Lessons from Communication Problems that Nearly Jeopardized Development of Hardware-Software Support for a 1000-Device IoT Testbed

Mateusz Banaszek, Inga Rüb, Maciej Dębski, Agnieszka Paszkowska, Maciej Kisiel, Dawid Łazarczyk, Ewa Głogowska, Przemysław Gumienny, Cezary Siłuszyk, Piotr Ciołkosz, Jacek Łysiak, Wojciech Dubiel, Szymon Acedański, Przemysław Horban, Konrad Iwanicki
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw
{m.banaszek, i.rub, iwanicki}@mimuw.edu.pl

Abstract

While prototyping devices dedicated for a 1000-node low-power wireless networking testbed, we encountered over a dozen nontrivial technical problems. This paper analyzes a few selected ones, which we faced during development of the hardware-software support for two different communication channels for supervising experiments on the testbed. The problems arose despite our employing popular standards and reputed components, minimizing risky features, and following modern quality assurance practices. Moreover, if it had not been for our emphasis on dependability, they would have likely passed undetected and doomed our project. In this light, we believe that the presented lessons we learned the hard way when debugging the problems may be of interest to the community.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications*

General Terms

Design, Experimentation, Management, Reliability

Keywords

testbed, experimental platform, prototyping, Ethernet, WiFi, low-power wireless network, Internet of Things

1 Introduction

In January 2016, we started a project whose goals involved, among others, building a 1000-device one-site testbed for experimentation with low-power wireless communication protocols for the Internet of Things (IoT). By October 2019, all devices had been manufactured, tested, and delivered, and we deployed over 75% of them. With those devices operational and some experiments on them already completed, we have every reason to believe that the undertaking may indeed ultimately be successful. Nevertheless, throughout the duration of the project, that was not obvious at all, considering the multitude of problems that we faced.

Although the scope of the undertaking entailed problems of virtually every conceivable nature, for this paper we selected just a few communication-related technical ones that we had encountered when prototyping the devices. We believe that an analysis of the problems may be of interest to the community for at least two reasons. They emerged when designing rather standard hardware-software functionality that was going to be used in a fairly typical manner, as such being completely unpredicted. At the same time, if they had passed undetected or we had not emphasized certain requirements, they would have likely doomed the entire project.

To give more context, serving as a platform for experimentation, our devices were expected to be extensively used. Their low-power wireless system-on-chips (SoCs) would be frequently reprogrammed with custom images whose operation would be monitored and altered in real time. Existing testbeds addressed that issue by providing remote supervision of the experiments by means of a *backbone control network*. Such networks were built out of ordinary computers or embedded ones, yet powerful enough to run classic operating systems and protocol suites. In an extreme case, each computer hosted only one low-power wireless SoC. That was also the approach we followed: combining in a single device an embedded computer for control and a board with a low-power wireless SoC for actual experimentation. Yet, due to the aforementioned and some other constraints of the project, we were unable to adopt complete off-the-shelf devices to this end, thereby being forced to design dedicated ones.

When deciding upon the hardware-software support that the embedded computers should have provided for the control infrastructure, apart from other constraints such as a tight budget and limited facilities of the target deployment site, two issues were of particular importance: the scale of the testbed and highly limited post-deployment access to individual devices. The poor physical access implied that the devices had to operate autonomously for extended periods of time. The embedded computers had to be responsive over the network in the presence of software bugs, potential mis-

International Conference on Embedded Wireless Systems and Networks (EWSN) 2020
17–19 February, Lyon, France © 2020 Copyright is held by the authors.
Permission is granted for indexing in the ACM Digital Library
ISBN: 978-0-9949886-4-5

use, or after global calamities like power or network outages. When it comes to the scale, in turn, in the case of a smaller testbed, even a major malfunction might be often overcome by replacing faulty components or even rebuilding parts of the infrastructure. In our case, that was hardly viable. In fact, the sheer scale of testbed made it challenging to merely collect logging statements from the low-power wireless SoCs, not to mention upgrading the operating system image on the embedded computers. Consequently, throughout the entire project we emphasized *dependability*: from individual components, to subsystems, to the entire testbed. In particular, we strove to ensure that the control network formed by the embedded computers of the devices would be truly reliable, much like in many industrial IoT applications [7].

