Development of Thread-compatible Open Source Stack

Lukas Zimmermann, Nidhal Mars, Manuel Schappacher and Axel Sikora

Institute of Reliable Embedded Systems and Communication Electronics, University of Applied Sciences Offenburg, D77652 Offenburg, Germany

lukas.zimmermann@hs-offenburg.de

Abstract. The Thread protocol is a recent development based on 6LoWPAN (IPv6 over IEEE 802.15.4), but with extensions regarding a more media independent approach, which – additionally – also promises true interoperability. To evaluate and analyse the operation of a Thread network a given open source 6LoWPAN stack for embedded devices (emb::6) has been extended in order to comply with the Thread specification. The implementation covers Mesh Link Establishment (MLE) and network layer functionality as well as 6LoWPAN mesh under routing mechanism based on MAC short addresses. The development has been verified on a virtualization platform and allows dynamical establishment of network topologies based on Thread's partitioning algorithm.

1. Introduction

Nowadays, several so-called megatrends like Internet of Things (IoT) are rapidly emerging and are leading to an increasing amount of new protocols and solutions to cover the manifold requirements of different applications. Wireless technologies based on embedded systems play a major role for the local communication. Since the use of IP-based addressing protocols became a necessity to give IoT devices the access to cloud services and interact together, the major drawback was the limited number of unique addresses provided by the IPv4 protocol. This issue was addressed by the development of IPv6 which allows a nearly unlimited number of devices and therefore provides the basis for the IoT. However, when using the IPv6 protocol for 'things' which are mainly embedded devices, several restrictions regarding the underlying technologies and performance of the devices must be considered. Therefore, the usage of IPv6 for embedded devices requires an adaptation layer named 6LoWPAN (IPv6 Low Power Wireless Personal Area Networks) [1] in order to be used in low power wireless networks.

Nevertheless, even with a common network layer there are still several points which are not defined by 6LoWPAN such as the routing protocol and its parameters what might lead to incompatibilities. Therefore, the Thread protocol defines all the layers and technologies required to create a common communication network. To evaluate such a Thread network and to analyse its operation, we extended an open source 6LoWPAN stack – namely emb::6[a] - in order to comply with the Thread specification for connected home applications. The paper is structured as follows: Section 2 provides an overview of the emb::6 stack and the Thread protocol explaining the basic ideas behind. Sections 3 and 4 give insight into selected Mesh Link Establishment (MLE) and network layer functionality. In section 5 we present the memory usage of our implementation. The last section concludes our work. Thread related content is based on [2].

[a] https://github.com/hso-esk/emb6

2. Related Work

The Thread implementation developed and presented herein is using the 6LoWPAN communication stack emb::6. It mainly covers the layer 2 and layer 3 requirements from the Thread Specification [2].

2.1. emb::6 networking stack

emb::6 [3][4] is an open source wireless communication stack based on the 6LoWPAN communication protocol [8]. The initial development of the emb::6 network stack has started as a fork of Contiki OS[b] with several changes regarding the basic architecture. In general, emb::6 covers the Application Layer, Transport Layer, Network Layer, Data Link Layer, Physical Layer, and the Radio Driver. The basic architecture of the emb::6 wireless network stack consists of its networking core, a Hardware Abstraction Layer (HAL), Radio Drivers (RF) and a separate so-called utility module that implements all common functionalities such as timer and event handling used by all other layers and modules. The networking core handles the network related tasks, mainly the communication part, where different tasks are distributed over several layers. Beginning on top at the Application Layer (APL), usually serving as interface for the device application, the stack forwards requests layer by layer down to the radio driver, which is responsible for the implementation of the RF module drivers.

2.2. Thread protocol

The Thread protocol is an open standard for reliable, cost-effective, low power, wireless device-to-device communication. It is designed specifically for connected home applications where IP-based networking is desired and a variety of application layers can be used on the stack. The Thread standard is based on IEEE 802.15.4 (2006) MAC and physical layer operating at 250 kb/s in the 2.4 GHz band. Figure 1 illustrates a general overview of our Thread stack implementation architecture [6]. This work mainly concentrates on MLE and network layer as described in chapter 4 and 5 in the Thread specification.

