R. Reussner, A. Koziolek, R. Heinrich (Hrsg.): INFORMATIK 2020,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2021 1185

Evaluation of Interoperability Between Various Implementations of the Thread Protocol Stack

Sebastian Miethe[1], Silvia Krug [2]

Abstract: The increasing popularity of Internet of Things (IoT) applications leads to a continuous expansion into further application areas. Since each application poses its own challenges and requirements, many different solutions have been developed to build parts of the IoT. However, this diversity results in new challenges as well. Especially if a cooperation between heterogeneous components or a later addition of components is required, interoperability becomes challenging. Precision agriculture is one example application area that requires both conditions. In this paper, we assess the interoperability of three different implementations of the Thread protocol stack. To this end, we propose an empirical analysis method that can similarly be applied to any other interoperability evaluation. Our results show that the versions under test exhibit various interoperability problems and only certified devices work without issues.

Keywords: Internet of Things; Protocol Interoperability; Thread; OpenThread

1 Introduction

Smart or precision agriculture opens up new possibilities for farmers to manage their assets efficiently. Therefore, farmers have become early adopters for digital technologies such as the Internet of Things (IoT) [KJL19]. However, the fast development of technologies results in many design options on the market that farmers cannot overview. Besides that, limited budgets can lead to successive installations. This leads to the requirement of future-proofed and extendable technologies, which is made possible through interoperability between heterogeneous system components. If two systems can exchange data and work together according to a protocol specification, they are considered interoperable [Zh08]. To what extend this can be guaranteed is however an open question.

There are several reasons for limited interoperability, ranging from constrained hardware resources through faulty or incomplete protocol implementations and even security reasons [KB17]. Interoperability analyses as such are nothing new. In [Ay18] for example, the authors show that no implementations of 6LoWPAN covers all features. The reasons for this are the complexity of the protocol and hardware restrictions. Missing implemented features caused package drops and so limits interoperability. In [Ko11], the authors analyze Routing Protocol for Low Power and Lossy Networks (RPL), an IoT routing protocol.

[1] IMMS Institut für Mikroelektronik- und Mechatronik-Systeme gemeinnützige GmbH (IMMS GmbH), Ehrenbergstraße 27, 98693 Ilmenau, Germany — sebastian.miethe@imms.de
[2] IMMS GmbH — silvia.krug@imms.de, https://orcid.org/0000-0003-0282-5471

doi:10.18420/inf2020_110

They show that certain versions can work together, but performance metrics might drop. Another analysis, targeting Constrained Application Protocol (CoAP), has shown that the choice of programming language may lead to faster responses to client requests while some implementations were not able to work with others at all due to different protocol versions [IOU17]. Reasons for limited interoperability and its effects are divers for different protocols and entire IoT-systems. Limited interoperability results in problems when *joining a network* or during *network operation* in terms of stability and transfer-related *performance parameters*. So further investigations on interoperability in IoT are needed.

In this paper, our focus was to experimentally evaluate the interoperability of three implementations of the Thread protocol stack and their impact on real deployments. We developed an adaptive model allowing various experiments to verify the extend of cooperation between the chosen implementations. Other more formal approaches to verify the interoperability were therefore out of the scope of this work. In addition, we evaluate typical performance metrics to observe variations caused by lacking interoperability. As explained later, Thread addresses interoperability directly however issues have to be expected nevertheless.

2 Fundamentals

2.1 The Thread Protocol stack

Thread is a protocol stack for wireless mesh networks first released in 2015. The main goals of Thread are in low power consumption, secure networking, being economical in purchase and operation, simple network installation and operation, and scalability. It is based on well-known standards and covers the physical through transport layers (see Figure 1). The application layer is intentionally unspecified to allow various IP-based applications. [Th17b]

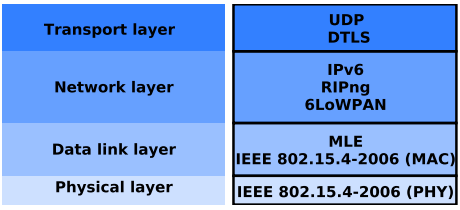

Fig. 1: Thread protocol stack [Th15b]

2.2 Joining a Thread Network

Since we aim to test the ability to join an network, it is necessary to understand how this process works. A so-called *joiner* needs to complete the three steps *discovery*, *commissioning*, and *attaching* in order for it to successfully join a network [Th15b].