The rest of this paper presents an analysis of selected major technical problems that arose when trying to achieve this goal. We start in Section 2 by discussing the hardware-software support for the control networking that our devices were expected to provide. Then, in Sections 3 and 4, we analyze the problems and offer short stories about their debugging process, the results of our investigation, and universal advice. Finally, we conclude in Section 5.

2 HW/SW Support for the Control Network

The embedded computer and the low-power wireless SoC were the two main components of our devices, dubbed *Cherry devices* (see Fig. 1). The selection of those particular components was motivated directly by the functionalities they provided and protocols they supported,[1] but their power consumption and our budget also played major roles.

In case of the embedded computer, the choice was mainly driven by our design of the control network. Ethernet was decided to be the primary communication channel due to its stability and throughput. The embedded computer thus had to support Ethernet out of the box. Moreover, we wanted to base our control network on an already existing LAN infrastructure at the deployment location, but free Ethernet sockets were hardly available there. Buying a few hundred network switches was not an option due to our strict budget. Therefore, we chose an embedded computer that provided 2 Ethernet ports and an integrated switch that could be configured to relay network traffic between those ports and the device itself. In that way, at each location where our Cherry devices were deployed, only one free Ethernet socket was required: one device had to be connected to it and other nearby devices could be connected to one another in a chain-like way.

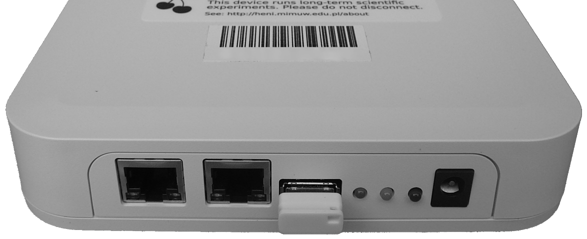
CherryMote
This device runs long-term scientific experiments. Please do not disconnect.
See: http://heni.mimuw.edu.pl/about

Figure 1. Cherry device

[1]To avoid positive/negative marketing of their manufacturers, we do not provide names or detailed specifications of these components.

Although the selected embedded computer had built-in Ethernet support, it did not have Ethernet ports. Only bare electrical pins were provided. For that reason we had to design a printed circuit board (PCB) that connected the embedded computer with the 2 Ethernet ports located at the front of Cherry device. However, the manufacturer of the computer provided also an evaluation board to facilitate quick prototyping with its products. Not only had the evaluation board exactly 2 Ethernet ports, but also electronic schematics diagrams presenting its design were publicly available. That way we could preliminarily test our design using those evaluation boards and then simply copy the Ethernet section to our PCB directly from the original schematics.

With the high dependability in mind, we decided that our control network needed a backup communication channel. In some locations it might have not been possible to connect Cherry devices to the LAN directly. Furthermore, someone could (accidentally or deliberately) disconnect our Ethernet cables relatively easy. Consequently, in addition to Ethernet, we decided to employ WiFi as a failover communication channel in the control network. For that reason the selected embedded computers were based on a Ralink RT5350 WiSoC [10] supporting the WiFi 802.11n standard. That standard was required by the expected volume of traffic within the control network. Furthermore, the embedded computer already had an on-board WiFi antenna, so no hardware modifications were required.

With both technologies on board, Cherry devices were to communicate via Ethernet always when it was possible, and to automatically switch to WiFi when the former route was no longer accessible. Importantly, we did not want Cherry devices to be connected to any external WiFi network but to autonomously form a dedicated one, in which Cherry devices connected via Ethernet acted as access points.

Regarding software, the embedded computers were powered by OpenWrt [1]. It was a natural choice for us since that Linux-based open-source operating system targeted exactly embedded and network devices. It could be found in many home WiFi routers. Moreover, it was recommended by the manufacturer of the embedded computer. The manufacturer even provided additional modifications and configurations for those devices. Since OpenWrt supported an extensive network configuration, to achieve the required behavior of the control network (Ethernet being the primary communication channel, WiFi being the fallback, relaying network traffic both via cable and wirelessly, etc.) all we had to do was to set proper options in OpenWrt configuration files.

3 Ethernet Problems

Employing Ethernet communication into Cherry devices required preparing a printed circuit board to connect the embedded computer with Ethernet ports and customizing OpenWrt network configuration (cf. Section 2). That process went smoothly, so we moved to evaluating our solution.

