Internet of Things for Smart Homes: Lessons Learned from the SPHERE Case Study

Atis Elsts, George Oikonomou, Xenofon Fafoutis, Robert Piechocki
Department of Electrical and Electronic Engineering,
University of Bristol, UK

***Abstract*—Building large-scale low-power Internet of Things (IoT) systems remains a challenge, as these systems have to meet the requirements of reliability, robustness, and energy-efficiency while running on resource-restricted microcontrollers without memory protection. In this paper we present the case study of IoT in SPHERE (Sensor Platform for HEalthcare in a Residential Environment), a project with the objective to develop a multipurpose, multi-modal sensor platform for monitoring people's health inside their homes. Atypically for academic projects, in 2017 the SPHERE software is going to be deployed in a 100-home study in volunteer homes, therefore it has to satisfy many real-world requirements. We discuss the requirements for IoT networking in this project, the IoT architecture (built on top of Contiki OS), software engineering challenges and lessons learned, as well as some of the general aspects that still make embedded low-power IoT software development difficult.**

I. Introduction

SPHERE[1] (Sensor Platform for HEalthcare in Residential Environment) is an EPSRC interdisciplinary research collaboration with the objective to develop a multipurpose, multi-modal sensor platform for monitoring people's health inside their homes [1]. In 2017 the SPHERE software is going to be deployed in a 100-home study in volunteer homes, where it is expected to be working for a period up to one year. Requirements such as these are not typical for academic software projects; as a consequence, the software has many requirements common with commercial applications: for instance, it has to be robust enough to adequately function outside of the lab for prolonged duration without any maintenance.

Even though there is a proliferation of commercially available wearable activity trackers and out-of-the box home-sensing systems, they are not a good fit for an academic project such as SPHERE. First, they lack flexibility and openness: these commercial solutions do not typically offer access to the raw data as required by the project. Second, these systems tend to store the data in a cloud, which is against the ethics requirements of SPHERE.

The solution chosen in the SPHERE project is to build a custom low-power embedded IoT system, based on state-of-art operating systems, libraries and network stacks. We chose the Contiki OS as the basis for our system, as it is one of the leading open-source operating systems in this field. On top of that, we run a fully standardized and interoperable IEEE 802.15.4 PHY / IEEE 802.15.4 TSCH / 6LoWPAN / IPv6 / CoAP network protocol stack. The hardware used is Texas Instruments CC2650 System-on-Chip (SoC), an ARM Cortex-M3 based Class-1 [2] IoT device with 20 kB RAM and 128 kB of flash memory. We believe that the problems faced in SPHERE are typical for projects using low-power embedded IoT systems. The contributions of this paper are:

- enumeration of the requirements for smart-home IoT systems (Section II);
- architecture of the system: a combined BLE and 6LoWPAN network of heterogeneous devices (Section III);
- analysis of the challenges to support the requirements and corresponding lessons learned (Section IV);
- last but not least, the implementation of the system[2].

II. Requirements

The SPHERE system for environmental and wearable data collection has to satisfy the following requirements (in approximate order of importance):

- ***R1:*** *Core functionality.* The IoT network has to collect readings from heterogeneous, spatially separated wireless sensor nodes, some of which are mobile, and deliver the readings to a Linux host with cumulative data rate up to 50 kbps.
- ***R2:*** *Security.* The health-related data is confidential and must be encrypted when transported over the air.
- ***R3:*** *Energy efficiency* for parts of the system. The battery-powered environmental sensors have to support hard constraints on battery lifetime: 1 year minimum, bounded by the maximal duration of SPHERE data collection experiments.
- ***R4:*** *Time synchronization.* Collected data should be accurately time-stamped with 10 ms or higher granularity and a comparable maximal network-wide timing error.
- ***R5:*** *Robustness.* The system should be able to self-repair & self-heal in order to function under limited maintenance, as the number of visits in people's homes should be minimized.
- ***R6:*** *Reliable data delivery.* The system should provide close-to-100 % packet delivery rate, assuming typical wireless link qualities for short links in indoor home environments ($\geq$50 % packet reception rate) and assuming no hardware and network deployment problems.
- ***R7:*** *Predictability.* The users of the system should be able to know before actually deploying the system what are the

[1] http://irc-sphere.ac.uk/

[2] We plan to release the SPHERE IoT code at https://github.com/IRC-SPHERE. Bugfixes are continuously contributed to the mainline Contiki.