2.3. Thread device types

The Thread network uses different types of devices as illustrated in figure 2.

- Border Router: A specific type of router that supports multiple interfaces besides IEEE 802.15.4 in order to connect with other networks, e.g. Wi-Fi, Ethernet, etc.
- Router: Used to provide routing services to the network and handle joining and security services for devices trying to join the network. Routers are not allowed to operate as sleepy end devices, but may downgrade their functionality and become REEDs (Router-eligible End Devices).
- Leader: The device that makes decisions within the Thread network and manages router ID assignments. The Leader is the first active router on the network and can be elected in case of losing connectivity.
- Router-eligible End Devices: REEDs have the capability to become routers without user interaction, if necessary.
- End Devices: End devices communicate only through their parent router and cannot forward messages to other devices. To save energy they can sleep for a time period and poll their associated router for data once they are awake.

[b] https://github.com/contiki-os/contiki

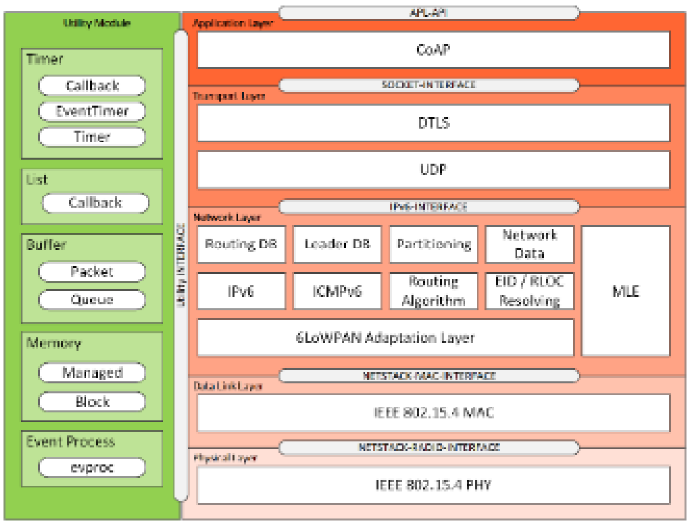


Figure 1. Thread network stack.

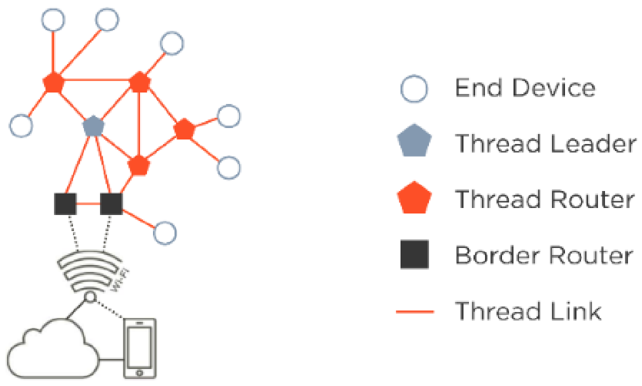


Figure 2. Thread device types [7].

3. Mesh Link Establishment

The existence of many asymmetric radio links within the IEEE 802.15.4 network represents one of the main issues while establishing links between nodes. Thread is using the so-called Mesh Link Establishment protocol (MLE) [8] to resolve such kind of problems in addition to other capabilities.

In this section, we give an overview about the capabilities of the MLE layer and its architecture. Furthermore, we will highlight the main processes of MLE that have been implemented.

3.1. MLE capabilities and architecture

MLE is a protocol that is used to configure and secure radio links dynamically as the topology and physical environment change. This is done by exchanging IEEE 802.15.4 radio parameters between nodes such as addresses, node capabilities and frame counters.