3.1 The Problem

When first prototype devices were produced, their Ethernet ports were tested. We observed no issues: Cherry device plugged to an off-the-shelf home router successfully established a connection. The *chaining* feature (cf. Section 2) was also simply tested: a laptop connected to Cherry device also established an Internet connection after a short while. Since the design of Cherry device's PCB strictly followed the original evaluation board, which we had used earlier to prototype the *chaining* feature, we assumed that no further Ethernet-related tests were required. However, when we later started conducting integration tests by assembling a small network from a succeeding version of our prototypes, some Cherry devices could not connect to the Internet!

Starting with simple tests it quickly became obvious that the problem was occurring only when two Cherry devices where connected to each other. Ethernet ports' LEDs were flashing abnormally and system logs where full of unexpected entries. However, connecting Cherry device to a laptop or a home router still worked flawlessly as it had before. Moreover, connecting two Cherry devices together was sometimes succeeding after short waiting periods. It seemed to us that the occurrence of the problem was correlated somehow with a selection of used Ethernet ports (2 devices with 2 Ethernet ports means 4 possible network configurations) and a particular cable model. However, when the same experiments were conducted using the original evaluation boards, everything worked smoothly in all possible configurations.

3.2 Debugging the Problem

To facilitate further debugging, a *'very bad cable'*[2] was found: it was so long that Cherry devices connected via it could never establish a connection, even regardless of selected ports. Meanwhile, a few laptops were tested also with the *'very bad cable'* and they all were able to connect to a variety of networks without any problems. That result disproved our guess that only certain Ethernet ports were flawed and allowed us to repeatedly reproduce the issue.

Then our suspicion was that there was a problem during an autonegotiation procedure. According to the Ethernet standard [6], it is automatically performed by just connected devices to choose right transmission parameters (speed, duplex mode, flow control, etc.). Based on symptoms we suspected that Cherry devices' network driver could not determine the right transmission speed. To verify that conjuncture an experiment was run in which all connection parameters were pre-configured so that the autonegotiation phase was not needed. However, it turned out that our guess was incorrect: even with those settings establishing a connection was still constantly failing.

Finally, we decided to use an oscilloscope to examine electrical signals transmitted by Cherry devices over an Ethernet cable. Even first measurements explained why they could not establish a connection: it was clearly visible from the eye diagram that the signal was heavily distorted (see Fig. 2). A corresponding eye diagram of a connection with the evaluation board looked significantly cleaner (see Fig. 3). It indicated that the problem was related not to software but to hardware. Moreover, since that Cherry device and that evaluation board employed identical embedded computers, we learned that the problem laid in the PCB which connected the embedded computer with Ethernet ports within Cherry device. To find out what we had done wrong when preparing that PCB we had to verify everything from the beginning.

Once again our design was compared with the corresponding part of the electronic schematic diagram of the evaluation board. They looked exactly the same. Since the design was correct, maybe there was an issue with the actual PCB, e.g. it had not been manufactured properly? However, it looked that not only had it been produced correctly, but it was also manufactured accurately according to our design and it was the right latest version of our design. Maybe we had incorrectly designed traces on the PCB? There are well-known recommendations [4, 9, 5] describing how Ethernet traces should be routed on a PCB: their widths should not vary, Tx+/Tx- and Rx+/Rx- pairs should be routed together, they should be separated by a proper distance, etc. Not following them may result in interference between individual signals. Every rule that could be verified was verified and also there no issues were noticed. We even analyzed whether an antenna of Cherry device's low-power wireless SoC was placed in a way minimizing possible interference, but it seemed to be mounted in a right place and direction. Looking for any clue on what could be wrong, once again all values and parameters of related electronic components were reviewed, but they still looked correctly.

At that point we were behind our initial schedule, and we still knew only that while Cherry devices did not work properly, the original evaluation boards did. So there must have been a difference. Not having any better idea how to find it, we compared both devices by just carefully looking at them: there it was! Eight resistors were not present on the evaluation board but they were mounted on Cherry device's PCB. Removing them from our design solved the problem! After just desoldering them Cherry devices were establishing Ethernet connection immediately, even when they were connected via the *'very bad cable'*. Measurements with an oscilloscope also confirmed that not mounting those eight resistors significantly improved quality of Ethernet signals.

