7-2011

# REST on Routers? Preliminary Lessons for Language Designers, Framework Architects, and App Developers

Joseph P. Kaylor

Konstantin Läufer
*Loyola University Chicago*, klaeufer@gmail.com

George K. Thiruvathukal
*Loyola University Chicago*, gkt@cs.luc.edu

# REST ON ROUTERS?
## *Preliminary Lessons for Language Designers, Framework Architects, and App Developers*

Joe Kaylor, Konstantin Läufer and George K. Thiruvathukal
*Department of Computer Science, Loyola University Chicago, Chicago, IL 60611, USA*
*{jkaylor,laufer,gkt}@etl.luc.edu*

Keywords: embedded systems, environmental monitoring, green computing, Linux, OpenWrt, programming languages, representational state transfer, resource oriented computing, REST, software frameworks

Abstract: In this position paper, we provide a preliminary assessment of hardware and software solution stack choices available to developers of resource-oriented web services on commodity embedded devices. As part of an ongoing interdisciplinary research project on air and water quality in a major urban ecosystem, we are developing an information infrastructure amounting to a role-based hierarchy of individually addressable, interconnected resources, ranging from sensors, analyzers, and other monitoring devices to aggregators and publishers. This infrastructure follows the Representational State Transfer (REST) architectural pattern and integrates non-networked or non-RESTful monitoring devices through RESTful proxy resources running on low-cost, low-energy, possibly wireless, always-on embedded servers. Commodity wireless routers running a suitable embedded Linux distribution are a good choice for this purpose, and we have started to survey the landscape of supported solution stacks, including programming languages and RESTful frameworks: Not only were our preferred, familiar choices unavailable for medium-end routers, but we had to develop our own lightweight REST layer for lower-end routers. Given the growing popularity of embedded Linux devices, however, we argue that programming language designers and framework architects should support them to a much greater extent than they do now. In addition, as the demand for green computing grows, we argue that memory- and processor-efficient languages and frameworks become increasingly important.

## 1 INTRODUCTION

The purpose of this position paper is to provide a preliminary assessment of hardware and software solution stack choices available to developers of resource-oriented web services on low-power equipment. The context for this discussion is an ongoing interdisciplinary research project on air and water quality in a major urban ecosystem.

The information infrastructure we are developing for this project amounts to a role-based hierarchy of individually addressable, interconnected resources, ranging from a large number of sensors, analyzers, and other monitoring devices to aggregators and publishers. In developing this infrastructure, we follow the Representational State Transfer (REST) architectural pattern (Fielding, 2000); accordingly, we incorporate non-networked or non-RESTful monitoring devices through RESTful proxy resources running on low-cost, low-energy, possibly wireless, always-on embedded servers.

Given that such devices are readily available in the form of commodity wireless routers running a suitable embedded Linux distribution, we have started to survey the landscape of supported solution stacks, including programming languages and RESTful frameworks: Not only were our preferred, familiar choices unavailable for medium-end routers, but we had to develop our own lightweight REST layer for lower-end routers. Because embedded Linux devices are becoming increasingly common for a wide range of uses, however, we argue that language designers and software framework architects should support them to a much greater extent than they do now. In addition, as the demand for green computing grows, memory- and processor-efficient languages and frameworks become increasingly important.

## 2 THE NEED FOR RESTFUL THINKING

The Representational State Transfer (REST) architectural pattern (Fielding, 2000) is centered around addressable resources with a uniform interface and hypermedia representations. In RESTful web services, URIs are used as addresses, HTTP verbs (request methods) as the uniform interface, and XML or JSON (JavaScript Object Notation) as representations.

By exposing our hierarchical information infrastructure as a collection of interconnected RESTful web services, we allow the available information to be consumed in flexible ways by user interface presentation layers, data analysis tools, web application mashups, and other planned or unforeseen programmatic clients.

**Addressable resources** Our information infrastructure can be conceived naturally as a RESTful resource set (Pisupati and Brown, 2006; Taherkordi et al., 2010).

- The singleton root resource corresponds to the information infrastructure itself.
- Locations can be grouped at multiple levels corresponding to places, organizations, or organizational units.
- Each location can be configured to house one or more devices.
- Each device is responsible for measurements, such as nitrogen monoxide (NO), nitrogen dioxide ($NO_2$), or ozone ($O_3$).
- For any measurement, the device can provide the current reading or historical values such as the minimum, maximum, or average over a given time period. A unit of measurement is associated with each reading.

