of which had fundamentally changed for decades or more. And then the Internet happened. Suddenly everyone had to have websites and online support and shopping carts and Like buttons. Such a web presence has wormed its way ever deeper into our expectations: increasingly, in our Internet-of-things world, all devices must be connected through a web application.

This creates a new challenge for designers of small embedded systems. Not only is it a new task, but smartphones have set the bar ridiculously high when it comes to how sophisticated the application interface should be. We have come to expect that small devices can operate with color, depth and flair.

So whether it's a meteorologist checking on the weather in Antarctica or a seismologist checking bore temperatures deep in the earth, cryptic text or clunky graphics won't cut it. These folks don't care how little processing power your device has. They simply want to see things the way they're used to seeing things.

Of course, a web application is nothing more than software, and to a system designer, C may just feel like the natural way to approach this. But using C can mean spending as much time on the user interface as you spend on your core technology. There are much faster and easier ways to get a high-performance web application to market.

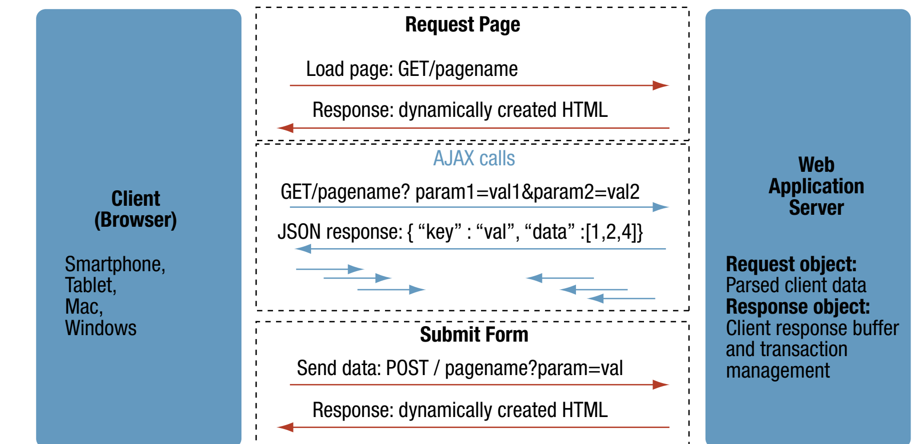

FIGURE 1

A web interface implements a series of requests, some explicit and some implicit, achieved ultimately in text with the assistance of abstraction technologies like AJAX and JSON.

Some Web Basics

While a good web application should provide a viewing experience that is natural to your user for the specific thing you're enabling them to do, in the end, it depends on an exchange of information over the Internet. That infrastructure can look deceptively simple. It's based on the HyperText Transfer Protocol (HTTP) and, as the name suggests, everything being communicated back and forth between the user's browser and your device—which the browser sees as a web server—is text. No

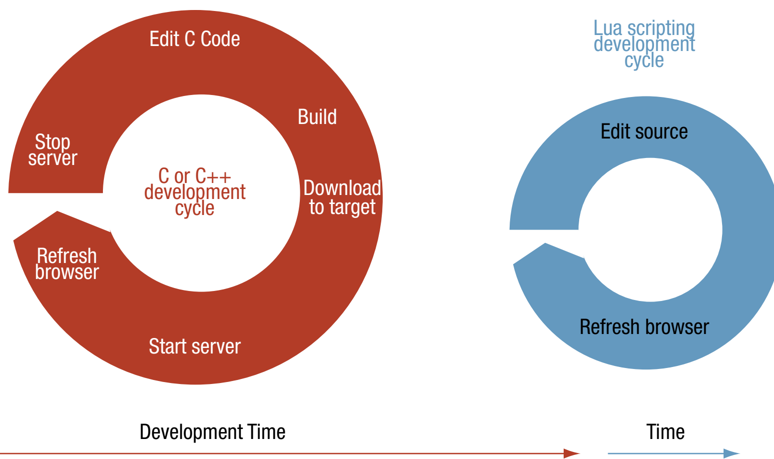

FIGURE 2

C code development takes much longer and is intrusive; Lua script development can be as much as 30 times faster without bringing the system down.

matter how complex the web experience, it all boils down to strings.