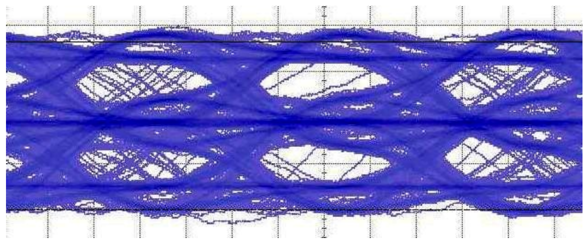
Figure 2. Oscilloscope measurements: Ethernet communication with Cherry device

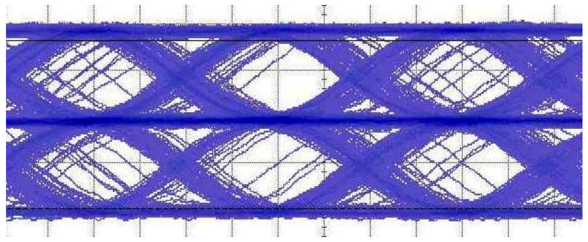
Figure 3. Oscilloscope measurements: Ethernet communication with the evaluation board

[2]It was a worn (but still functional) long (over 10 meters) shielded Ethernet cable.

3.3 Why Did We Make the Mistake?

Initially we took a guess that the electronic schematic diagrams used by the manufacturer of the evaluation board differed from the published ones. Only a few months later, during a post factum analysis, we did notice that those resistors were marked with `NA`. The meaning of `NA` was not explained on the schematics but, judging from the context, it probably indicated that those components should not have been mounted on a PCB (`Not Applicable`?). Although it is a common practice to place some components on schematics and do not mount them on an actual board, they are usually marked with `DNM` (Do Not Mount), `DNP` (Do Not Populate or Do Not Place), simply `Not Mount` or something similar to these. No one from our team had seen the `NA` mark ever before. Moreover, the published schematics provided for those resistors all parameters that were needed to choose the right ones. Additionally, capacitors connected via those resistors to Ethernet signal traces were present both in the design and on the actual evaluation board, despite not being connected to anything in that case. The resistors seemed to be related to analog signal, so having found the solution we did not investigate that issue further. However, one could find similar sections (with such resistors) in many designs related to Ethernet connections [4, 11, 8].

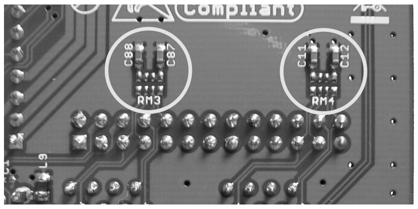

Figure 4. The problematic part of the evaluation board. Note prepared places for the resistors (`RM3` and `RM4`) and mounted capacitors (`C88`, `C87`, `C11`, `C12`)

It was probably the combination of all abovementioned reasons that resulted in our faulty design. Unfortunately, even peer-reviews done by an experienced person during the design process did not prevent the issue.

3.4 Our Advice

If you are an author of a design, make sure that you follow typical conventions and indicate clearly all non-standard solutions. Try to also minimize a possibility of any misunderstanding, e.g. if you cannot entirely remove an obsolete component from your design, at least do not provide its parameters. When using a technical documentation question each and every detail that seems negligible but its meaning is unclear, even if it is just two additional letters. Then, to project a timeline of a design process, plan some additional time for debugging problems which will highly probably occur even if you are following all standard good practices.

4 WiFi Problems

Preparing the wireless communication channel of the control network seemed to be an easier task than adapting Ethernet for our purposes, since no hardware modifications were required and only a proper OpenWrt network configuration had to be created (cf. Section 2). However, experiments carried out to evaluate dependability of those WiFi connections revealed issues, whose puzzling symptoms required significant efforts to identify and eliminate sources of the problems.

4.1 Preliminary tests

Initial tests of WiFi communication were designed to verify if basic WiFi-related functionalities were correctly supported as well as to check how much time for Cherry devices was needed to find an AP, establish connections and send some data. Preliminarily, we took measurements that involved only two prototype Cherry devices (one of them being the AP) and carried out experiments in various smart-home-like conditions and places, starting with our own houses. The tests were run for two sets of devices: first, with vanilla embedded computers on which Cherry devices were based, with their default configuration and random MAC addresses set by their manufacturer; then with Cherry devices running modified version of the operating system (mainly enhanced with our testbed-related applications), customized network configuration and MAC addresses from the pool `00:aa:aa:aa:aX:XX` (`X` digits are the device ID).