A topic for further investigation is the federation of disjoint resource sets (across physical servers) into a single, seamlessly browsable distributed resource.

**HTTP verbs as the uniform interface** Our resources support the main verbs of the uniform interface of HTTP, that is, the request methods GET, PUT, POST, and DELETE. These are similar to the familiar CRUD (Create, Read, Update, Delete) operations for manipulating resources but do not correspond one-to-one. Specifically, PUT is idempotent and corresponds to creating or fully updating a specific resource, while POST corresponds to adding a child resource, partially updating a resource (in absence of widespread support for the PATCH request method), or other non-idempotent operations. While most interaction with environmental sensors is read-only, some sensors do provide mutable resource state for configuration settings such as the unit of measurement, calibration settings, and the like. Our resources naturally support the uniform interface as follows:

- Obtaining a measurement reading or device setting from the sensor maps to the GET method.
- Providing a specific new value for a device setting is idempotent and, thus, maps to the PUT method.
- Toggling or cycling among several options is not idempotent and, thus, maps to the POST method.
- Some nodes in our infrastructure cache historical data as their resource state; explicitly deleting some of those data maps to the DELETE method.

**Hypermedia representations** The resources in our information infrastructure are naturally interconnected, and hypermedia representation formats expose these connections as links. For example, the representation of an aggregator node includes a link to its list of (statically known and/or dynamically discovered) children. In addition, in following the Hypermedia as the Engine of Application State (HATEOAS) principle (Fielding, 2008), the representations include links that represent the next actions, corresponding to state transitions, currently available to the consumer. For example, the representation of a reading includes a link to the device that produced the reading, and the representation of a device includes links to the various device settings, which can be modified with sufficient authorization.

**Implementation** Using the Restlet framework for Java, to which the second author contributed example code and documentation, we have implemented a RESTful proxy for monitoring devices that are network-capable but not RESTful on their own. Our implementation runs on a conventional Linux server and includes an adapter component for a class of devices that support the widely used TCP-based Modbus protocol. It is currently serves as a RESTful proxy for several Thermo Scientific air quality analyzers available at our institution. For example, our proxy maps routes of the form `/{location}/{device}/{measurement}/{reading}` to a resource that obtains a reading from a device. The fragment of the externalized configuration metadata (for the Spring Framework dependency injection container) in Figure 1 shows a specific location with a nitrogen oxide analyzer. The measurement register settings specify which Modbus data registers

```
<entry key="baumhart">
  <bean class="DefaultLocation">
    <property name="devices"><map>
      <entry key="42i">
        <bean class="ModbusDevice">
          <property name="hostname"
            value="147.126.68.251" />
          <property
            name="readableSettings"> ...
          </property>
          <property
            name="measurementRegisters"><map>
            <entry key="no2"><map>
              <entry key="current"
                value="0"/>
              <entry key="min" value="10"/>
```

Figure 1: Resource configuration in Spring

correspond to which readings from the analyzer. The complete code for this example is available online.

# 3 GREEN PERVASIVE COMPUTING

As pervasive computing becomes increasingly prevalent, more and more attention is given to green technology in the form of low-power, embedded devices. Indeed, such devices are increasingly common as part of the Internet of Things (Guinard et al., 2010) and the emerging Web of Things and serve a variety of needs, including home automation, home and small office security, home entertainment, weather and environmental monitoring, RFID and identity management, and near-field communication for presence and proximity applications.

Accordingly, one of the key nonfunctional requirements for our information infrastructure and its constituent devices is minimal power consumption. Other requirements include low cost, always-on operation, and, in some cases, wireless network connectivity. To this end, we examine the lower end of the server hardware spectrum, starting from the top.

**Conventional x86-based servers,** including low-energy versions such as Atom, Via C7, etc., typically include several gigabytes of RAM. These systems support the full spectrum of available software solution stacks, but at the expense of power consumption and memory use. Idle power consumption ranges from 30 watts for low-power fanless systems to several hundred watts for conventional systems. Cost starts around US$200.