In the *discovery process*, a joiner actively searches for an existing network via Mesh Link Establishment (MLE). It sends out a *Discovery Request* and waits for a *Discovery Response*. Afterwards, the *commissioning process* according to the Mesh Commissioning Protocol (MeshCoP) starts [Th15a]. Every network needs exactly one router assuming the role of *commissioner*. The joiner needs to authenticate at the commissioner with a key and its Medium Access Control (MAC) address. Then, the network credentials can be passed to the joiner via Datagram Transport Layer Security (DTLS). After the commissioning step, the joiner needs to find a parent to which it can *attach* itself. The joiner requests this with a multicast message for potential parent [Th15a]. After *attaching*, it becomes a full member of the network. If the number of routing devices in the network is below a certain threshold, the joiner request the leader to become a routing device, via CoAP [Th15b].

2.3 Thread Product Certification

Several companies develop own implementations of Thread. *The Thread Group, Inc.* offers a certification program which is voluntary and shall ensure interoperability between Thread devices [Th17a]. If a SoC and protocol implementation passes all tests, it can use the label *Thread Certified Component*. This is relevant because we test certified and uncertified products allowing us to draw conclusions on the significance of the certification.

3 Measurement Setup and Implementation

The interoperability of the three implementations Openthread, Kinetis Thread Stack and Mbed Thread Stack will be evaluated. OpenThread and the Kinetis Stack implement version 1.1.1, while Mbed implements version 1.1.0 of the Thread-specification. These versions should work together well, because only errata were corrected in version 1.1.1.

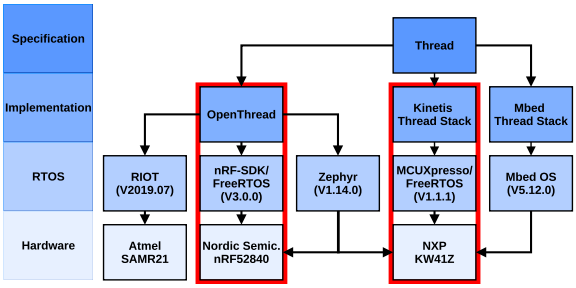

Fig. 2: Utilized soft- and hardware

Figure 2 gives an overview of the implementations, cross-referenced with the respective Real-Time Operating Systems (RTOSs) and its version as well as the used hardware platform. The two combinations outlined in red are officially *Thread Certified Components*. Using

these implementations, we will perform experiments in selected scenarios: *joining a network* and the *network operation with* and *without application data traffic*.

3.1 Modeling

We developed a general model as common approach for testing the interaction between implementations. It is applicable to other protocols. Figure 3 shows the model for testing the implementations in the process of joining a network. The SUT (System Under Test) contains two or more IUTs (Implementation Under Test) and observes their interactions. To control the SUT and its IUTs, inputs are needed. Most of the used RTOSs offer a Command Line Interface (CLI) which allows the user to manage things like joining a network manually via commands. We use the CLIs as an input. The CLI can also be used as output because it allows the user to request information about the network status and settings. E.g. a *neighbor table* can be requested to check if nodes unexpectedly lose connection to the network. Since Thread uses *IEEE802.15.4*, an *IEEE802.15.4*-compliant sniffer with *Wireshark* is used to capture the network traffic adn compare it according to the analyses in Section 2.2.

3.2 Joining a Network

To verify if every implementation is able to join a network of nodes with a different implementation, we set up a node (IUT A) which advertises a network. Network settings like channel, network name, and Pre Shared Key for the Device (PSKd) have to be adjusted in the source code or via the CLI. Then, the alternative implementations (IUT B) join the network one by one. We captured the packet exchange during this process in order to compare it to the expected exchange according to the specification. Figure 3 shows an incomplete finite state machine for the joining process used as reference for evaluation.