978-1-5090-5279-6/17/$31.00 ©2017 IEEE

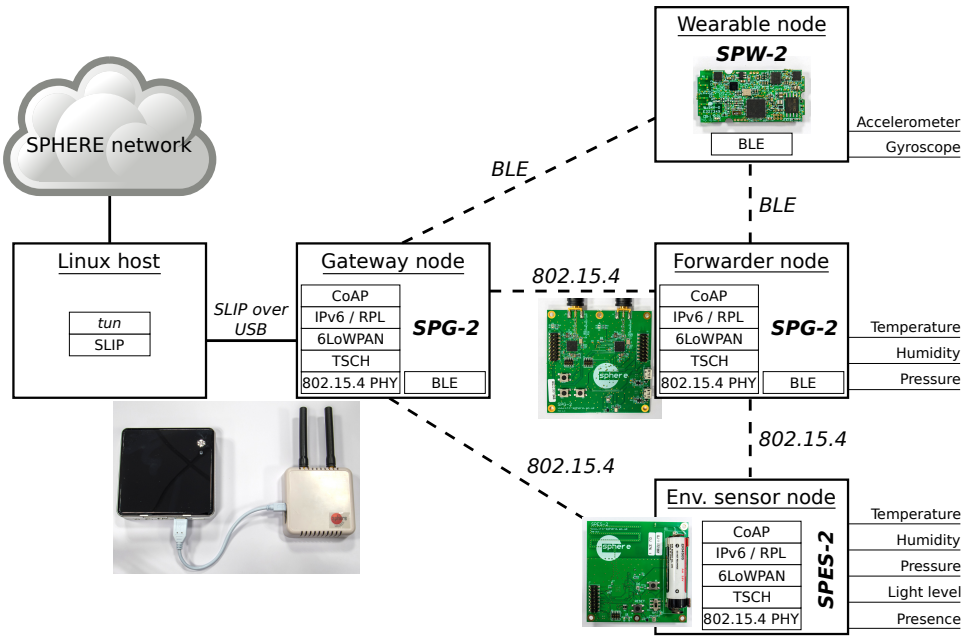

Fig. 1: **The SPHERE IoT architecture.**

minimal link-qualities, maximal component failure rates, etc., for the system to provide acceptable performance.

- ***R8:*** *Visibility.* The system should provide a continuous stream of run-time performance statistics.
- ***R9:*** *Interoperability.* If possible, the system should stick to existing low-power IoT standards and protocols in order to (1) be amenable to future extensions with third-party components; (2) reduce the learning time for new personnel.

The requirements are fairly typical for low-power IoT systems. Note that additional requirements will appear if the SPHERE system is extended with new applications; for example, having low latency would become an important requirement if real-time human pose estimation were added as an application.

III. System Overview

There are two main data sources (***R1***) in the system: environmental sensors and on-body sensors (Fig. 1). The environmental sensors are attached to SPES-2 and SPG-2 nodes [3], the on-body sensors: to SPW-2 wearable nodes [4]. All the nodes are based on TI CC2650 System-on-Chip [5], selected because its flexibility: it supports both IEEE 802.15.4 and Bluetooth Low Energy (BLE) communication stacks. The CC2650 also supports AES-128 hardware acceleration, which the nodes use to encrypt the packets sent over the air (***R2***).