MLE allows all nodes to synchronize periodically and share radio link parameters to adapt to any change that might happen on the topology such as joining of new devices. Furthermore, MLE can detect unreliable links before any effort is spent for configuring them. For example, a link between two devices that is strong in one direction may be unusable due to weak signal strength towards the other direction. MLE resolves this by allowing a node to send periodical link-local multicast messages containing an estimated link quality for all links. In addition, MLE exchanges link costs between nodes by sending MLE advertisement messages.

MLE advertisement messages are used to exchange bidirectional link quality between neighboring routers. All routers exchange periodically single-hop MLE advertisement packets containing link cost information. Those periodic advertisements allow routers to quickly detect changes in the set of neighboring routers. For instance, if a new router joins the network, an existing router has been downgraded to REED or if a router lost connection to the Thread network.

Regarding the architecture, MLE cannot be placed in OSI Model clearly. Instead, it operates alongside the stack using UDP (User Datagram Protocol) as transport protocol. This architecture is also given for other systems that make use of MLE, such as ARM mbed OS [9]. Figure 3 shows the different protocol modules used by ARM mbed OS and the interaction of the MLE protocol with the existing layers.

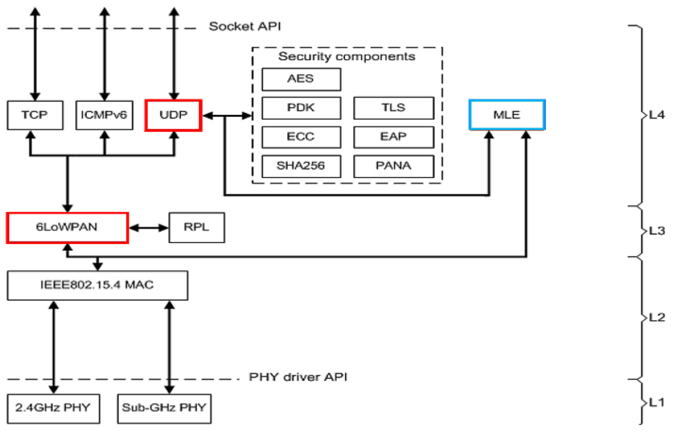


Figure 3. ARM 6LoWPAN stack alongside OSI model [9].

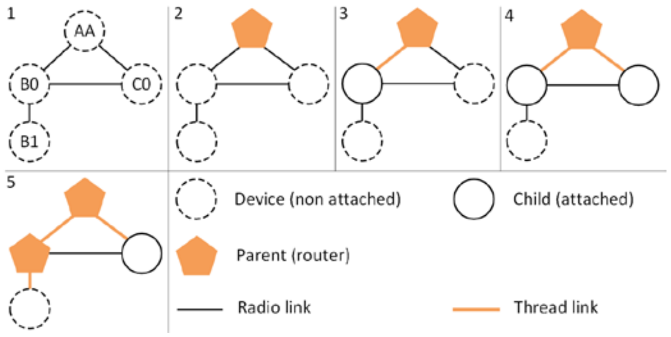


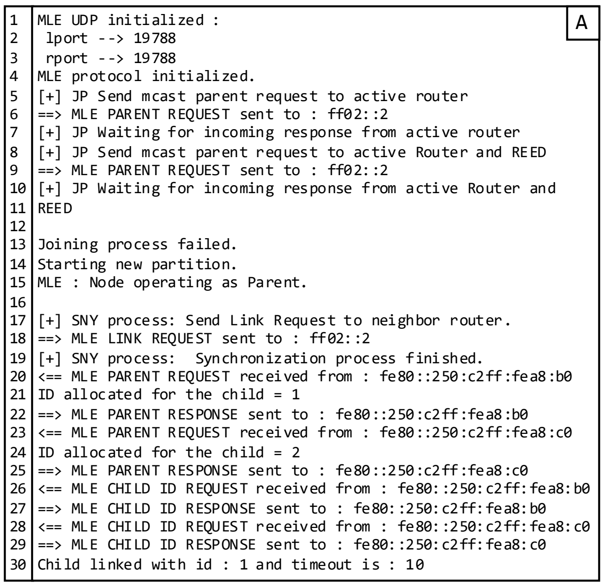
Figure 4. Testing scenario for MLE joining process.