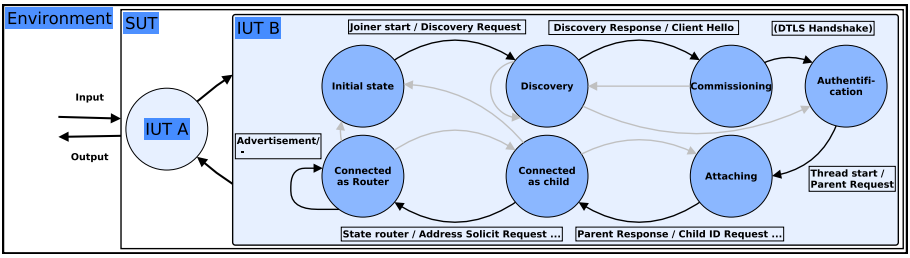

Fig. 3: Packet exchange for joining a network

3.3 Network Operation Without Application Data Traffic

When a node has joined a network, no disconnections among any other nodes should happen. After topology changes the network is kept alive via so-called *advertisements*.

Those are periodically sent by every node, as determined by the *Trickle Algorithm* [Th17b]. To examine mutual influences between pairs of implementations, we set up individual networks consisting of two nodes and observed them for one hour each. We recorded the packet exchange and neighbor table of each setup, to check that nodes remain continuously connected. In addition, we set up a network with all combinations of implementations and hardware platforms and observed them for five days to evaluate any long-term effects.

3.4 Network Operation With Application Data Traffic

Finally, we verify if parameters such as latency, UDP- or MAC-packet error rate (PER) are influenced by the implementation. We use UDP to generate user data traffic since it is used at the highest layer of the Thread stack (see Figure 1).

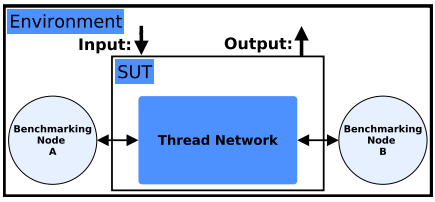

Fig. 4: Model for measuring latency and PER

Figure 4 shows two nodes with an application for benchmarking UDP-traffic in a Thread-network. This network consists of two nodes that were varied in their implementation. After *Benchmarking Node A* has sent a UDP-packet it waits for a UDP-acknowledgment from *Benchmarking Node B*. The *latency* [*ms*] is measured as the Round-Trip-Time (RTT) from sending the UDP-packet until obtaining the UDP-acknowledgment. Furthermore, the *UDP-PER* [%] is recorded as the percentage of retransmissions needed due to missing UDP-acknowledgments. Similarly, the *MAC PER* [%] is determined.

We performed the measurement in two different configurations. In configuration I 1.000 UDP packets were sent with a payload of 20 Byte each and an acknowledgment timeout of 120 ms. In configuration II the payload and timeout are increased to 100 Byte and 216 ms respectively. The increase in size forces the 6LoWPAN-layer to perform fragmentation and show a variation in delay. We chose the *ack timeout* based on empirical tests. It is quite generous, because we are interested in packet loss due to interoperability issues and not due to high latency. The transmission rate is not fixed, the sender sends the next packet after getting the acknowledge or after the time exceeds. To avoid package loss due to low signal strength caused by obstacles and path loss, we kept the nodes in close proximity and a maximum distance of one meter. We set up the wanted network topology manually with the help of MAC filtering via the CLIs configuring MAC-blacklists. In addition, we ensured that all nodes rated incoming packets with *Link Quality* = *3*, which corresponds to an RSSI of more than 20 dB, which would be rather high in a real world scenario. The network

was varied by permuting the three tested implementations as table 1 shows. We chose the *nRF-SDK* on the *nRF52840* as Openthread variant since it worked without errors in previous tests. For Kinetis, the *MCUXpresso-SDK*, and for the Mbed Thread Stack *MbedOS* on the *NXP KW41Z* was used (see Figure 1). We tested every permutation five times with the two configurations and calculated the average values from these measurements.

