Old Wine in New Skins? Revisiting the Software Architecture for IP Network Stacks on Constrained IoT Devices

Hauke Petersen
Freie Universität Berlin
hauke.petersen@fu-berlin.de

Martine Lenders
Freie Universität Berlin
martine.lenders@fu-berlin.de

Matthias Wählisch
Freie Universität Berlin
m.waehlisch@fu-berlin.de

Oliver Hahm
INRIA
oliver.hahm@inria.fr

Emmanuel Baccelli
INRIA
emmanuel.baccelli@inria.fr

Abstract

In this paper, we argue that existing concepts for the design and implementation of network stacks for constrained devices do not comply with the requirements of current and upcoming Internet of Things (IoT) use cases. The IoT requires not only a lightweight but also a modular network stack, based on standards. We discuss functional and non-functional requirements for the software architecture of the network stack on constrained IoT devices. Then, revisiting concepts from the early Internet as well as current implementations, we propose a future-proof alternative to existing IoT network stack architectures, and provide an initial evaluation of this proposal based on its implementation running on top of state-of-the-art IoT operating system and hardware.

Categories and Subject Descriptors

C.2.1 [**Computer-Comm. Networks**]: Network Architecture and Design; C.2.2 [**Computer-Comm. Networks**]: Network Protocols; C.2.6 [**Computer-Comm. Networks**]: Internetworking; C.3 [**Special-purpose and application-based systems**]:

Keywords

IoT; OS; Network Stack; IPv6; Deployment; Software Architecture

1. INTRODUCTION

The Internet of Things (IoT) promises a future where all machines have started talking to one another, including billions of cheap, tiny, programmable, communicating devices (aka Things) such as wired or wireless sensors, and actuators. Based on various types of low-cost microcontrollers and communication chips, those devices will significantly increase heterogeneity within the Internet.

The past has shown that the success of the Internet depends on the availability of network stack(s) that allow for flexible composition of standards and enabling a wide variety of optional features to fit heterogeneous use cases.

The design and implementation of networks stacks challenged system engineers since the very beginning of computer networking. At that time computers exhibited severe hardware resources constraints, similar to IoT devices nowadays in terms of memory (kBytes instead of GBytes), and in terms of CPU (Mflops instead of Gflops). However, in contrast to the 80s and early 90s, the heterogeneity and thus the set of options and scenarios is much larger in the IoT, which increases complexity [20, 7, 22, 10]. Furthermore, Moore's Law does not apply to microcontrollers, and thus, such tiny devices will remain prevalent in the future [6]. Since full-featured systems such as Linux cannot be accommodated on such tiny devices, novel solutions are needed.

In this paper, we argue to revisit the design and implementation space of network stacks for constrained devices. Recent operating systems (e.g., RIOT [15]) support Linux-like functions but comply with the hardware constraints of IoT hardware, which gives potential to build flexible network stacks with low memory foot-print.

We target class 1 devices [6] or bigger, i.e., devices with at least 10 kByte of RAM and a few tens of kByte of ROM. We believe that for even more constrained devices, there is no way around specialized, simplified, and highly optimized implementations. Therefore, note that our goal is not to engineer the most memory-efficient network stack but to design a clean, structured, and universal network stack that can be reused for many different IoT use cases, while still being able to cope with constrained environments. In detail, our contributions are as follows:

1. we identify functional and non-functional requirements for the software architecture of the network stack on IoT devices (see Section 2),
2. we analyze existing IoT network stack architectures, from a systems point of view (see Section 3),
3. we propose an alternative architecture which we argue is more future-proof than existing architectures, because easier to use for continuous extensions, to configure for

IoT-Sys 2015, May 18, 2015, Florence, Italy.

http://dx.doi.org/10.1145/2753476.2753485.

different IoT use cases, leveraging cleaner interfaces and newly available IoT operating system services (see Section 4),
4. we provide initial evaluation of the proposed architecture and show it complies with typical resource constraints of IoT hardware (see Section 5).

2. ASSUMPTIONS & DESIGN OBJECTIVES

As the complexity of software for embedded devices has increased over the last decade, it has become state-of-the-art to use operating systems even on memory and CPU constrained machines, such as IoT devices. A full-featured network stack is one of the most complex pieces of software to run on an embedded platform. By full-featured, we refer to a stack allowing for a complete implementation of the specifications per design. This point is especially important, as one can easily simplify parts of an implementation at the price of limiting the extent of completeness that this implementation can achieve in the end. In the following, we will make the assumptions listed below.