The network is dual-stack. The stationary part uses open source 6LoWPAN stack that is based on IEEE 802.15.4 physical layer and implemented in Contiki, a major open-source IoT OS. The wearable node code is based on Texas Instruments RTOS and use BLE, as this communication standard offers better support for mobility compared to 6LoWPAN / RPL. The data produced by the wearable nodes is picked up by the dual-radio SPG-2 nodes and forwarded by the IEEE 802.15.4 network stack, as this standard offers a more mature support for multihop compared with BLE.

The environmental sensors are battery-powered (***R3***) low-rate data sources, sampling with $\leq$0.2 Hz frequency, while the wearable nodes produce much more data: they sample the 3-axis accelerometer with 12.5 or 25 Hz frequency depending on configuration settings. Project requirements mandate that all the raw data is collected, leading to a high datarate in the IEEE 802.15.4 network: up to tens of packets per second. At the MAC layer, we use Contiki implementation [6] of the IEEE 802.15.4-2015 TSCH protocol to efficiently manage the medium access, avoid packet collisions, and synchronize time (***R4***). As compared to CSMA-based low-power IoT MAC protocols, TSCH provides high reliability (***R6***) and predictability (***R7***) through scheduled operation and channel-hopping [7], exploiting frequency diversity to fight external interference and multipath fading.

At the network layer, we use IPv6 on top of TSCH as described in the emerging 6TiSCH standards [8]. This allows to reuse (***R9***) RPL for routing and 6LoWPAN for IPv6 header compression. The application layer in turn is based on CoAP, which supports both the client/server traffic pattern through client-initiated queries for server resources, and the

TABLE I: SPHERE code size (w/o comments & empty lines).

Component	Lines of code	Language
CoAP resources	2436	C
Other application code	1059	C
Newly-developed sensor drivers	946	C
Newly-developed SoC drivers	633	C
802.15.4 / BLE bridging code	342	C
Additions to TSCH	1374	C
Additions to SLIP	265	C
The gateway script	3199	Python

subscribe/notify traffic pattern through the "observe" feature. While each observation of a resource must be similarly initiated by a query from a client, subsequent data coming from that resource is autonomously published by the server, either in event-driver or periodic fashion. The sensors supported by the nodes as well as monitoring data are all exported as observable CoAP resources with remotely controllable reading period and other settings. We opted for CoAP as opposed to MQTT or HTTP because CoAP uses UDP as the transport protocol, while the latter two require TCP, which is known to suffer from performance problems in low-power wireless networks.

SPES-2 and SPG-2 nodes have two hardware watchdog timers, capable of rebooting the system in case of a software failure (***R5***): one that part of the CC2650 SoC, and one that is a dedicated chip mounted on the printed circuit board.

Table I illustrates the approximate development effort for the SPHERE IoT software components. Using a SoC already supported by the Contiki OS reduced our total effort significantly; several existing TI SensorTag sensor drivers were also reused. Using an existing network stack was an even bigger gain; while the application level code for CoAP resources is high is size, it is low in complexity. The main effort here was to port TSCH to the CC2650 hardware platform, and extend TSCH to support our scheduling requirements and to collect more performance metrics (***R8***).

The data coming from the sensor network is received by the SPHERE gateway script running on a Linux host. The script implements a reliable version (***R6***) of the SLIP protocol, CoAP client & server, bidirectional CoAP / MQTT translation, and an HTTP server as an alternative option for data access and node control. We use `aiocoap`, a CoAP library based on Python 3 asynchronous I/O. It has coroutine-based API and consequently offers the simplicity of thread-like syntax without the complexities of locking; a design and benefits similar to those of the Contiki protothread API.

Any changes in the system are first prototyped using a mock-up platform called *Z1x* in the Cooja simulator, which is also used for regression tests (***R5***) of the full IoT system.