	homogeneous			heterogeneous		
IUT A	Kinetis	Mbed	Openthread	Kinetis	Mbed	Openthread
IUT B	Kinetis	Mbed	Openthread	Mbed	Openthread	Kinetis

Tab. 1: Combinations of IUTs whose latencies and PER were to be measured

4 Results and Discussion

4.1 Joining a Network

Now, we examined the experiments. Beginning with the network joining process, all combinations where observed three times to validate that occurring errors are reproducible. Table 2 summarizes the results. It shows all tested combinations of a given *commissioner* and a *joiner* node.

			Joiner					
			nRF52840		KW41Z			SAMR21
	Hardware	Software	nRF-SDK	Zephyr	Zephyr	Mbed	Kinetis	RIOT
Commissioner	nRF-52840	nRF-SDK						1*
		Zephyr					2*	
	KW41Z	Zephyr						
		Mbed						3*
		Kinetis						4*
	Atmel	RIOT						

Tab. 2: Overview of the results for the network joining process

The Mbed Thread Stack does not implement the joiner and commissioner. Since we focused on testing the initial state of each implementation, we were not able to test the Mbed Thread Stack at this point. Combinations marked with green exhibited the expected packet exchange as described in section 2. No issues occurred so interoperability appeared to be given. Combinations marked with yellow were also able to join, but some deviations from the expected packet exchange occurred. This shows that all combinations with *Zephyr OS* on the *KW41Z* System-on-a-Chip (SoC) were affected. We observed that duplicates of packets appeared frequently during the DTLS-handshake. At a closer look, the duplications happened because *Zephyr* exceeds the *THR_DTLS_INIT_RETRANSMIT_TIMEOUT* = 8 s during the DTLS-handshake. This leads to retransmissions as stated in the specification

[Th17b]. In some combinations, it also happened that retransmitted packets were interpreted by the receiver as another DTLS handshake request. This caused a temporary processing of two DTLS handshakes in parallel between the same two nodes. Thus, in a worst case scenario, 46 % more packets had to be sent for the joining process, which is especially detrimental for energy-constrained devices. Furthermore, a long runtime on batteries is mandatory for IOT-systems to be practicable for farmers. All combinations with *RIOT* are highlighted with red. When *RIOT* is the joiner, the process triggers a *RIOT kernel panic* on the CLI and it stops working. As commissioner, it does not reply to the joiner during the DTLS handshake. This behaviour prevents any joining and could happen due to package drops caused by missing features in the implementation.

4.2 Network Operation Without User Data Traffic

4.2.1 Individual Observation

In the next step, we observed all implementations separately in networks of two nodes. The network credentials where predefined in source code to skip the commissioning. Nearly all combinations behaved as expected, except for the ones marked with 1* through 4* in Table 2. In combinations 1* and 2* *RIOT* respectively *Zephyr*, stopped sending the periodical *advertisements* at some point. These are necessary to keep connections up between the participants of the network and thus keeps the network as a whole alive. As a result, the remaining node discarded the routing entry for the other node. This way, *RIOT* had not been part of the network for about 4:50 min and *Zephyr OS* even for 8:30 min.

In combinations 3* and 4*, *RIOT* did not reply to the requests from the Mbed Thread Stack and Kinetis Thread Stack to become routing devices. Therefore, they remained as a child. It was interesting for us that *RIOT* did reply to all other combinations, which are based on Openthread, namely *Zephyr*, *nRF-SDK* and *RIOT* itself. Conversely, *Zephyr* and *nRF-SDK* replied to *Mbed* and Kinetis Thread Stack. This indicates that there is an issue with *RIOT* in combination with Openthread, but not with Openthread itself. The reason might be that *RIOT* drops the CoAP request due to implementation incompatibilities.