3.2. MLE processes and test cases
In Thread networks, all devices join to the network either as an end device or as a REED. Joiner devices always try to attach to an active Thread router from which they allocate a 16-bit short address. In case such a join process fails, a second request is sent to both routers and REEDs.

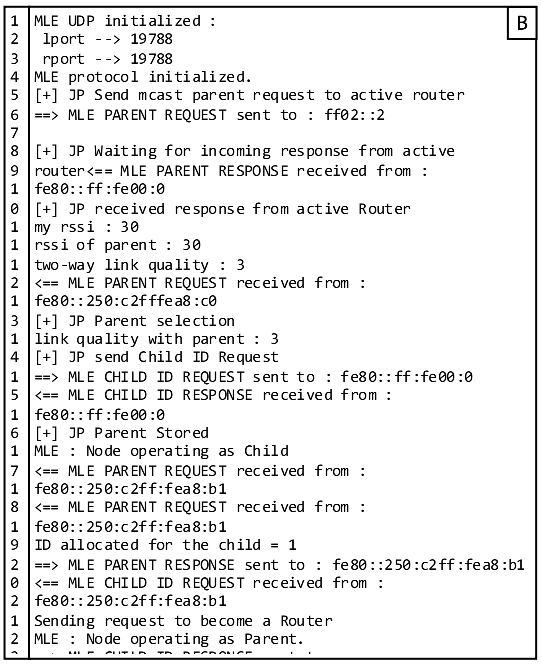
Figure 4 briefly shows such a mesh link establishment scenario. We use four nodes that have 0xAA, 0xB0, 0xC0 and 0xB1 as last two bytes of their MAC address, respectively. Possible radio links are defined statically by the environment and are delineated by the gray line.

Figure 5 shows the trace output of the nodes for this scenario. Window A corresponds to node 1 (0xAA) and window B to node 2 (0xB0).