IV. Lessons learned

A. Energy efficiency

1) Partitioned Power Management: Older chips traditionally had preset "power modes" (or profiles) that were built into the chip. For example, the TI CC2538 SoC [9] has 6 power consumption profiles: "Active", "Sleep mode" and four additional modes called "Power Mode 0" (PM0), ..., PM3. In all PMs 0-3 the MCU is in deep sleep. Each mode offers incremental power savings by turning off additional chip peripherals, but also adds limitations in terms of wake-up sources. For example, in PM0 it is possible to leave some clocks running, whereas in PM1 all system clock sources are powered-down. PM3 is the mode that achieves the lowest power consumption, but the chip can only be woken up by an externally triggered GPIO interrupt. To enter a low power mode, a developer has to follow these steps:

1) shut down / configure peripherals (LEDs, sensors, etc.);
2) select one of the pre-defined power modes by writing some in a hardware register;
3) configure wakeup sources;
4) enter this power mode.

This approach is relatively simple to support in software, but does not allow the developer to power on/off individual chip peripherals.

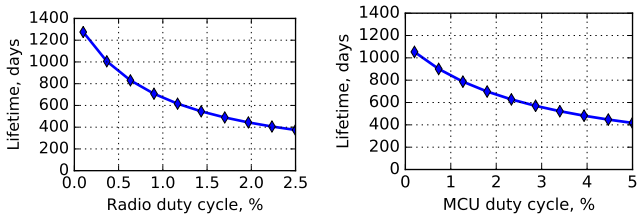

Fig. 2: **Estimated node lifetime,** assuming a 2600 mAh battery from which 90 % of power can be. *Left:* variable radio duty cycle, MCU duty cycle fixed to 2.0 %. *Right:* variable MCU duty cycle, radio duty cycle fixed to 1.0 %.

Compared to their predecessors, chips of the TI CC26xx and CC13xx family have a much more sophisticated power management module [10]. Power profiles are no longer predetermined. Instead, the chip is partitioned into Voltage Domains (VDs): MCU and Always-On (AON). Within each VD reside Power Domains (PDs) and within each PD reside digital modules. The clock to each module can be gated individually. For example, the UART module resides within the "SERIAL" PD, which is within the MCU VD. The radio module resides on a separate domain that can also be controlled individually.

This design gives very fine-grained flexibility to the developer: it is possible to gate clocks to digital modules, or power-down digital modules and PDs on an individual basis. Additionally, some digital modules have retention that can be disabled by the developer.

Contiki's port for MCUs of the CC13xx/CC26xx family takes full advantage of the chip's fine-grained power-management capabilities. Depending on requirements, it is possible to enter a state of extremely low energy leakage, or to sacrifice some energy consumption in order to keep a peripheral powered on (*e.g.*, to use it as a wakeup source). The SPHERE code uses Contiki's low-power module (LPM) and achieves as little as 3 μA power-down current on SPES-2 nodes (while retaining RAM and keeping some sensors, such as the PIR, active), leading to a high system lifetime (Fig. 2).

However, CC26xx's extremely low power consumption comes at a cost. Powering-off peripherals and PDs results in an increase of the time required to wake-up. Of particular importance is the state of the 24 MHz crystal oscillator (XOSC), which is required for radio operation. The chip will not enter deep sleep unless the XOSC is powered-down, but the XOSC has a relatively long startup time. A trade-off can therefore be made when the anticipated duration of low-power state is short. Leaving the XOSC powered will have a negative consequence on power consumption, but the wake-up sequence can be sped-up by an order of milliseconds. Conversely, for longer

periods of low-power operation, it makes sense to power the XOSC off in order to achieve lowest possible consumption. Contiki's wake-up sequence takes into consideration the long XOSC startup time. The sequence requests a XOSC power-up immediately after a wake-up event occurs. The sequence then continues with powering-up and initializing other modules, while the XOSC is powering-up in parallel. It is only required to block waiting for the XOSC before calibration of the radio's frequency synthesizer. By parallelizing XOSC power-up and the chip's startup sequence, the total wake-up time is reduced.