Multi-Process & Hardware Independence. We assume that the network stack is built atop such an OS that provides the following features: (i) support of threads/processes, (ii) a lightweight process model, (iii) efficient inter-process communication (IPC), (iv) lightweight hardware abstraction, (v) a clean driver model, and (vi) a memory foot-print suitable for IoT devices. Assumptions (i)-(iii) allow for a modular network stack that is split over multiple processes without a significant overhead through administrative data structures and run-time drawbacks. Assumptions (iv) and (v) enable the network stack to be independent from specific IoT hardware platforms and network devices. We also assume that the OS allows the network stack to be open source, maintained by a lively community (similarly to Linux).

It is worth noting that our assumptions are reasonable; the operating system RIOT [15] matches all of them and thus allows us a proof of concept of our proposed architecture.

2.1 High-Level Objective & Approach

The usual approach to deal with the heterogeneity of embedded systems in the IoT (i.e. hardware constraints, use cases) is to implement multiple network stacks — each designed for a specific setup. While this yields optimized results for a small group of scenarios, there are drawbacks: multiple implementations vastly increase efforts for implementation, testing, maintenance, and incur extra efforts to ensure interoperability.

We thus pursue a different approach: we aim for a single, full-featured network stack that is flexible enough to work in a broad range of IoT scenarios, while still being efficient and small enough to run on constrained and battery-driven devices. In the following, we break down the various aspects of this high-level objective.

2.2 Functional Requirements

Focus on IPv6. The network stack should enable end-to-end connectivity between IoT devices and any other Internet device. IPv6 is a good candidate for this functionality, together with the 6LoWPAN suite of IP protocols for low-power lossy networks (including RPL, UDP, CoAP etc.). Note that that our design should also easily apply to other layered network stacks. For this reason we will focus, but not exclusively, on IPv6.

Full-featured. We aim for a full-featured network stack in a sense that supported protocols should implement their specifications completely as a long-term goal. The point is to prohibit design decisions which will limit future extensions of an implementation. The rationale behind this is to allow for a generic solution, which can be tailored to fit various use cases, instead of a solution that is too specific by design.

Support for multiple network interfaces. IoT scenarios do not only include basic sensors with a microcontroller and a single low-power radio, but also border routers with multiple interfaces (e.g., Ethernet and IEEE 802.15.4) as well as upcoming IoT devices, which are likely to have multiple radio interfaces (e.g., IEEE 802.15.4 and Bluetooth). Thus, the network stack must be able to handle multiple network interfaces, and we argue that, if designed carefully, the overhead of multi-interface support is negligible compared to single interface support, even on constrained devices.

Parallel data handling. Most embedded network stacks achieve their small memory footprint by reducing their functionality, to the point where they are only able to handle a single network packet at a time. While this might be reasonable in some use cases, this is unrealistic in general. In particular, using IPv6 over spontaneous wireless networking, multiple services run in parallel, e.g., both routing and neighbor discovery protocols are tightly coupled to data transfers between nodes. Thus, the network stack must be able to handle multiple packets and data streams in parallel.

2.3 Non-functional Requirements

Open Standards and tools. Decades of experience with the Internet indicate that deployment success depends on (open) standards. To achieve future-proof interoperability despite heterogeneity amongst IoT devices, the network stack must be standard compliant. Heterogeneity is not only found in IoT hardware but also in development environments and processes. We argue that a standard network stack should only depend on open tools and standard paradigms (e.g. ANSI C) to allow easy integration. Exotic tools and programming languages become a fatal hurdle on the way to reaching the critical mass of developers necessary to develop and maintain in the long run a piece of software as sophisticated as a network stack (a typical example of this phenomenon is TinyOS with the nesC language [18])

Configurability. The objective is the design of a versatile network stack that can be adapted to a variety of IoT scenarios. However, the granularity of configuration should avoid too many configuration options that have unclear meaning and effects (and thus are only usable for experts). Key configuration parameters must be well documented and accessible from a central point to achieve a user-friendly and flexible solution.