- Node 1 (0xAA) is the first active node in the network. The joining process fails since no other routers are available (line A.12). Consequently, the node creates a new partition and starts operating as a parent (lines A.13 and A.14).
- Node 2 (0xB0) attaches to node 1 after exchanging four handshake messages. Node 2 operates as a child after receiving the CHILD ID RESPONSE (lines B.18 and B.20).
- Node 3 (0xC0) sends a multicast PARENT REQUEST. Node 1 and node 2 receive the message (line A.22 and B.13), but only node 1 replies due to a scan mask TLV (the first request should be replied only by the active router). In case that more than one parent responds, the joining device compares them and selects the best device to be its parent using the connectivity TLV received in the parent response and the calculated two-way link quality (calculated using the received link margin TLV in the parent response and the RSSI of the response itself). Then the handshake process will continue normally between node 3 and node 1. Finally, node 3 (0xC0) will start operating as a child.
- Node 4 (0xB1) has one possible radio link with node 2. Although, node 2 (0xB0) only replied for the second PARENT REQUEST (line B.24). This is explained by the fact that on the first request only the active router should reply (node 2 is operating as a child at that moment). Once node 2 receives a CHILD ID REQUEST, it sends a request to the leader to become a router (line B.26). Finally, node 2 switches its mode to active router (line B.27) and node 4 starts operating as a child.



```
A
1  MLE UDP initialized :
2   lport --> 19788
3   rport --> 19788
4  MLE protocol initialized.
5  [+] JP Send mcast parent request to active router
6  ==> MLE PARENT REQUEST sent to : ff02::2
7  [+] JP Waiting for incoming response from active router
8  [+] JP Send mcast parent request to active Router and REED
9  ==> MLE PARENT REQUEST sent to : ff02::2
10 [+] JP Waiting for incoming response from active Router and
11 REED
12
13 Joining process failed.
14 Starting new partition.
15 MLE : Node operating as Parent.
16
17 [+] SNY process: Send Link Request to neighbor router.
18 ==> MLE LINK REQUEST sent to : ff02::2
19 [+] SNY process:  Synchronization process finished.
20 <== MLE PARENT REQUEST received from : fe80::250:c2ff:fea8:b0
21 ID allocated for the child = 1
22 ==> MLE PARENT RESPONSE sent to : fe80::250:c2ff:fea8:b0
23 <== MLE PARENT REQUEST received from : fe80::250:c2ff:fea8:c0
24 ID allocated for the child = 2
25 ==> MLE PARENT RESPONSE sent to : fe80::250:c2ff:fea8:c0
26 <== MLE CHILD ID REQUEST received from : fe80::250:c2ff:fea8:b0
27 ==> MLE CHILD ID RESPONSE sent to : fe80::250:c2ff:fea8:b0
28 <== MLE CHILD ID REQUEST received from : fe80::250:c2ff:fea8:c0
29 ==> MLE CHILD ID RESPONSE sent to : fe80::250:c2ff:fea8:c0
30 Child linked with id : 1 and timeout is : 10
```



```
B
1 MLE UDP initialized :
2  lport --> 19788
3  rport --> 19788
4 MLE protocol initialized.
5 [+] JP Send mcast parent request to active router
6 ==> MLE PARENT REQUEST sent to : ff02::2
7
8 [+] JP Waiting for incoming response from active
9 router<== MLE PARENT RESPONSE received from :
1 fe80::ff:fe00:0
0 [+] JP received response from active Router
1 my rssi : 30
1 rssi of parent : 30
1 two-way link quality : 3
2 <== MLE PARENT REQUEST received from :
1 fe80::250:c2fffea8:c0
3 [+] JP Parent selection
1 link quality with parent : 3
4 [+] JP send Child ID Request
1 ==> MLE CHILD ID REQUEST sent to : fe80::ff:fe00:0
5 <== MLE CHILD ID RESPONSE received from :
1 fe80::ff:fe00:0
6 [+] JP Parent Stored
1 MLE : Node operating as Child
7 <== MLE PARENT REQUEST received from :
1 fe80::250:c2ff:fea8:b1
8 <== MLE PARENT REQUEST received from :
1 fe80::250:c2ff:fea8:b1
9 ID allocated for the child = 1
2 ==> MLE PARENT RESPONSE sent to : fe80::250:c2ff:fea8:b1
0 <== MLE CHILD ID REQUEST received from :
2 fe80::250:c2ff:fea8:b1
1 Sending request to become a Router
2 MLE : Node operating as Parent.
```

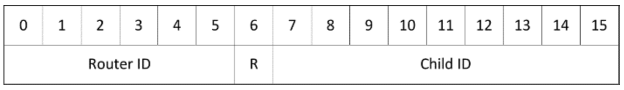
Figure 5. Trace output of Thread nodes.

4. Network Layer

In this section, we provide information regarding the Thread network layer. Each Thread device must at least support the IEEE 802.15.4 (2006) standard but might support additional interfaces. This makes the Thread protocol appear heterogeneous from a PHY layer point of view. Above the MAC layer, Thread turns into a homogeneous protocol supporting IPv6 and IP routing.