2) Asynchronous Sensor Drivers: Many platforms currently distributed as part of the Contiki release use synchronous sensor drivers. To summarize, when the user application requests a sensor reading, the driver will power-on the sensing element (*e.g.*, a light sensor) and will block waiting for a reading to become available. This is adequate for sensing elements that have a very small startup time, for example in the order of μs.

However, the SPHERE platform uses a number of sensors with long startup times. In the example of the light sensor, a Texas Instruments OPT3001 [11], the time between the power-on and the availability of a reading can be up to 880 ms. A synchronous sensor driver will therefore block waiting for up to this time interval waiting for a reading to become available. This has a number of disadvantages:

- All other operating system tasks will be delayed until the driver has finished sampling the sensor for a reading. This can delay, for instance, the transmission of a network packet.
- While the driver is waiting for the reading to become available, the MCU is powered on and operating at full speed, wasting energy.

To overcome this, SPHERE's sensor drivers are asynchronous and have been built-on Contiki's event-driven design. Sampling a sensor is undertaken by following the procedure below:

1) The application requests a sensor value.
2) The driver triggers the sensor's power-up sequence and returns immediately.
3) The driver monitors the sensor for availability of a new reading. When a new reading is available, the driver latches it and powers down the sensor.
4) The driver uses a broadcast event to notify all processes that a new reading is available.
5) The user application receives this event and retrieves the reading from the driver.

This design allows other tasks to execute or even the chip to enter a low-power state while the sensor is powering-up, therefore saving considerable energy.

B. Memory management

TI CC2650 does not have a memory mapping unit (MMU) or a memory protection unit (MPU), therefore the operating system is not able to recognize memory access violation or to isolate different components from each another. While some of ARM Cortex-M3 based microcontrollers have an MPU, CC2650 is not one of them [5], citing "cost constraints". Furthermore, Contiki and other state-of-art IoT OS do not have software support for this feature.

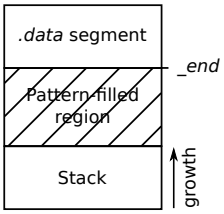

Fig. 3: Stack and *.data* segment layout on sensor nodes.

CC2650 has 20 kB RAM, which is not a large amount for a 32-bit system, taking into account the Contiki OS footprint and the large RAM requirements for network neighbor and routing tables [12]. Not having a neighbor or a routing table entry for an active neighbor can severely reduce the performance and stability of the network [13]; consequently, it is optimal to reserve as many entries as possible, aiming for an entry for each device in the network.

Contiki does not use dynamic memory allocation; the end of the allocated memory (`.data` and `.bss` segments) on many platforms is marked by a linker symbol named `_end` (Fig. 3). However, at the end of the physical memory space there is a stack region that "grows" in the direction of lower addresses. Therefore there is a real possibility of the stack region overlapping with the `.data`/`.bss` segments, which due to the lack of MPU cannot be detected in hardware. Nevertheless, most Contiki HW platform drivers do not include software-based stack overflow protection. The sole precaution CC2650 platform offers is "reserving" 256 bytes for the stack during the linking stage.

As a result, during the testing the system experienced stack overflow on the gateway node, which manifested itself in a cryptic fashion: as random reboots due to the the watchdog timer expiring. The real cause of that was memory corruption, and it was fixed by reducing static memory usage. We found that 256 bytes are not sufficient for the stack: using a pattern-fill technique (Fig. 3), we were able to measure the real stack usage of the SPHERE networking stack during a pilot deployment and found it to be as high as 1820 bytes on one node; in particular, the Contiki CoAP implementation is characterized by heavy stack usage. Our stack usage estimate is now one of the stats periodically collected from the network.

C. Robustness and self-repair

The deployed system has to survive occasional software, hardware, and power problems by reliably coming back to a good state. Contiki has support for a watchdog timer, which reboots the system when software becomes stuck in a loop. However, we faced several problems with this functionality.