Extensibility via clean interfaces. Clean interfaces yield two important advantages. First, it focuses modules on their core functionality, thus preventing entangled code. Second, it yields testability by design. Furthermore, modules and clean interfaces enable substitution of parts of the network stack, which can easily be tailored according to the IoT scenario. For example, it is straightforward to switch between two different implementations of a neighbor cache, one being optimized for run-time performance using a heap data structure, and another being optimized for memory efficiency using a simple circular list. However, again, the granularity of modules should remain coarse enough to avoid

the pitfalls of ultra-fragmented code, which quickly becomes unmanageable, as analyzed in [18].

Low memory footprint. While we do not aim for the smallest possible memory footprint (we have other goals, as stated above), we aim for very limited resources. For a concrete upper bound we aim for a maximum of 30Kb of ROM and 10Kb of RAM for a single interface configuration running 6LoWPAN, RPL and UDP. These target numbers align well with the available resource on class 1 devices [6], which we expect to one of the most significant classes of IoT devices in the near future.

Low-power design. Many IoT devices are expected to run for years on small batteries. Experience shows that optimizations for low-power are harder to add on, and thus should built-in by design, from the very beginning. This has mainly two consequences: (i) the design of the network stack must allow to easily vary the protocols used in different scenarios, as best suited, and (ii) the implementation must use efficient data-structures and algorithms allowing maximum sleep intervals for the CPU.

3. RELATED WORK: EXISTING NETWORK STACKS

Today's Internet is unthinkable without Linux/Unix and their network stacks, successors of the BSD 4.4 network stack [8, 25]. Although they were originally developed in times when the memory constraints of a typical computer were roughly comparable with that of current IoT devices, their development followed fundamentally different design objectives, focusing predominantly on throughput (this manifests itself e.g., in the way buffers are designed). Over the years this lead to a drastically increased memory footprint and made it inconceivable to run or port these stacks to IoT devices.

Over the last decade, and even more since 6LoWPAN has evolved, a number of network stacks have been developed specifically for embedded devices. One category of stacks are ultra-minimalistic implementations, such as the work by *Santos et al.* [9], which – by design – are not extensible and cannot become a full-featured IP stack. Thus, they do not meet the requirements from Section 2. Various other stacks, as presented by several surveys, can be roughly be put in three groups (i) discontinued, (ii) proprietary and closed-source, or (iii) open-source and freely available [21, 24, 26]. In the following we will focus on the third group (the analyzed requirements disqualifies the others).

Sensinode's open *NanoStack 1.1* [17] was superseded by the proprietary implementation of *NanoStack 2.0*, thus does not satisfy the requirements we derived in Section 2.

A number of relevant network stacks were based on *TinyOS* [19]. However, since they were using *TinyOS'* exotic programming language nesC, they do not match the requirements from Section 2. Additional, we argue that due to the high complexity of *TinyOS'* system design and therefore limited number of available developers (as analyzed by P. Levis [18]), it is very unlikely that development of these stacks will keep up with the evolution of new IoT protocols.

An interesting approach towards a fully configurable network stack for embedded environments was proposed with *CiAO/IP* [5]. However, it does not match the derived requirements for similar reasons as the stacks for *TiniyOS*, since it is based on an exotic C++ dialect and an exotic compiler. Moreover, the intended granularity of configurability is too fine grained to be manageable by most application developers.

The two most prominent embedded network stacks today are *uIP* [12] and *lwIP* [11]. Both were developed at the same time by the same author as pure IPv4 stacks. Over time *uIP* has evolved from being developed as a stand-alone network stack to being maintained as the default network stack of the Contiki operating system, supporting a full 6LoWPAN protocol stack [13, 14]. The stack does however not support multiple network interfaces and is further based on an event loop paradigm. This makes it hard to program for a typical programmer experienced in traditional networking applications and more difficult to implement several protocols and mechanisms from the TCP/IP suite [16]. The *lwIP* stack is similar being developed over time, IPv6 support being recently added. For use in the IoT *lwIP* is missing support for 6LoWPAN. Although both stacks can be configured to a good extent, they are missing clear documentation and interfaces for easy extensibility. For these reasons we see both stacks failing to comply to the derived requirements.

