LWIG WG
Internet-Draft
Expires: September 7, 2012

C. Williams
MCSR Labs
G. Mulligan
Proto6
March 6, 2012

On Lightweight and Embedded IP Programming Interfaces
draft-williams-lwig-api-00.txt

Abstract

This document takes a look at various aspects of Application Programming Interfaces (APIs) used in embedded sensors and controller applications such as IP Smart Objects and IP based Wireless Sensor Networks. These devices may be interconnected via many different types of media, including 802.15.4 (lowpans), power line control (PLC), RS485, but the common characteristic is that the devices have extremely limited code space and memory space for both the stack and application. Just as there is no one single API for IP networking stacks today (though the "Berkeley sockets" might be considered de-facto standard) there is not likely to be a single standard in the embedded space, but there can be some common understanding about facilities that can and should be provided to the application developer.

Table of Contents

1. Introduction

Just as there is no one single Application Programming Interface (API) for IP networking stacks today (though the "Berkeley sockets" might be considered a de-facto standard) there is not likely to be a single standard in the embedded space, but there can be some common understanding about facilities that can and should be provided to the application developer. This document takes a look at various aspects of APIs used in embedded sensor and controller applications and Wireless Sensor Networks, such as IP Smart Objects and Machine to Machine (M2M) applications.

Today in some embedded IP applications the IP stack is implemented in a separate external processor that provides a service like a Modem, while the application is run in its own processor. This has provided workable but more costly solutions by requiring two processors. This has the advantage that is separates the application from the network stack and offloads the network processing, but for small low-cost embedded systems the cost of the second processor can be prohibitive. In these cases combining the application and communications processors and providing an application environment in the single processor will provide a lower cost solution.

2. Definition Of Terms

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [1].

LowPan Network:
It is a wireless (usually) or wired network generally characterized by having constrained nodes. These constraints may be processing, memory or power or any combination

3. Assumptions

- The 6LowPan host nodes either autoconfigure IPv6 address based on the prefix received in the Router Advertisement, or it uses DHCP for short address assignment. It can receive multiple Router Advertisements and should configure at least one default router as its immediate nexthop. It may configure multiple default routers, but this is implementation specific. The 6LowPan host nodes always send their packets to the default router. If one default router becomes unavailable, it chooses the next available default router or it may restart the ND process.

4. Applicability Statement

This document aims to guide implementers in choosing appropriate programming interfaces for use in embedded IP devices, such as IP Smart Objects and other highly constrained devices with limited RAM or processing power

5. API requirements

5.1. Network Parameters

Different types of networks will require different types of functions to set-up and manage the network interface. In the case of wireless networks such a IEEE 802.15.4 it is necessary that functions be provided to select the channel and the PANID and possible set or read other IEEE 802.15.4 MAC parameters.

5.2. Sending and Receiving packets

All APIs must provide some mechanism to send and receive IP packets, otherwise what is the point of a networking stack. Additionaly they may provide higher functions that will compose TCP streams and UDP packets.

5.3. Managing network errors

All stacks must provide some set of functions to pass network erros to the application. These errors might include local networking errors, such as no IP address set for the interface, no next hop, no route to the destination. These errors might also include remote network errors such as those received via ICMP.

6. To RTOS or not

One fundamental question for all embedded stack developers is whether they should provide their stack with an RTOS or just a set of library functions. There are pros and cons to each.

6.1. Using an RTOS

Certainly providing and RTOS does offer a more complete solution, but it tends to contrain the application developer and may add unnecessary overhead (processing, code, memory and power).

TinyOS and Contiki are two open source RTOS. Each provides networking functionality and both now offer support for 6lowpan and

IPv6. They both have the supporters and detractors. Some developers shy away from TinyOS because they don't want to delve into the world of nesC. Some developers have chosen to not use Contiki because of the design using "protothreads".