Firstly, this watchdog timer is necessarily disabled during the so-called deep-sleep mode of CC2650, in which the MCU is not active. The CC2650 port of the Contiki was known to suffer from occasional "deep freezes" when the system does not wake up from deep sleep. After code changes in the power-management module, this bug has not been observed anymore. Nevertheless, there was not a sufficient number of devices or time to verify that with high certainty before the final hardware

revision of SPHERE device. Even a low-frequency failure (for example, once per 2-3 years on each node) would still lead to unacceptably high number of maintenance visits in a 100-home, 1000+ node deployment. Secondly, the watchdog reboot of the system did not always bring the system back to a good state. In particular, if just one MCU rebooted from the two MCU present on SPG-2 nodes, software initialization of the SPI connection between the two sometimes would fail.

We solved the issues by adding an external, very low-power watchdog (Texas Instruments TPL 5010, 35 nA operating current) to the system that operates independently of the MCU and reboots *both* MCUs on the SPG-2 nodes. This shows that real-world IoT systems can immensely benefit from software/hardware co-design: in this case, rather than spending hundreds of hours verifying that the software bugs have been fixed for good, we introduced a fail-safe that would restore the operation even if the software is not perfect.

D. Reliability

The Contiki implementation of the IPv6-over-TSCH stack [6] is known to provide high end-to-end packet delivery rate (PDR): 99.99 % and more, even in challenging conditions [7]. Nevertheless, in our first testbed experiments in the SPHERE house [1], a little-interfered environment we only received 99.9 % PDR, *i.e.*, had a ten times higher proportion of dropped packets. Upon inspection, the problem was even worse on the gateway node which did not use wireless connections *at all*, but only managed to deliver 99.7 % of packets. The issue was caused by packet loss on the wire, on the serial-over-USB interface and only showed up in real-world conditions.

Contiki does not offer any reliability features for the SLIP protocol. Besides the packet loss, this might occasionally lead to corrupt data being collected, which would have adverse impact on the main results and goals (*e.g.*, human activity analysis) of SPHERE. We ameliorated the issue by (1) appending 16-bit CRC and 8-bit sequence number for packets sent over serial line and (2) requesting that packets with invalid CRC are retransmitted by sending a NACK upon their reception. Our experiments show that buffering 4 most recently transmitted packets is sufficient to achieve 100 % PDR over the SLIP.

We faced an additional serial port related problem during the integration testing phase, when an unrelated Node-RED (a popular visual programming tool for IoT) based component running on the Linux host would occasionally try to read data from the gateway node's serial connection, corrupting the data received by the gateway script.

Recently, connecting IoT to a LAN through an Ethernet connection has become a more popular option. Ethernet offers reliable, high-speed connectivity, a very-well understood multiple-access MAC level (instead of the point-to-point RS-232), and due to latching RJ45 connectors, safety from accidental physical disconnections. In retrospect, connecting the border router through Ethernet would have been a better idea, as it would not only make the system more robust, but also save a week or more of the development time.

E. Other issues

Security The Contiki OS offers IEEE 802.15.4 compatible link-layer security; however, a key distribution scheme is missing. In the SPHERE project, we have a centralized database of pre-shared keys, which are manually programmed ("flashed") in the sensor nodes. This adds some administrative overhead, as a single project wide binary image cannot be used. Instead, each device has to be separately prepared before its deployment. This scheme is also much more limited, when compared to *e.g.*, the dynamic establishment of session keys in the WirelessHART industrial standard.

Time scheduling In line with the very limited API that Contiki offers for memory management and energy management between components, there is no system-level API to reserve blocks of time or even to disable interrupts in a platform-independent way. One issue we faced was related to a water flow sensor driver developed by an undergraduate intern. Initially, the driver blocked for 400 ms to count rising edges on the GPIO pin. With high rate TSCH and BLE traffic in the network, this would lead to either (1) many missed packets (if the driver disabled interrupts) or (2) invalid sensor readings (if system interrupts were able to pre-empt the operation of the driver). We reduced the time-to-read to 20 ms, ending up with a lower-granularity, but more accurate sensor.