4. GENERAL ARCHITECTURE

The key design rule for the proposed network stack software architecture is a strict module-driven design. We emphasize especially on a clean definition of the interfaces between these software modules as this ensures interchangeability of modules (i.e. to choose from different implementations for different scenarios) and interoperability of these modules. In this section we will give a brief overview on the most relevant design decisions.

4.1 Modular Design

The top level of the software architecture consists of a number of high-level modules, one for each functional entity of the network stack, for example UDP, IPv6, 6LoWPAN, or RPL. The novelty of the proposed architecture is that each high-level module is executed in its own thread while each module services the same API utilizing the operating systems IPC. The unified interfaces allows for chaining multiple modules together and the concept is comparable to Unix *STREAMS*, as proposed in the 80s [23], with the difference that we transferred the *STREAMS* concept to work via IPC.

Figure 1 illustrates as network stack configuration with three devices. The *netapi* depicts the unified IPC API between the high-level modules. Although each of these modules can roughly be mapped to layers of the TCP/IP model, the architecture does not enforce this mapping.

This design allows for a very flexible configuration of modules (even at run-time if needed) and, as important, it enables a straight-forward extension by new features or adding other layers. During design and implementation of modules this design enables further a clear separation of concerns and it enables for efficient testing of the modules. Using a unified IPC API yields further benefits when adding integrated network devices into the system that include already parts of the protocol stack, like Texas Instrument's CC3000 which already provides a full TCP/IP stack [4]. For a given network interface that e.g. already includes a full IP implementation one simply needs to write a host-side device driver that can service *netapi* and make it known to one or more transport layer modules.

One might argue that IPC comes with a high price w.r.t.

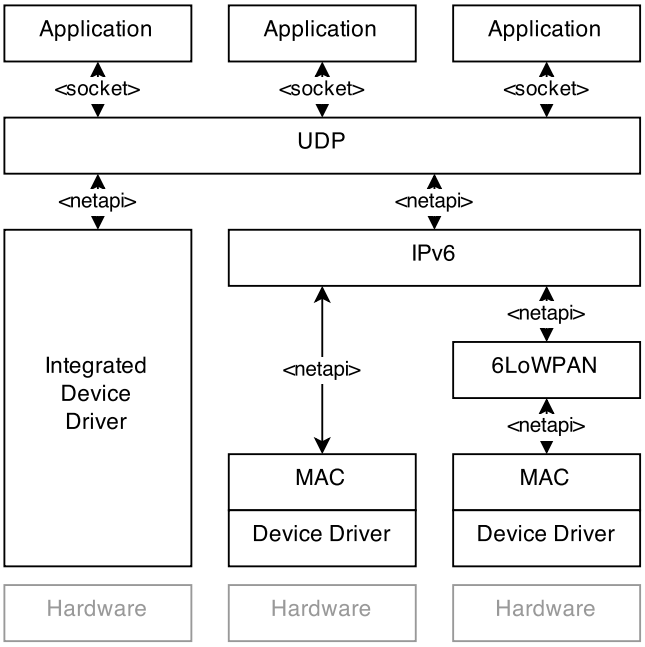

Figure 1: Sample configuration of a network stack. Each box depicts a high-level module running in it's own thread.

run-time performance and therefore energy usage. However, our measurements using RIOT on state-of-the-art IoT hardware (a 32-bit ARM Cortex-M3 platform) show that sending a message from one thread to another, including context save, running the scheduler and context restore, requires a number of CPU cycles that is only one order of magnitude more compared to the number of cycles needed for a direct function call. The benefits thus outweigh this overhead because (i) packet throughput on IoT devices is typically low, and (ii) there are few layers going up the stack, typically yielding IPC on less than 4 occasions.

4.2 Inter-module Communication: netapi

We introduce a unified interface for communication between high-level modules called *netapi*. This interface is built around a small set of messages sent between the modules utilizing the operating systems IPC. The idea behind this interface is that every layer in the network stack services an identical interface. The core of the *netapi* interface is a minimal set of messages, of which the most essential are *WRITE_DATA*, *REGISTER_RECEIVE_CALLBACK*, and *SET_ and GET_OPTION*. As each module must be able to parse the general format of *netapi* messages, it can implement any subset of possible message types and reply with an *ENOTSUP* ("Operation not supported", *POSIX.1-2001*) for all other message types [1].