The advantage that any RTOS provides is that the developer usually does not need to deal with many of the basic timings and can ostensibly write their application as just that, an application, and leave the details to the OS. The disadvantage is that if the application requires either some very specifc timing or possibly some detailed device or network control (sleeping, interrupts, ...) if these are not provided by the RTOS then the application may become more complex than otherwise required.

Additionally, as already mentioned, the RTOS may provide unnecessary functionality which could impact code size or memory requirements. The RTOS might also require a change in the design of existing embedded applications in order to be integrated into this environment.

6.2. Providing libraries

Providing libraries is not a panacea either. While a set of libraries offers the most flexibility (other than writing the complete stack and application), it does put the burden on the programmer to deal with all of the nuances normally taken care of by the OS. As such the programmer must have a better understanding of the specific of the particular microprocessor.

7. Low level programming interfaces

7.1. Berkeley Sockets

A number of companies have implemented embedded networking stacks and provide an interface to the stack (either an RTOS or set of libraries) via a Berkely Socket like set of functions. The major advantage is that Berkely Sockets is widely known, understood and taught. This can greatly speed up the development of networked embedded systems. If the set of library functions it properly written the basic network calls should be able to be used just as if they were a being written on a Unix or Linux system. In the embedded environment it is probably not possible to provide a complete POSIX compliant network interface, but a sufficient subset of functions can be implemented.

7.1.1. Network management

TBD

7.1.2. UDP

Sending and receiving UDP packets

7.1.3. TCP

TBD

8. High level programming interfaces

8.1. Java

The Sun SPOT team built an embedded stack based on the JAVA VM. The stack provided a mechanism to write embedded networking code, including "mobile code", in Java. All of the implementation of the protocols was provided as part of the JVM. The implementation of the JVM required a processor with 8MB of flash and 1MB of RAM. This is larger than would typically be considered an embedded system.

8.2. Python

At least one company has developed a python engine for small embedded devices. They provide a reduced set of phyton library functions, but they do include the ability to send a receive IP packets.

More information TBC (to be completed)

8.3. Proprietary

TBC

9. Modem type device interface

Some providers of embedded communications devices have chosen to provide a "closed" external processor that is used to send a receive packets, much like a modem of yesteryears. Some companies have gone as far as to overload the old AT command set to manage the network interface. For example to set the destination address the application processor send the string "ATDT ipaddress/port" as though it was asking the communications process to dial a phone number. Prior to that the application would send other AT commands to define various other parameters such as whether to treat the data as

datagrams (UDP) or a stream (TCP). In one case this was overloaded using ATDT (Dial Tone) for TCP and ATDP (Dial Pulse) for UDP and then the application just sends bytes and the outboard processor deals with all the details of managing the connection and sending and receiving data.

As already mentioned, while this provides an extremely simple interface and insulates the application developer from any details of the network, it adds cost to the overall system and may not provide an abstraction that allows the system to meet power or timing constraints.

10. Security Considerations

No known security considerations.

11. IANA Considerations

There are no IANA considerations for this document.

12. Acknowledgements

The ideas behind this came from discussion with a number of people including Eric Gnoske, Colin O'Flynn, Kris Pister, David Ewing and folks working in the 6LowPAN WG.

13. References

13.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

13.2. Informative References

[2] Montenegro, G. and N. Kushalnagar, "Transmission of IPv6 Packets over IEEE 802.15.4 networks", RFC 4944, September 2007.

[3] Kushalnagar, N. and G. Montenegro, "6LoWPAN: Overview, Assumptions, Problem Statement and Goals", RFC 4919, August 2007.

[4] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6), Specification", RFC 2460, December 1998.

[5] IEEE Computer Society, "IEEE Std. 802.15.4-2003", , October 2003.

Authors' Addresses

Carl Williams
MCSR Labs
USA

Email: carlw@mcsr-labs.org

Geoff Mulligan
Proto6
USA

Email: geoff@proto6.com