Light-Weight Implementation Guidance
(LWIG)
Internet-Draft
Intended status: Informational
Expires: September 10, 2013

O. Hahm
E. Baccelli
INRIA
K. Schleiser
SpreeBytes

March 9, 2013

Painless Class 1 Devices Programming
draft-hahm-lwig-painless-constrained-programming-00

Abstract

In order to fit the constraints of Class 0 devices (offering much less than 10KiB of RAM and 100KiB of ROM) there are no alternatives to implementing IP protocols in special software environments, which impose programming paradigms that make implementation of protocol specifications significantly more complex. However, our experience implementing RFC 4944 and RFC 6282, TCP and UDP on Class 1 devices (offering approximately 10KiB of RAM and 100KiB of ROM) shows that there are alternatives concerning software environments in which to implement IP protocols, which avoid such complexity by providing a more developer-friendly environment. This draft shares this experience.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

In order to implement IP protocols with regard to the constraints of Class 0 devices [draft-ietf-lwig-terminology-01], tailored software environments must be used, such as [TinyOS] [Contiki-OS]. However, such software environments impose programming paradigms that can prove painful to use. For instance, [draft-kovatsch-lwig-class1-coap-00] mentions some such issues, encountered while implementing COAP [draft-ietf-core-coap-13]. However, our experience with implementing RFC 4944, RFC 6282, TCP and UDP shows that such issues can be avoided on Class 1 devices, depending on the choice of software environment in which specifications are implemented, as detailed in Section 3.

2. Why implementations on Class 0 devices can be painful

In order to deal with the very limited RAM and ROM available on Class 0 devices, network protocol implementers generally code on specific operating systems based on an event loop with a shared stack for all processes (cooperative multi-threading). The main advantage of a cooperative multi-threading is that it can optimize memory usage on very constrained devices. However, the use of an event loop makes network protocol implementation a peculiar exercise, as it imposes an unusual programming paradigm.

2.1. Learning curve

The event loop programming paradigm is not straight-forward to master for programmers who are used to code on more conventional platforms, such as Linux or Unix for instance. Hence, programmers do not only have to deal with the constrained resources (RAM, ROM) of the platform, but must also tussle with a fundamentally different way of designing the implementation of a protocol specification. This means extra costs for code development and maintenance.

2.2. Implementing from scratch

The distinct programming paradigm stemming from an event loop typically imposes a non-standard programming language. Some event loop operating systems introduce a new language (e.g. [TinyOS] imposes nesC), while other operating systems "hijack" elements of a standard programming language to use it in a different way (e.g. the switch/case structure for Protothreads in [Contiki-OS], using a C dialect). This results in:

- the necessity for an entirely new code base,

- the inability to easily port pre-existing, mature code for well-known functions, routines, or protocols,

- the inability to use standard software engineering tools for debugging and analysing code.

Conversely, it is not easy to port code developed for Class 0 devices running on an event loop operating system, to less constrained devices running conventional operating systems.

2.3. Increased complexity

A system based on an event loop in combination with a shared stack for all processes makes it necessary to use somewhat tricky programming techniques. For instance, in order to not fully block the whole system during long-lasting processes, functions must be designed with split phase execution. However, this technique breaks up logical contigous code into multiple pieces and thus increases code complexity. Another example concerns handling multiple stateful connections at the same time: an event loop in combination with a shared stack typically forces programmers to implement complex state machinery to manage simultaneous connections (e.g., TCP connections).

3. Why implementations on Class 1 devices can be painless

When we were tasked with implementing RFC 4944, RFC 6282, TCP and UDP on Class 1 devices, we wanted to avoid the issues described in Section 2. We have thus decided to look for alternatives to using an operating system based on an event loop. Our implementation of the aforementioned protocol specifications was thus carried out in a different operating system, [RIOT-OS], which supports multi-threading similarly to operating systems running on "traditional" Internet hosts and routers (e.g., Linux or Unix). Our experience with this choice is that network protocol implementations for Class 1 devices can be comparatively painless, as described in the following.

3.1. Average programmer background is OK

As the operating system we chose supports fully separated threads and the ability to react in real-time, there was no need to fundamentally change the programming paradigm, compared to programming on less constrained platforms. Hence, we benefited from a drastically reduced learning curve for programmers with a usual background, i.e. that had never before programmed for constrained devices. Practiced network programmers could simply adapt the implementation concepts and techniques known to be efficient on non-embedded systems.

3.2. Leveraging more well-known tools

As the operating system we chose allows to implement in ANSI C, existing code could be reused, taken from the IP protocol suite currently deployed on mature operating systems such as Linux or Unix. For instance, our 6LoWPAN and TCP implementations reuse constants and data structures for packet headers and protocol options. The well-known BSD socket API was ported without significant modifications to functions like bind(), send() or recvfrom(). Useful helper modules like inet_pton and inet_ntop were integrated without any changes. In a nutshell: the usage of standard programming languages in combination with a common programming paradigm reduces the amount of code that has to be developed and maintained.

3.3. Safer and quicker coding

Coding can be significantly safer and quicker on Class 1 devices, depending on the choice of operating system upon which to build. Network protocols from the IP suite were designed with traditional, multi-threading based operating systems in mind. Thus, where possible, it makes sense to avoid unnecessary issues that stem from using fundamentally different software environments. Our experience shows that for implementations on Class 1 devices, it is possible to benefit from a software environment that leverages substantially more well-known tools than what an operating system based on an event loop can achieve, thus facilitating both code development on Class 1 devices, and porting of code to/from less constrained hardware.

4. Security Considerations

This document does not have any security considerations.

5. IANA Considerations

This document does not have any IANA actions.

6. Informative References

[RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, September 2007.

[RFC6282] Hui, J. and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks", RFC 6282,

September 2011.

[draft-ietf-lwig-terminology-01]	Bormann, C. and M. Ersue, "Terminology for Constrained Node Networks", draft-ietf-lwig-terminology , Feb 2013.
[draft-kovatsch-lwig-class1-coap-00]	Kovatsch, M., "Implementing CoAP for Class 1 devices", draft-kovatsch-lwig-class1-coap , Oct 2012.
[draft-ietf-core-coap-13]	Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap , Dec 2012.
[TinyOS]	"TinyOS http://www.tinyos.net", 2012.
[Contiki-OS]	"Contiki Operating System http://www.contiki-os.org", 2012.
[RIOT-OS]	"RIOT Operating System http://www.riot-os.org", 2012.

Authors' Addresses

Oliver Hahm
INRIA

EMail: Oliver.Hahm@inria.fr

Emmanuel Baccelli
INRIA

EMail: Emmanuel.Baccelli@inria.fr
URI: http://www.emmanuelbaccelli.org/

Kaspar Schleiser
SpreeBytes

EMail: kaspar@schleiser.de