4.1.1 The Problem With Connectivity

After deploying the experimental setup we observed that, as expected, the Cherry device connected to Ethernet became an access point and broadcast information on its network. The Non-AP Cherry device, in turn, received the broadcast packets and reacted correctly, that is: it attempted to join the network. With no success. Even though, according to the collected logs, the client device was instantly authenticated every time it tried to establish the connection, 2 or 3 attempts on average were needed to complete the whole operation. We immediately envisioned a setting, in which the connection was not possible at all and the watchdog restarted the device infinitely. Oddly enough, client Cherry devices were perfectly able to join networks governed by 'ordinary APs' (such as home routers) in the first try. After careful investigation, just when we started to question each and every line of the code responsible for WiFi connectivity, it turned out that switching from WPA2 to the older authentication protocol, WPA, solved the problem but...WPA is flawed and insecure [2]. Therefore, it was WPA2 in OpenWrt that was the culprit and needed to be fixed. Unsure how to fix it ourselves, we analyzed history of changes applied to `hostapd` – a piece of software that handled AP's behavior and acted as an authentication server. One look at the newest `hostapd` from OpenWrt official repository was enough to realize that `hostapd` we used (the one included in the latest release of OpenWrt dedicated for the embedded computer by its manufacturer) missed at that time recent and significant modifications. As soon as our OpenWrt-based image for Cherry devices incorporated the newest `hostapd`, the problem vanished completely and the devices became able to connect via WiFi with WPA2 in the first attempt. Unfortunately, the difference between both versions was so immense that we could not analyze it and point the exact problem in `hostapd`.

4.1.2 The Problem With Interference

At the deployment location hundreds of users connected daily to a wireless network available there. It was, therefore, paramount to minimize interference between data transfers in our setup and the high-volume WiFi traffic that already

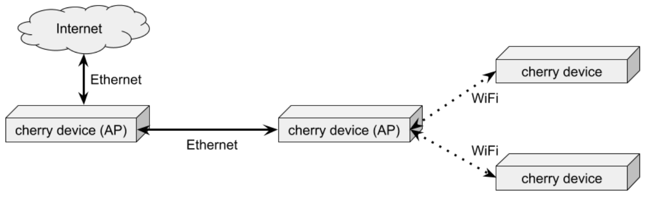

Figure 5. The setup for the integration test

existed in the building. One way of achieving that goal was to use the less congested 5 GHz WiFi, but devices supporting it were significantly more expensive back then, so our embedded computer provided only 2,4 GHz WiFi. The other way to minimize interference was to use the least occupied WiFi channel for the new network.

Most of the modern access points automatically sought the unexploited ranges of frequencies and chose the quietest channel for their network. We expected such behavior from our Cherry devices too since `hostapd` in OpenWrt implemented Automatic Channel Selection (ACS): a survey-based algorithm that relies on information on the current situation on the medium. Specifically, prior to selecting the channel ACS monitors the noise floor and intervals when each of the WiFi channels is busy. This way ACS should minimize interference between networks present in a single location.

When we ran preliminary tests, however, regardless of circumstances our APs always selected the same channel. Their choice did not change even if other channels seemed better and, as we verified manually, were indeed better. It raised our concerns whether ACS worked correctly and soon most pessimistic suppositions got confirmed. The ACS procedure had not even run on the devices since it missed important data on the noise floor, without which the algorithm failed unconditionally. From developers' mailing list [3] we learned that Ralink RT5350 WiSoC, which powered WiFi in the embedded computers, did not support ACS and it would have been extremely difficult to provide ACS with indispensable input. Fortunately, the main wireless network present in the building exploited only certain channels, so for the sake of our particular deployment we decided to set manually a channel that was designated by administrators as unused. Further tests indicated that it was a fully satisfactory solution for us.

4.2 Integration tests

Successful preliminary tests encouraged us to perform an integration test with a more complex setup: four prototype Cherry devices connected as presented in Figure 5. Our goal was to verify if such a miniature testbed was able to revive after possible network failures, e.g. caused by a suddenly disconnected Ethernet cable or cutting off the access point. Before we even got started, a disruptive issue appeared, which was much more troublesome than any of the potential problems being the subject of the trial.