**Plug computers,** usually ARM-based, have recently emerged as an alternative to conventional systems and typically offer half a gigabyte of RAM or more. These systems also support standard available solution stacks. Idle power consumption ranges from 5 to 15 watts. Cost starts around US$100 but can reach two or three times that amount for fanless systems with diverse I/O ports, such as eSATA and USB.

**Wireless routers,** network-attached storage (NAS) devices, and similar devices are typically based on ARM or embedded MIPS CPUs and feature 0-32MB on-board flash memory and 2-64MB RAM. Although these are sold as special-purpose consumer devices, numerous models can be converted to general-purpose embedded servers by installing a suitable embedded Linux distribution that supports a subset of the standard solution stacks (discussed in more detail below). Virtually all of these devices are fanless, and idle power consumption is around 1 to 3 watts. Cost ranges from US$40 to US$120 for routers depending on memory, wireless radio chipset, and presence of USB ports. The differences in power consumption and cost when compared to plug computers seem to be minor but quickly scale up when tens or hundreds of devices are involved.

**Single-board embedded computers** and microcontrollers, often ARM- or Atmel-based, are designed to perform device control tasks and are often limited to flash memory and RAM well below one megabyte. Some of these systems run embedded Linux distributions with very limited software and operating system stacks. Idle power consumption is well below 1 watt. Cost starts around US$20 for a bare board without case, not including the cable required to connect the chip to a host computer and program it. These devices are generally not suitable as stand-alone servers but could be useful when attached to, say, an embedded host computer.

Wireless routers and related devices in the second-last category appear to be the most economical devices in terms of cost, availability, power consumption, and physical footprint that can be used as general-purpose embedded Linux servers. Based on our evaluation, wireless routers and related devices offer the sweet spot in terms of these requirements:

**Reliability:** These devices are based on mature chipsets with a common architecture, such as ARM or MIPS. They are fanless and have no other moving parts, yet they are not so small as to impede good air flow for cooling.

**Ease of software development:** There are several choices of embedded Linux distributions for these

devices. Software development for these targets is well supported. Development typically takes place using an integrated development environment or other tools on a development host; the resulting code is then either cross-compiled for or directly interpreted on the embedded target.

**Active community support:** There are active, knowledgeable communities for both hardware and operating system.

Specific device choices include

- low end: ASUS WL520gU with a 200MHz Broadcom CPU, 4MB flash, and 16MB RAM for US$40
- mid-range: ASUS WL500gP v2 with a 240MHz Broadcom CPU, 8MB flash, and 32MB RAM for US$65
- high end: Buffalo WZR-HP-G300NH with a 400MHz Atheros CPU, 32MB flash and 64MB RAM for US$90

The remaining step is to choose an embedded Linux distribution. Some of the available choices, such as Tomato and DD-WRT, focus primarily on router functionality, while others, such as Embedded Debian, have too large a footprint for our target devices. We have chosen OpenWrt (OpenWrt, 2010) for the following reasons: support for a wide range of devices, including the three mentioned above; open and flexible with excellent build system; extensive documentation; mature code base under active development, and a competent and helpful community. Various embedded distros, including OpenWrt, replace the C library (usually glibc) with uClibc, which provides essentially the same functionality with a much smaller memory footprint.

We typically configure our devices to run as wireless clients (in so-called station mode) on an existing wireless network infrastructure. We have confirmed that the low-end WL520gU can run for four hours on four rechargeable NiMH AA batteries. In the near future, we plan to add a small solar panel to charge the battery pack continually. The advantage of such a configuration is that it can be deployed where desired but without the need for any wired connections.

# 4 RESTFUL SERVICES FOR EMBEDDED DEVICES

In practice, we have found it challenging to apply RESTful thinking to green computing on embedded devices. We cannot simply deploy a service developed for a conventional platform to an embedded one. For example, the RESTful sensor proxy example shown above, implemented in Java using the Restlet and Spring frameworks, will not run on the limited Java ME (Micro Edition) virtual machines available on embedded Linux platforms.

In the remainder of this section, we will discuss preliminary results from our ongoing effort to evaluate programming languages and REST frameworks. This effort is quite similar to the *implementing-rest* project (Amundsen et al., 2011), but with the added constraint of embedded Linux devices as deployment targets. As we will discuss below in more detail, this added constraint requires us to shift focus from Java and .NET to cross-compilation, scripting, and other lightweight approaches.