4.3 Driver Interaction: netdev

The proposed architecture introduces a second unified interface for communication between device driver and medium access control (MAC) protocols, called *netdev*. In contrary to the *netapi* interface this API is based on direct function calls instead of IPC. The practical reason for introducing a second interface at this stage are the tight timing constraints of MAC protocols (e.g. schemes based on TDMA). Using the *netdev* API allows (i) for independent implementations of device drivers and MAC protocols and (ii) for better re-use and exchangeability of both, subsequently increasing the portability.

4.4 Packet Buffering

A key issue to solve in the design of a network stack for constrained devices is the handling of buffers for user data and protocol headers, as these are stored in RAM being one of the most limited resources. Typical design choices for these buffers include centralized approaches, copying data from module to module as well as mixed concepts. The data handling in the proposed network stack is designed around a 'copy twice' concept. Outgoing data is copied once from the user application (socket) into a central buffer and once into a network interface's device buffer by the device driver. The same is true for receiver data, which is copied on arrival once from the network interface into the central buffer and once more when handed over to an application.

The central packet buffer is designed as a central module accessible from all high-level modules through a well defined API. The buffers task is to centrally provision memory for storing header and user data while it is passing through the network stack, either as packets in one piece or as fragments. By accessing the packet buffer though a defined interface, it is further possible to transparently exchange the packet buffers implementation at compile time, e.g. one that manages a fragment of statically allocated memory against an implementation using dynamic memory on the heap.

The major advantages of a central buffer are (i) flexibility, (ii) efficiency through less data copying and (iii) the possibility to globally define the (maximum) amount of memory used. A drawback of a buffer taking chunks of data in different sizes is fragmentation, but we argue that with efficient implementation this disadvantage is marginal. By including means of prioritization for memory allocations in the packet buffers API, we can further make sure that no network module is being starved by missing buffer space, thus removing the major source for dead-locks.

5. PRELIMINARY EVALUATION

We implemented a proof of concept of our approach for the operating system RIOT [3]. To illustrate the principle feasibility we present the required amount of memory. Note that the values are still subject to optimization.

Our evaluation is based on a simple configuration using UDP, 6LoWPAN, and a single IEEE802.15.4 network interface built for the IoT-LAB_M3 hardware [2]. Table 1 shows the ROM usage for relevant modules of the network stack. Our modular network stack, which is based on common programming techniques and system calls, requires less than 30 kByte of ROM and is thus in line with IoT resources.

Module	IEEE 802.15.4	6LoWPAN and IPv6	UDP	Socket API	Helper Functions
Bytes	1,112	15,708	886	1,280	2,530

Table 1: Preliminary code size of main network stack modules on an IoT-LAB_M3 node (ARM Cortex-M3)

The RAM usage is mainly driven by two factors: (i) buffers and (ii) stacks. While the size for the central packet buffer is

dynamically configurable during compile time, we estimate that networks like IEEE802.15.4 require less than 1-2 kByte of RAM. The memory consumed by stacks is dependent on the number of high-level network modules that are configured. In our setup, we use one thread per network function (i.e., UDP, IP, 6LoWPAN, and the link-layer). With a default stack size of 1 kByte for ARM Cortex-M3 platforms, this estimates to an additional memory usage of 4 kByte. Overall, the required RAM size complies with the target platforms (i.e., < 10 kByte).

6. CONCLUSION & OUTLOOK

In this paper, we questioned the applicability of current network stack solutions for the Internet of Things (IoT). Following the observation that several IoT scenarios introduce constrained devices but do not require an ultimate memory-efficient network stack, we elaborate the design space and introduce a software architecture for a modular, full-featured network stack. Our proof of concept is based on a common system environment and requires < 10 kBytes of RAM and < 22 kBytes of ROM. Our next steps will be to complete our implementation for the open source operating system RIOT and explore the limits of our concept in different IoT scenarios.

Acknowledgements

This work was partially supported by ANR and BMBF within the projects SAFEST and Peeroskop.

7. REFERENCES

[1] IEEE Std 1003.1, 2004 Edition. http://www.unix.org/version3/, January 2002.

[2] IoT-LAB: Very large scale open wireless sensor network testbed. https://www.iot-lab.info/hardware/m3/, January 2015.

[3] RIOT Github Repository. https://github.com/RIOT-OS/RIOT, January 2015.