4.2.1 The Mystery of Packet Loss

When all four devices were connected, we experienced difficulties with collecting diagnostic information from them. After taking a closer look we realized that our `ssh`-based tools were barely able to initialize the `ssh` session via WiFi and the wireless data transfer was slow or, momentarily, impossible. Our initial observations were soon confirmed quantitatively: the packet loss was abnormally high, moreover, it soared with the increasing number of devices connected to the AP and occurred for all our devices. With additional experimental setups we compared performance of Cherry devices with an off-the-shelf router and a laptop. Collected results (see Table 1) led to a conclusion that our devices were not handling WiFi connections properly, regardless of whether they played the role of the AP or not.

Table 1. Results of the WiFi connectivity tests

The AP Device	Connected Devices	Packet Loss Resp.
off-the-shelf router	laptop	0%
off-the-shelf router	Cherry device	7%
Cherry device	laptop	38%
Cherry device	Cherry device, laptop	9%, 60%
laptop	Cherry device, Cherry device	5%, 17%

Presented results refer to the chosen WiFi standard, 802.11n. In case of the older standards the devices performed better: for 802.11b there was no packet loss at all; for 802.11g the packet loss depended on the network size (0%, 15%, 35% for, respectively, one, two and three devices connected to the Cherry device AP). Whereas sticking to 802.11b could have been our fallback option, its low speed and features of 802.11n convinced us to put some effort into debugging the problem of flawed reception of data.

Could it be the fault of the particular kind of the embedded computers' chips? Doubtfully, since Ralink RT5350 WiSoC was widely used in home routers and access point devices. Then we checked if the problem had source in our modifications of the operating system: Cherry devices with original OpenWrt behaved identically to those with the modified software. When signal strength, modulation and channel were taken into account the outcome was similar: we run tests with network traffic sniffers, and they did not detect any abnormalities concerning the signal quality. It seemed that that even though the packets were transmitted correctly, some of them were not received.

Confused by the arisen issue, unnoticed during simple preliminary tests, we found information provided by the manufacturer on how to fix WiFi connectivity in their embedded computers: to assure flawless wireless communication one needed to restore certain factory settings of the chip. The instructions had been valid for a newer revision of the embedded computers, therefore we had had to put some effort into adjusting them before they were applied. Afterwards the packet loss for 802.11g equaled 0%, 1% and 9% respectively for one, two and three Cherry devices connected to the Cherry device AP. To our dissatisfaction, the situation did not improve for 802.11n.

Determined to fix the problem we bought a bunch of embedded computers of the most recent revision, which was said to contain WiFi-related modifications. The new devices, similarly to their predecessors, got configured with our version of OpenWrt as well as our factory image, used consecutive MAC addresses in accordance with their IDs and run a simplistic program to send data. They did not show any improvement with regard to our particular issue. Similarly disappointing results were brought about by installation of the up-to-date Linux kernel.

Another examination of the sniffed packets revealed that, after all the actions we had taken, the situation had not changed a bit: a lot of packets were retransmitted as though they were ignored by the receiver. It seemed that the source of the problem lied in the firmware or the radio driver and since the latter turned out to be a simple and seemingly harmless program, suspicion was cast upon the former one. We checked the factory partition: maybe our image had overwritten the radio firmware? Or maybe it had configured radio incorrectly? To verify those possibilities we took new vanilla embedded computers (with the factory image) and run the experiment without applying any modifications. WiFi connection (802.11n) worked. Taken aback, we installed on the very same devices a factory image taken from Cherry devices. WiFi connection (802.11n) worked perfectly.

4.2.2 *The Problem Demystification*

There was one overlooked difference between the devices with working WiFi and those that were not able to transfer data via 802.11n. It was their MAC addresses. The embedded computers with original, random MAC addresses worked well while the devices with our MACs had their radio malfunctioning. That valuable observation let us to finally understand the problem. In the Ralink's documentation [10] of more than 200 pages, on page 179, in the third line of the fourth table we found the missing puzzle pieces:

> *"In multiple-BSSID AP mode, BSSID is the same as MAC_ADDR, that is, this device owns multiple MAC_ADDR in this mode."*

Any embedded computer being the AP was, by default, functioning in 8-BSSID mode. This means it was in fact owning 8 consecutive MAC addresses (including the address manually set by us – the only address we were aware of since we put just one wireless network in Cherry device's configuration). Some of those addresses were assigned by us to other devices (we used consecutive numbers for IDs and, thereby, consecutive MAC addresses for other devices), what resulted in double MAC addresses in the network – the direct reason for the packet loss. To fix the problem it was enough to ensure that no two MAC addresses that we set were within the range of 8 from each other. From that point onward we were using MAC addresses ended with 0 (separated by 15 MAC addresses between) so that they were not doubled by any additional MAC addresses used in 8-BSSID mode.