4.1. EID / RLOC separation

In a Thread network, each interface must have a routing locator (RLOC) address and an endpoint identifier (EID) address. The RLOC address is based on the router ID and child ID assigned to an interface. If for example one child switches over to another parent, only the RLOC will change, but not the child's identity at all. Thread nodes are arranged in partitions forming closed affiliation areas for participating devices whereas each partition has one node designated as the leader. The leader is responsible for assigning and managing router IDs. A Thread EID is a stable IPv6 address that uniquely identifies a Thread interface within a Thread partition. EIDs are not directly routable, because the Thread routing protocol only exchanges route information for RLOCs. To deliver an IPv6 datagram with an EID as the IPv6 destination address, a Thread device must perform EID-to-RLOC lookup. When attaching to a partition, a node must retrieve an RLOC IPv6 address from a router. The RLOC's 16 least significant bits are called RLOC16 and map router ID and child ID of the node. Routers assign child ID 0. Figure 6 shows the RLOC16 structure.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Router ID						R	Child ID								

Figure 6. RLOC16 structure.

A router retrieves his router ID from the partition leader by sending a CoAP address query message. RLOC addresses are only used for communicating control traffic and delivering IPv6 datagrams to their destinations. Since no RLOC address is available when initially sending an address query message, the EID is used. Intermediate nodes must perform EID-to-RLOC lookup in order to forward the packet to the partition leader and vice versa. The child ID is allocated by the parent node and communicated through MLE attachment process.

4.2. Routing algorithm

A Thread network has up to 32 active routers that use next-hop routing for messages based on its routing database. This database includes path cost calculation that is performed by applying distributed Bellman-Ford algorithm (cf. RIPng) [10]. The routing database is a set of neighbor router table (Link Set), routing table (Route Set) and all valid router IDs (Router ID Set). All routers advertise their routing table periodically. The rate at which routing advertisements are sent is determined by an instance of the Trickle algorithm. For routers, in order to keep track of the validity of shared data in the network, an incremental ID sequence number is attached to the routing data. After looking up the shortest path for a route, a router generates the IPv6 RLOC address of the destination router using its router ID.

All tables that are part of the routing database have been implemented by using linked lists. When looking for a routing entry, linked list structures can be used as a mask when iterating trough all list entries. The benefit of this approach is that predefined fields can be accessed easily. Since embedded devices usually underlie memory constraints, we implemented least recently used (LRU) replacement policy for Link Set and Route Set. The last recently accessed item is inserted at the head of the linked list by modifying appropriate pointers. As a result, when transmitting fragmented packets the lookup iteration for subsequent fragments will terminate after the first list element. When inserting a new list element, the last one of the linked list is removed if the number of elements would exceed a defined maximum.

4.3. Link cost determination

The link set stores information about neighboring routers including the measured link margin (RSSI) in dB. Furthermore, the link margin plays a leading role in parent selection during attachment process. The measured one-way link margin may change during runtime due to noise floor or altered environmental conditions. To smooth out short-term volatility, Thread devices must perform exponentially weighted moving average (EWMA) method of the link margins for each neighbor. Equation (1) shows the EWMA calculation where M_{t-1} is the currently stored link margin for a specific neighbor, Y_t is the last recently measured link margin and M_t will be the newly calculated link margin for that neighbor.

$$M_t = \alpha \cdot Y_t + (1 - \alpha) \cdot M_{t-1} \tag{1}$$

To avoid costly floating point computations on the micro- controller, equation (1) has been rewritten to equation (2).

$$M_t = \frac{Y_t + \left(\frac{1}{\alpha} - 1\right) \cdot M_{t-1}}{\frac{1}{\alpha}} \tag{2}$$

The exponential smoothing fraction α (equation (3)) is used as weighting and defined as either $\frac{1}{8}$ or $\frac{1}{16}$ [11].

$$\alpha = \{\alpha \in R \mid 0 \leq \alpha \leq 1\} \tag{3}$$

The preceding transformation results in a bit shift to the left of the numerator around the reciprocal value of α. This allows to exploit the benefits of integer calculations without glaring rounding errors.