In the early days of the web, people wrote static web pages that consisted of Hyper Text Markup Language (HTML) text. A browser would request a page at some location, specified by a Uniform Resource Locator (URL), and the server would find that page and simply ship it back.

Then some clever people realized that, rather than having a page fixed in some location, you could dynamically create the page using a more sophisticated database of content along with the intelligence to assemble the information into a page.

One way of doing that is to have the server return a page with lots of JavaScript code. The browser then executes the JavaScript to render the page. Unfortunately, different browsers work differently, and relying entirely on JavaScript can create compatibility nightmares. The preferred alternative is to move much of the responsibility for assembling the page back to the server.

Then some more clever people created technologies like Asynchronous JavaScript And XML (AJAX) for requesting data in real time and JavaScript Object Notation (JSON) for communicating data objects. So what might seem like a single "Please give me a page"/"OK, here's the page you requested" exchange, is typically much more involved than that. Figure 1 illustrates a tiny portion of a typical exchange.

First the browser requests a page via a URL. If the requested page is, for example, a form, the server takes the data from the URL and stitches together a single HTML text page that the browser will render as a form. Now the user tries to populate the form. But if the form includes a pull-down list, for example, how does that pull-down get populated, especially if the contents depend on other form data?

This is where AJAX comes in. If the user clicks the pull-down, the browser quickly sends the server an AJAX request for the data that should be in the list. Those AJAX requests may use URL-encoded data to send parameters back to the server. If the request involves some high-level data object, then the server may respond using JSON, where data is structured and can be easily queried. A number of AJAX requests may be required before the form is ready to be submitted.

At some point, the user hits the "Submit" button, and the browser sends a new request to the server; the server responds with a new page. To the user, this looks like "go to a page and get a form; fill in the form; hit 'Submit'." But that apparent simplicity hides a complex conversation between the browser and server. Any small glitches or browser inconsistencies can throw the whole thing off.

Writing the Application

With that background in mind, your primary focus should be on your application, which is what directs which pages get sent when. When writing that application, your choices for language are typically two: C or web scripting languages.

You will likely need to write some portions of your application in C. Scripting environments intentionally restrict scripts from getting down to the hardware level. So you will need to write some C routines—similar to drivers—that will connect your hardware to your web application. For everything else, you can choose a scripting approach. So which is best, C or a scripting language? To figure that out, we can break the development work down into three categories. The first is managing the data, which is hopefully structured. The second is the parsing of requests, and the third is assembling responses.

With C, structures can't be entered into lightly: everything must be strictly typed, and memory must be explicitly reserved and released as needed. Parsing isn't rocket science, but it's tedious and extremely easy to get wrong, requiring incredible attention to every detail and making maintenance difficult. Assembling the page requires string concatenation on a grand scale. Just the simple act of joining two strings using C involves:

- Determining the length of both strings
- Reserving a spot of the appropriate size for the result
- Combining the two strings
- Sending the result off
- Releasing the memory used for the result

Much of this work has already been done for you, however, in the form of application servers and scripting environments. Because scripts are more free form and are compiled just-in-time, a single line of script can implement the entire string concatenation. The lower-level details are handled for you. Scripts also let you access and manipulate data without worrying about whether you've defined the right data type or organization.

Figure 2 contrasts the impact of developing with C vs. scripting, referring specifically to the Lua scripting language, which we'll discuss shortly. When you use C, you start by getting something basic up and running just to bring up the infrastructure. From then on, you've got this cycle of stopping the server, making changes (the most time-consuming portion), loading the new code, and starting the server again to see how things look. On the other hand, with a scripting language, not only

is the coding time dramatically reduced, but you can simply swap in the new scripts without interrupting anything else.

Meanwhile, you can use an application server to abstract the web server, giving you access to request and response objects and their associated APIs. This means your scripts really only deal with the high-level application behavior and data. The application server handles the parsing and page-building details.

Bottom line: you can develop your web application in as little as 1/30th the time by using scripts and pre-built infrastructure instead of custom C wherever possible.