4.3 The Wisdom of Hindsight

If you rely on third-party software do keep it up to date with changes published by the mainstream. It is also advised to verify the quality of implementation of important functionalities (like the discussed channel selection) and, in general, not to rely on a belief that some features are standard and omnipresent. One should be careful especially in case of a large-scale project like ours. Since economizing on the cost of a single device results in thousands of dollars in savings, the purchase of cheaper products is much more tempting or even is the only option possible. Such low-end products tend not to support functionalities considered to be basic in recommended pricey counterparts that one may be used to when working with single devices or small networks.

To avoid further surprises, study thoroughly description of all modes (literally *all* of them: default ones and those set manually) and applied settings in documentation: had we read everything about the AP mode in the Ralink chip's datasheet, we would have known about BSSID from the very beginning. Also, if you debug a problem that involves third-party accessories, try to recreate the issue using the original products with factory configurations. Only then apply your modifications for further experiments.

Finally, tests run on a few prototypes (like our preliminary tests) do not prove the whole network of the devices would work well. You may be unlucky to test working subsets of the prototypes (e.g. with compatible MAC addresses) without noticing the problem that would appear in other settings.

5 Conclusion

We hope that these examples already illustrate that designing a new device that is expected to operate reliably at scale is not a trivial task. Even having highly experienced people on the team, employing standard popular components, minimizing risky features, and following modern quality assurance practices may sometimes not be enough. We learned that in addition a close attention has to be paid to very tiny details in documentation, that successful tests with individual devices do not guarantee successful functioning of the entire system, that even the tiniest sign of atypical yet totally plausible behavior has to be investigated, and much more. The assumptions behind our undertaking definitely made our task harder. Solely at the device design stage we experienced a dozen other serious technical issues, from reliable and precisely controllable power management of low-power SoCs to not sufficiently reliable persistent storage, which could likely contribute to a few more papers, not to mention problems of a different nature or at different stages, notably during the deployment. At the same time, the goals and constraints of the project motivated us to act more carefully. As a consequence, the fatal technical problems were detected early enough to allow us to fix them.

Acknowledgments

This work was conducted within the HENI project, supported by the National Center for Research and Development (NCBR) in Poland under grant no. LIDER/434/L-6/14/NCBR/2015 within the LIDER-VI program.

6 References

[1] OpenWrt. `https://openwrt.org/about`.
[2] OpenWrt: Configure WiFi encryption. `https://openwrt.org/docs/guide-user/network/wifi/encryption`.
[3] rt2800usb and automatic channel selection. `http://rt2x00.serialmonkey.com/pipermail/users_rt2x00.serialmonkey.com/2014-December/013535.html`.
[4] Digi International Inc. *PCB Layout for the Ethernet PHY Interface*, 2010. TN266.
[5] Freescle Semiconductor Inc. *i.MX28 Layout and Design Guidelines*, 2009. AN4215.
[6] IEEE. *IEEE Standard for Ethernet*, 2018. IEEE Std 802.3-2018.
[7] K. Iwanicki. A distributed systems perspective on industrial IoT. In *Proc. 38th IEEE Int'l Conf. Distributed Comput. Syst. (ICDCS)*, 2018.
[8] Microchip. *ENC28J60 Stand-Alone Ethernet Controller with SPI Interface, Data Sheet*, 2004. DS39662A.
[9] Pulse Electronics. *Layout Considerations for Pulse Ethernet Magnetics and Ethernet Connector Modules*, 2019.
[10] Ralink. *RT5350 Preliminary Datasheet*, 2010. Rev. 1.
[11] Texas Instruments. *AN-1469 PHYTER Design & Layout Guide*, 2013. SNLA079D.