**Java** As mentioned above, Java on OpenWrt is limited to the Java ME platform with the Connected Device Configuration (CDC, JSR 218). The CDC Foundation Profile is a set of APIs designed for headless servers and other devices without a GUI. Java ME, still based on Java 1.4.2, is missing important recent additions to the language, most notably, annotations and complete support for reflection, as well as java.util.concurrent. Because most modern REST frameworks, dependency injection containers, and other commonly used frameworks and tools rely on these language features, they cannot be used on our target devices out of the box. Even the NetKernel resource-oriented platform, which explicitly supports Java 1.4.2, will not work out of the box on Java ME because it uses the String.replaceAll method instead of String.replace; we are currently investigating how much effort it would take to port NetKernel to this platform. Furthermore, many frameworks rely heavily on XML, which can be memory-intensive and for which Java ME support is limited (JSR 280); we propose to rely more on JSON than XML for lightweight externalized configuration and data exchange. Consequently, if one wants to develop Java services for embedded Linux devices, one is limited to a solution stack of older versions of the relevant layers, such as the Jetty 6.1.x: web server, db4o 7.x object database, beanshell 2.0b4 scripting environment, and PicoContainer 1.3 dependency injection container. Instead, we hope that there will at some point be a Java "Micro Enterprise Edition" that is more up-to-date language-wise and offers better support for RESTful service development for embedded devices.

**.NET/Mono** We hope that .NET on the Mono runtime will eventually be a viable alternative. Mono is known to run on ARM, but the pertinent documentation refers to Mono 1.x, while the current ver-

sion is 2.8.x. We successfully cross-compiled the Mono 2.8.1 runtime for OpenWrt and installed it on x86, ARM, and MIPS. While the x86 installation passed all tests included with the Mono runtime, only very simple programs worked on ARM and MIPS. This confirms that the problem is not using Mono on a uClibc-based system but possibly the just-in-time compilation for these non-x86 processors. We hope that this problem will be addressed eventually because of the wealth of REST and other frameworks available for .NET.

**Cross-compilation** In contrast with the byte-code-based Java and .NET platforms, using cross-compilation to generate binaries for the target devices is well supported. Languages such as C and C++ work well. In particular, C++ along with the Boost libraries is a promising choice for interfacing with external sensors or microcontrollers. By adding the POCO C++ libraries for building network-based applications, C++ could be an overall winner. We have not evaluated these libraries yet, but they appear to be well documented and under active development. Although Objective-C works as a language, the associated GNUstep framework is too resource-intensive for embedded targets. We were also interested in the Embedded ML project, which translates ML code to C code, which can then be cross-compiled. Unfortunately, the resulting binaries crashed immediately on x86, ARM, and MIPS, so we suspect that the generated code is not compatible with uClibc.

**Other interpreted and scripting languages** We have also experimented with various interpreted and scripting languages, which can very conveniently be developed on a host and interpreted on target at source or byte-code level. While all of these languages work more or less well on conventional hardware, the question is how well they scale down to embedded hardware, and this is where differences become apparent. Our preliminary experience is as follows:

**Erlang** is well supported on OpenWrt. There is a package for the Mnesia database, and one can manually install the RESTful Webmachine framework. We have already confirmed that this solution stack runs well on a mid-range router. Given how interesting Erlang is as a functional language, we are eager to evaluate this stack further.

**Lua** is directly supported in the form of a module for the extremely lightweight uhttpd server. We implemented a very minimal Lua script service that exposes data from a USB input device as a RESTful resource (see Figure 2) in a similar way as

```
sensors = {
  baumhart = {
    ts42i = {
      nitrogen = {
        no = {
          current = function()
            return read_sensor(device, 7)
          end,
        ...
```

Figure 2: Resource configuration in Lua

```
function map_path_to_resource(path, resource)
  pos = resource
  for word in
    string.gfind(path or "", "[^/]+") do
    pos = pos[word]
  end
  if type(pos) == "table" then
    header_ok()
    print("[\"" .. table.concat(keys(pos),
      "\", \"") .. "\"]")
  elseif type(pos) == "function" then
    header_ok()
    print(string.format("{ \"value\": %u }",
      pos()))
  else
    header_notfound()
  end
end
```

Figure 3: Mapping from URI path to resource

the previous Restlet/Spring example. The function shown in Figure 3 maps the request URI path to this Lua resource set object and returns a representation of the resource in the JavaScript Object Notation (JSON). The two auxiliary functions generate the HTTP response headers that precede the response body with the representation. The complete code is available online.