Visibility Contiki is mainly used as a research OS, and shaped primarily by the expectations and assumptions of the research community. For example, one frequent assumption is that over-the-serial logging is available, which is not the case in real deployments. Even this logging is limited and not enabled by default – for example, oversized packets are silently dropped by the stack without any indication.

The SPHERE deployment requires continuous monitoring for alerts, debugging, as well as performance analysis and replication of interesting situations in the lab based on the real-world network performance traces. We spent significant effort to introduce additional network performance statistics (for example TSCH per-channel statistics) and to make them available as CoAP resources. These resources are queried every 30 min by the SPHERE gateway script and provide statistics about per-channel per-neighbor packet reception rates, TSCH time synchronization performance, background noise RSSI levels, node energy usage estimates, stack usage, etc.

V. Discussion

What has (and what has not) changed? Software engineering for wireless sensor networks (and hence low-power IoT) is difficult as this field has both the complexity of embedded system programming and the complexity of distributed system programming, areas considered difficult on their own. In a well-known paper from 2006, Langendoen *et al.* talk about the practical difficulties in deploying a sensor network [14]. Similarly to this work, they used what at the time was a state-of-art operating system and protocol stack.

It is fair to say the research community has made large progress on solving some of the problems. In particular, compared to the low-power CSMA used by Langendoen *et*

al., the MAC layer we're using (TSCH) is: (1) low-power (<1 % [7] *vs.* 11 % radio duty cycle [14]), (2) highly reliable (has >99.99 % PDR [7]) in varied environments thanks to the channel hopping feature, (3) collision-free if required, more predictable, and supporting higher data rates due to its TDMA scheduled nature, (4) standardized [15] and interoperable.

On the other hand, many of the problems have *not* been solved. Similarly to Langendoen *et al.*, our system experienced random, difficult-to-debug watchdog reboots due to various reasons. Although routing protocols have advanced a lot, for example, in terms of standardization, the reliability of the modern 6LoWPAN / RPL network stack is still badly affected by the limited number of neighbor and routing table entries [13]. This paper once again raises the issue of the scarcity of easily available statistics and tha lack of visibilty of the network state. Finally, we are still lacking component sandboxing in the state-of-art IoT operating systems.

Everything is connected. One overreaching theme in Section IV is the lack of component isolation. Unwanted interactions between components have negative impact on energy consumption, cause random failures because of memory overflows, initialization failures after reboot, and resource scheduling problems. The unavoidable high coupling between components puts a very high requirement on the developers of the system. Since the components cannot be safely sandboxed, for *each* of the components used by the system, a person with expertise in the internals of that component must either be part of the team or easily available for consultations. Furthermore, some bugs only became obvious when an expert in several, functionally unrelated parts of the system was involved.

Going higher? During the last decade, a lot of research in software development for low-power IoT has proposed new, higher-level software development abstractions, languages, and tools [16] [17] [18]. Their main promise is to reduce the risk for the developers to "shoot themselves in the foot", and their approach is to reduce the number of unsafe, low-level operations they need to perform, or to prohibit some operations altogether (often implicitily, by restricting API capabilities).

However, while the approaches and tools are helpful, it is clear that at the moment they cannot solve all of the problems discussed in this paper. Typically they do not attempt and often *cannot* deal with problems that arise at the OS level, *e.g.*, lack of API for resource management, or due to hardware restrictions, *e.g.*, lack of MPU. (It is interesting to note that there has been some regress in terms of the API offered by C-based operating systems such as Contiki, compared to the API of the component-oriented TinyOS [19].) Therefore the user of these high-level abstractions cannot be reliably shielded from problems appearing at the lower-levels, resulting in so-called "leaky abstractions". Furthermore, some of the approaches can have a detrimental effect: for example, because of the additional memory usage required to support some of the high-level abstractions the stack overflow issue would become *more* likely if they were used.