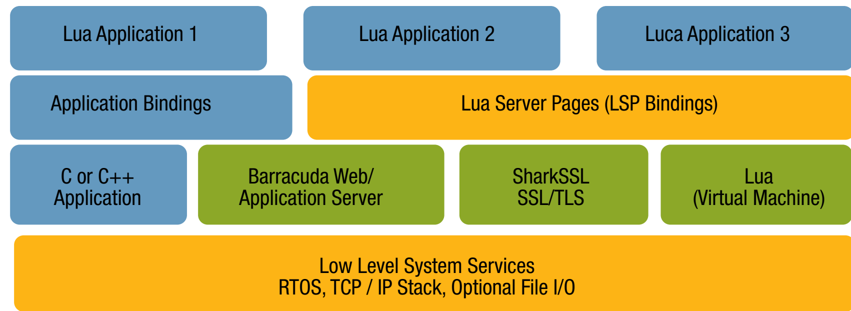

FIGURE 3

Applications written as Lua scripts interact with the application server and other blocks, including custom C routines. These save many months of development time and can run up to 20 times faster than LAMP.

Infrastructure and Scripting Options

Older websites were originally implemented using the Common Gateway Interface (CGI). In truth, CGI is only an interface. It's generally cumbersome to manage and requires a full-up OS like Linux that can load external programs, meaning that deep embedded monolithic systems cannot use CGI. Basic web servers typically specify only function hooks; you must write the functions.

With CGI on standard web servers, Perl scripting is very common, but most embedded environments don't support Perl, meaning you have to revert to C to get things done. All of this makes CGI an unattractive option. The most common modern alternative to CGI combines Linux as an OS, Apache as a web server, MySQL as a database, and PHP as a scripting environment—collectively called a LAMP setup.

LAMP setups work well in full-up web server implementations. Unfortunately, they demand far more processing power—primarily CPU speed, but also around 65 Mbyte of memory—than is available in a small embedded device. The application becomes unacceptably slow.

An example of this can be seen in one specific network-attached storage (NAS) device that provides a web interface. Even though the processor runs at 900 MHz, its PHP response is so bad that every page request is met by a rotating hourglass. In other words, because they couldn't speed up the interface, they had to stuff an hourglass in there as a "please wait" indicator to keep the user from thinking that nothing is happening.

For small implementations like this, you need a web server that has been designed to operate efficiently and quickly with modest processing power—as slowly as 60 MHz—and little memory (1 Mbyte or less of RAM and ROM). And, because many of these devices may be located far from the person trying to communicate with the device—like our meteorologist who, mercifully, isn't in the Antarctic—the system must be easy to manage remotely. One way to simplify management is by compressing web pages together to reduce their footprint; updates to the system can then be made simply by swapping zip files. An example of such a server is the Barracuda Web/Application Server.

For efficient scripting in an embedded environment, there's a quiet, unassuming scripting language called Lua. You may not have heard of it, but, whether you've used Adobe's Lightbox program, played World of Warcraft, or used any number of other programs, you've used Lua. It's specifically designed to operate on small platforms, and yet handles things like garbage collection, callbacks and type coercion automatically and transparently. It's specifically defined as an extensible language and is implemented as a library that gets compiled with your application.

As a result, an application server can build on the language to add powerful capabilities. So the interplay between Lua and the application server you choose can also determine how powerful your environment will be. Running together, as shown in Figure 3, the Barracuda Web/Application Server and Lua deliver applications that run as much as 20 times faster than they would in a LAMP setup. The system services, application server, SSL stack and Lua virtual machine allow you to focus on your application logic using high-level data structures in Lua scripts. You use C only for low-level access to your hardware, something the scripts are forbidden to do.

In the end, what really matters is that your users experience your system in a way that meets the standards they're already used to. Whether your users access your device by desktop or smartphone, what they see should look like a desktop or smartphone application. They won't be forgiving just because you have a small device acting as a server. After all, today's smartphones appear to be small devices. The typical user does not understand how much compute power lies beneath the covers.

Even though you must give them the look they want, you can't spend a lot of time building that look. The interface should look as advanced as your system technology, but you should be spending most of your development time on your system, not the interface. You can spend months—even years—trying to code a server in C. However, if you use a web server that's designed for an embedded environment and then do your own customization using Lua scripts wherever possible, you can take that development time down from months to days.

That gets you back to focusing on your own technical innovations. And it lets you present those innovations to your customers in the best possible way. ◢

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