Notably, all packages required for this configuration, together with our Lua code, fit within the 4MB flash memory of the low-end WL520gU router. This is a key requirement for the following reason: The external input device is plugged into the single USB port of this router. Exceeding the available flash memory would require a USB memory stick and a USB hub. The additional required power would take us further away from the goal of battery- or solar-powering the router.

On mid-range systems, there are additional choices. Among many other packages, Lua provides the Orbit web framework, which supports the main HTTP request methods. Portions of this framework are written in C, but the luarocks package management system can be set up for cross-compilation. We have gotten basic server functionality to work on a mid-range router and plan

to evaluate Orbit during the next few months.

**Perl** is well supported with over 135 packages available, but we have not had an opportunity to evaluate it yet. Several RESTful frameworks for Perl have been mentioned on stackoverflow.com.

**PHP** is well supported with over 30 packages available. It runs within the lighttpd web server through FastCGI and appears to consume relatively little memory and other processor resources. Given that there are several RESTful frameworks for PHP, this choice looks promising and merits further evaluation.

**Python** also appears to be well supported with over 20 packages available. Based on our initial explorations, there appear to be some issues with Python's package management systems that must be resolved before further evaluation is possible.

**Ruby** is well supported in terms of the availability of packages and tools. Nevertheless, the gems package management system runs out of memory, sometimes requiring a reboot of the device. In addition, the WEBrick web server toolkit example works but caused over 100 processes under relatively light load, so this stack appears to be too heavyweight overall for our target device classes.

# 5 CONCLUSION

Based on our ongoing investigations, we recommend that developers of RESTful web services for embedded Linux devices be open toward alternatives to the mainstream Java and .NET platforms: several promising choices are available, including Erlang, Lua, and PHP. By choosing an appropriate solution stack, it is possible to use these devices as low-power servers with nearly equivalent functionality as their conventional x86-based counterparts. Conversely, language designers should be more supportive of embedded target platforms, and framework architects should be more aware of the limitations of current language support on these targets.

In the near term, we will conduct a broad-based systematic evaluation of the various language and framework combinations using web server performance tools such as httperf and siege along with lightweight memory profiling.

In the medium term, we plan to expand our explorations to devices in the next-lower device class of single-board embedded system and microcontrollers. Here, we expect C/C++ and possibly Lua to be the most viable options.

In the long term, we intend to apply RESTful thinking to novel hardware architectures. Although not in the direct scope of this paper, general purpose computing on graphics processing units (GPGPU) and other novel architectures are in dire need of more resource-oriented thinking to allow for better integration in various distributed systems scenarios. While these architectures are not low in absolute power consumption, they are very power-efficient when considering their computational performance. An example is where a lower-power device, say, is taking sensor readings and needs to offload the analysis to a more powerful device for data analysis (e.g., time-series, compression, etc.) Here, a GPU would be a power-efficient way to support collective operations for large numbers of data supplier devices.

# ACKNOWLEDGEMENTS

We are grateful to Loyola University Chicago's Center for Urban Environmental Research and Policy (CUERP) for its support. We thank Sergio Fanchiotti for valuable discussions and suggestions for future investigations.

# REFERENCES

Amundsen, M. et al. (2011). implementing-rest: Exploring the implementation aspects of the REST architectural style. http://code.google.com/p/implementing-rest.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Fielding, R. T. (2008). REST APIs must be hypertext-driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

Guinard, D., Trifa, V., Karnouskos, S., Spiess, P., and Savio, D. (2010). Interacting with the SOA-based Internet of Things: Discovery, query, selection, and on-demand provisioning of web services. *IEEE Trans. Serv. Comput.*, 3:223–235.

OpenWrt (2010). OpenWrt: a Linux distribution for embedded devices. http://openwrt.org.

Pisupati, B. and Brown, G. (2006). File system framework for organizing sensor networks. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 935–936, New York, NY, USA. ACM.

Taherkordi, A., Rouvoy, R., and Eliassen, F. (2010). A component-based approach for service distribution in sensor networks. In *Proc. 5th Intl. Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, MidSens '10, pages 22–28, New York, NY, USA. ACM.