VI. Conclusion

We have presented the requirements and the architecture of the SPHERE IoT system. The focus of this paper is on some of the lessons learned and challenges faced during the development phase, namely, the challenges to achieve energy efficient operation in low-power modes, to detect and avoid stack memory overflows, to recover the system after reboots, to have reliable data delivery throughout the system, to avoid time scheduling conflicts, and to have high visibility of the state of the network in real deployments. We identify high coupling between unrelated components as the main cause of several problems, manifested in the form of unpredictable and undesirable interactions.

Acknowledgments

This work was performed under the SPHERE IRC funded by the UK Engineering and Physical Sciences Research Council (EPSRC), Grant EP/K031910/1.

References

[1] P. Woznowski *et al.*, "SPHERE: A Sensor Platform for Healthcare in a Residential Environment," in *Designing, Developing, and Facilitating Smart Cities: Urban Design to IoT Solutions*. Springer, 2017.
[2] C. Bormann *et al.*, "RFC 7228: Terminology for Constrained-Node Networks," https://tools.ietf.org/html/rfc7228.
[3] X. Fafoutis *et al.*, "Demo: SPES-2 – A Sensing Platform for Maintenance-Free Residential Monitoring," in *EWSN 2017*.
[4] X. Fafoutis, B. Janko *et al.*, "SPW-1: A Low-Maintenance Wearable Activity Tracker for Residential Monitoring and Healthcare Applications," in *Proc. Int. Summit on eHealth (eHealth 360)*, 2016, pp. 294–305.
[5] "CC2650 SimpleLink Multistandard Wireless MCU," SWRS158, Texas Instruments, 2015.
[6] S. Duquennoy, A. Elsts, B. A. Nahas, and G. Oikonomou, "TSCH and 6TiSCH for Contiki: Challenges, Design and Evaluation," in *IEEE DCOSS*, 2017.
[7] S. Duquennoy, B. Al Nahas, O. Landsiedel, and T. Watteyne, "Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH," in *ACM SenSys*, 2015, pp. 337–350.
[8] "IPv6 over the TSCH mode of IEEE 802.15.4e IETF working group," https://tools.ietf.org/wg/6tisch/.
[9] "CC2538 Powerful Wireless Microcontroller System-On-Chip for 2.4-GHz IEEE 802.15.4, 6LoWPAN, and ZigBee®Applications," SWRS096D, rev. D, Texas Instruments, 2015.
[10] "CC26xx SimpleLinkTM Wireless MCU Technical Reference Manual," SWCU117A, rev. F, Texas Instruments, 2016.
[11] "OPT3001 Ambient Light Sensor (ALS)," Texas Instruments, 2014.
[12] G. Oikonomou and I. Phillips, "Experiences from porting the Contiki operating system to a popular hardware platform," in *IEEE DCOSS*, 2011, pp. 1–6.
[13] O. Iova, G. P. Picco, T. Istomin, and C. Kiraly, "RPL, the Routing Standard for the Internet of Things... Or Is It?" *IEEE Communications Magazine*, vol. 17, 2016.
[14] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture," in *IEEE IPDPS*, 2006.
[15] "IEEE Standard for Local and metropolitan area networks—Part 15.4," IEEE Std 802.15.42015, 2015.
[16] L. Mottola and G. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, Apr. 2011.
[17] A. Elsts, J. Judvaitis, and L. Selavo, "SEAL: a Domain-Specific Language for Novice Wireless Sensor Network Programmers," in *EUROMICRO SEAA*, 2013, pp. 220–227.
[18] A. Elsts, F. H. Bijarbooneh, M. Jacobsson, and K. Sagonas, "ProFuN TG: A tool for programming and managing performance-aware sensor network applications," in *IEEE LCN*, 2015, pp. 751–759.
[19] P. Levis *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient intelligence*. Springer, 2005, pp. 115–148.