4.2.2 Collective Observation

After observing all implementations separately, the operation of all nodes at once was monitored for five days. We did this three times to verify that observed effects where reproducible. The duration in Table 3 show how long the nodes participated in the network during the total duration of 120 h.

nRF52840		KW41Z			SAMR21
nRF-SDK	Zephyr	Zephyr	Mbed	Kinetis	RIOT
120 h	36.4 h	70.7 h	120 h	120 h	120 h

Tab. 3: Overview of the results for the network operation for five days

The *RIOT* node was part of the network all the time but disrupted the rest of the network. That is why we marked it with yellow. *RIOT* exceeded the *MAX_NEIGHBOR_AGE* = 100 s period for sending *advertisements* [Th17b]. We found that this caused the other nodes to deleted its routing entry and created it again when it started sending again. This happened periodically about every 72 min and is problematic in two ways. At first, nodes consider their *RIOT* peer gone and thus are not able to send packets to it. After the *RIOT* node starts sending again, all nodes want to disseminate the change in the network. This results in a significant increase in the number of messages exchanged and thus again in a higher energy consumption. The combinations of cells marked with red completely disconnected from the network. As we noticed this happened because they stopped sending *advertisements* after a certain time. As the microcontroller did not respond any more via the CLI, *Zephyr OS* probably crashed.

4.3 Network Operation With User Data Traffic

To evaluate differences in performance between the selected implementations, we performed the experiment as detailed in Section 3.4 with one participant of each of the three implementations. Figure 5 shows the measured values for the homogeneous networks. The latency shows that Openthread is the fastest implementation, which could be due to differences in the used hardware platforms. Interesting is the UDP-PER with the Mbed Thread Stack, which is extremely high compared to the others. We found out that this was caused by missing acknowledgments on the MAC layer that were also responsible for the higher MAC-PER. Those missing acknowledgments by the Mbed Thread Stack also caused connection losses and reattachments with the other participants. Those disappearing paths to neighboring nodes reduces the redundancy in a network and therefore reduce its resilience.

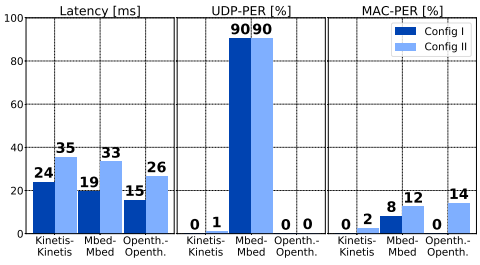

Fig. 5: Homogeneous networks

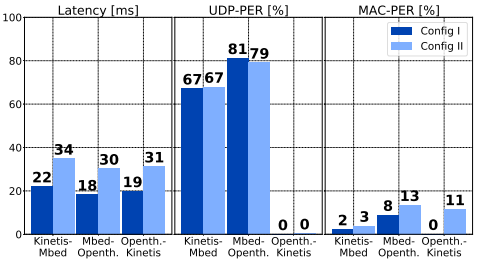

Fig. 6: Heterogeneous networks

The results with the heterogeneous networks are rather predictable. The combinations with a Openthread participant e.g. have the lowest latency because Openthread already was the fastest in the homogeneous networks. The same is true for the UDP-PER. We found that combinations with the Mbed Thread Stack show bad results since Mbed Thread Stack was again responsible for missing acknowledgments. Interestingly, the Kinetis Thread Stack-Mbed Thread Stack combination showed around 14 % less PER than Mbed Thread Stack-Openthread. Still, a PER of 67 % is not good compared to the 0 % at Openthread-Kinetis Thread Stack.

The overall results show that interoperability between various Thread stacks cannot be taken for granted. The fact that errors occurred with OpenThread used in combination with *RIOT* and *Zephyr* but not with the *nRF-SDK*, even thought they all use the same version of Thread, shows that not just the implementation but the used RTOS can be the cause of problems. Only the Kinetis Thread Stack and OpenThread on the *nRF-SDK* exhibited no issues, full interoperability and stable network operation. All other combinations caused problems that restricted the interoperation with other nodes and made operation of the network unreliable and therefore less resilient.