[4] TI's CC3000. http://www.ti.com/product/cc3000, January 2015.

[5] Borchert, C., Lohmann, D., and Spinczyk, O. CiAO/IP: A Highly Configurable Aspect-oriented IP Stack. In *Proc. of ACM MobiSys* (New York, NY, USA, 2012), ACM, pp. 435–448.

[6] Bormann, C., Ersue, M., and Keranen, A. Terminology for Constrained-Node Networks. RFC 7228, IETF, May 2014.

[7] Brandt, A., Buron, J., and Porcu, G. Home Automation Routing Requirements in Low-Power and Lossy Networks. RFC 5826, IETF, April 2010.

[8] Chesson, G. L. The Network Unix System. *SIGOPS Oper. Syst. Rev. 9*, 5 (Nov. 1975), 60–66.

[9] da Silva Santos, E. R., Vieira, M. A., and Vieira, L. F. Routing IPv6 over wireless networks with low-memory devices. In *Proc. of IEEE PIMRC* (2013), IEEE, pp. 2398–2402.

[10] Dohler, M., Watteyne, T., Winter, T., and Barthel, D. Routing Requirements for Urban Low-Power and Lossy Networks. RFC 5548, IETF, May 2009.

[11] Dunkels, A. Design and Implementation of the lwIP TCP/IP Stack. Tech. rep., Swedish Institute of Computer Science, 2001.

[12] Dunkels, A. Full TCP/IP for 8-bit architectures. In *Proc. of MobiSys* (2003), ACM, pp. 85–98.

[13] Dunkels, A., Gronvall, B., and Voigt, T. Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on* (2004), IEEE, pp. 455–462.

[14] Durvy, M., Abeillé, J., Wetterwald, P., O'Flynn, C., Leverett, B., Gnoske, E., Vidales, M., Mulligan, G., Tsiftes, N., Finne, N., and Dunkels, A. Making sensor networks IPv6 ready. In *Proc. of ACM SenSys* (2008), ACM, pp. 421–422.

[15] Hahm, O., Baccelli, E., Petersen, H., Wählisch, M., and Schmidt, T. Simply RIOT: Teaching and Experimental Research in the Internet of Things. In *Proc. of ACM/IEEE IPSN* (2014), ACM.

[16] Hahm, O., Baccelli, E., and Schleiser, K. Painless class 1 devices programming. Tech. rep.

[17] Lembo, S., Kuusisto, J., and Manner, J. In depth breakdown of a 6LoWPAN stack for sensor networks. *International Journal of Computer Networks & Communications (IJCNC) 2*, 6 (2010).

[18] Levis, P. Experiences from a Decade of TinyOS Development. In *OSDI* (2012), pp. 207–220.

[19] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. TinyOS: An Operating System for Sensor Networks. In *Ambient Intelligence*. Springer Berlin Heidelberg, 2005, pp. 115–148.

[20] Martocci, J., Mil, P. D., Riou, N., and Vermeylen, W. Building Automation Routing Requirements in Low-Power and Lossy Networks. RFC 5867, IETF, June 2010.

[21] Mazzer, Y., and Tourancheau, B. Comparisons of 6LoWPAN Implementations on Wireless Sensor Networks. In *Proc. of SENSORCOMM* (June 2009), pp. 689–692.

[22] Pister, K., Thubert, P., Dwars, S., and Phinney, T. Industrial Routing Requirements in Low-Power and Lossy Networks. RFC 5673, IETF, October 2009.

[23] Ritchie, D. M. The unix system: A stream input-output system. *AT&T Bell Laboratories Technical Journal 63*, 8 (1984), 1897–1910.

[24] Sarwar, U., Rao, G. S., Suryady, Z., and Khoshdelniat, R. A comparative study on available IPv6 platforms for wireless sensor network. *World Academy of Science, Engineering and Technology 62* (2010), 889–892.

[25] Wehrle, K. *The Linux networking architecture : design and implementation of network protocols in the Linux kernel*. Pearson Prentice Hall, Upper Saddle River, N.J, 2004.

[26] Yibo, C., mean Hou, K., Zhou, H., Shi, H.-L., Liu, X., Diao, X., Ding, H., Li, J.-J., and de Vaulx, C. 6LoWPAN Stacks: A Survey. In *Proc. of WiCOM* (2011).