The two certified stacks thus are to be preferred for application development because this lowers the risk of having to deal with limitations and issues during development and later operation. For the considered domain of agricultural applications in particular, robust network operation and future-proofing interoperability are prime concerns. Especially, topology changes are crucial for agriculture applications. There, it is likely that nodes get disconnected e.g. to save power or due to changed vegetation properties altering the signal strength at the receiver. Dealing with such harsh conditions, requires a stable and robust network operation as base in order to cope with different dynamic effects. This aspect will be covered in future studies.

5 Conclusion and Future Work

In this paper, we evaluated the interoperability of the Openthread, Kinetis- and Mbed thread stack, targeting the application in agriculture scenarios. We achieved this evaluation by modeling typical scenarios and performing empirical tests. The investigations confirmed that interoperability cannot be taken for granted and should be evaluated. The results confirm that officially Thread-certified products work together flawlessly with other certified stacks. This makes networks more stable as no unexpected behaviour could be observed. All other implementations experienced issues.

Continuing this work, further Thread-certified components should be analyzed to confirm that they effectively prevent interoberability issues. In addition, more scenarios besides the network joining process and operation should be considered. One crucial aspect we will focus on is the behavior in case of topology changes due to harsh environmental conditions as typically experienced in outdoor agriculture applications.

Acknowledgements

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

The EXPRESS project is supported by funds of the Federal Ministry of Food and Agriculture (BMEL) based on a decision of the Parliament of the Federal Republic of Germany. The Federal Office for Agriculture and Food (BLE) provides coordinating support for digitisation in agriculture as funding organisation, grant number FKZ 28DE102C18.

References

[Ay18] Ayers, H.; Thomas Crews, P.; Hua Kian Teo, H.; McAvity, C.; Levy, A.; Levis, P.: Design Considerations for Low Power Internet Protocols. 2018.

[IOU17] Iglesias-Urkia, M.; Orive, A.; Urbieta, A.: Analysis of CoAP Implementations for Industrial Internet of Things: A Survey. Procedia Computer Science 109/, pp. 188–195, 2017.

[KB17] Konduru, V. R.; Bharamagoudra, M. R.: Challenges and solutions of interoperability on IoT: How far have we come in resolving the IoT interoperability issues. In: 2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon). Pp. 572–576, 2017.

[KJL19] Klerkx, L.; Jakku, E.; Labarthe, P.: A review of social science on digital agriculture, smart farming and agriculture 4.0: New contributions and a future research agenda. NJAS - Wageningen Journal of Life Sciences 90-91/, 2019.

[Ko11] Ko, J.; Eriksson, J.; Tsiftes, N.; Dawson-Haggerty, S.; Terzis, A.; Dunkels, A.; Culler, D.: ContikiRPL and TinyRPL: Happy Together, 2011.

[Th15a] The Thread Group, Inc.: Thread Commissioning, 2015, URL: www.threadgroup.org/Portals/0/documents/support/CommissioningWhitePaper_658_2.pdf, visited on: 07/10/2019.

[Th15b] The Thread Group, Inc.: Thread Stack Fundamentals, 2015, URL: www.threadgroup.org/Portals/0/documents/support/ThreadOverview_633_2.pdf, visited on: 06/07/2019.

[Th17a] The Thread Group, Inc.: Thread Certification Program Opens: Top Reasons To Get On Board Now!, 2017, URL: www.threadgroup.org/news-events/blog/ID/146/Thread-Certification-Program-Opens-Top-Reasons-to-Get-on-Board-Now#.XRCFY69R09l, visited on: 06/16/2019.

[Th17b] The Thread Group, Inc.: Thread Specification, version 1.1.1, 2017.

[Zh08] Zhong, N.; He, Z.; Kuang, J.; Zhuo, Z.: Optimal Protocol Interoperability Test Generation via Heuristic Algorithm. In: 2008 International Conference on Internet Computing in Science and Engineering. Pp. 278–281, 2008.