4.4. Routing advertisement

Distributed routing algorithms reduce node-sided computational costs in terms of shared route data. When using non-distributed algorithms, each node has to expand a graph by incrementally improving path costs. In Thread, MLE advertisements are used from nodes acting as routers for advertising their routing table to neighboring routers. A practicable approach of determining the rate at which advertisements are sent is to deploy a dependency on the change of routing data. Thread uses the Trickle algorithm to generate dynamic and random transmission windows. If the routing entries are stable, the rate is reduced to a minimum. The flowchart in figure 7 shows our implementation of the Trickle algorithm. We use a timer that recalculates its expiration time after a timeout. The limits of the time slots can be defined via C macros. After initializing the Trickle timer, it runs independently from other processes.

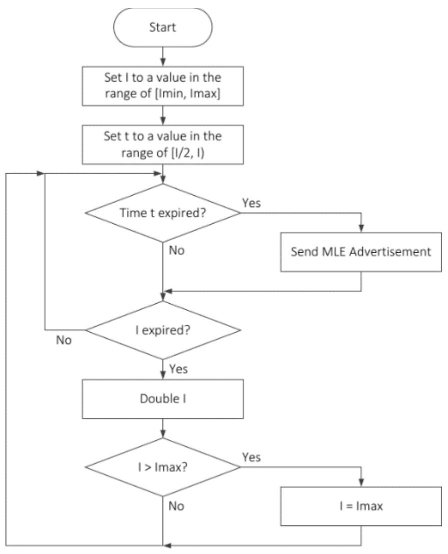


Figure 7. Flowchart of Trickle timer implementation.

4.5. Unicast packet forwarding

Routing inside a Thread network is performed using RLOC IPv6 addresses. Unicast packets are forwarded by applying a mesh under strategy on the 6LoWPAN layer. Router packets include the 6LoWPAN mesh header carrying the originator and final RLOC16 addresses. When receiving a packet including a 6LoWPAN mesh header, a routing table lookup is performed without uncompressing the packet. Therefore, we extended the emb::6 implementation in order to support mesh under routing for unicast packets. During MLE joining process, the 16-bit short MAC address is set to the RLOC16 assigned to the router. Usually, IPv6 packets from higher layers, e.g. application layer, are using the EID of the destination device. Then, routers must perform EID-to-RLOC lookup to retrieve the router ID of the destination router. The EID-to-RLOC lookup mechanism consists of CoAP messages targeting CoAP resources provided by routers [12]. The router that is responsible for the given EID is sending a response message including its router ID. Each router maintains a EID-to-RLOC map cache to hold a list of recently used lookups. This prevents from frequently sending lookup messages when transmitting fragmented packets. For instance, an end device does not have routing capability and therefore must forward packets to its parent router. In this case the packet is sent without adding 6LoWPAN mesh header.

5. Memory usage

An analysis of our implementation in terms of memory usage is shown in table 1. Both figures show the RAM and flash memory usage for selected network layer modules and MLE. Mesh under routing is enabled on 6LoWPAN layer. The 'net-thread' bar shows all additional Thread-specific implementations such as routing database, partitioning etc. The reason for the high memory usage of this part is the static memory allocation of the routing database.

Table 1. Thread implementation memory usage.

Feature	RAM (kB)	Flash (kB)
net-ipv6	4,056	25,025
net-sicslowpan	460	14,826
net-thread	12,692	33,014
mle	2,088	15,638

6. Conclusion

This paper presents the architecture and basic information about the Thread protocol. The current status of the implementation allows to create nodes inside a hardware virtualization environment dynamically. Thread nodes autonomously establish connection and create network topologies as specified by Thread algorithms. An important next step will be the full integration of Thread Network Data and commissioning protocol in order to validate the compatibility with existing Thread devices.

References

[1] G Montenegro, N Kushalnagar, J Hui, D Culler, *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*, RFC 4944, (Sept. 2007)
[2] Thread Group, Inc., *Thread Specification*, Revision 1.1.0 (July 2016)
[3] A Yushev, A Sikora and J Sebastian E, *Open source 6Lo protocol stack for wireless embedded systems*, 2016 Wireless Telecommunications Symposium (WTS), London, UK, 2016, pp. 1-7.
[4] M Schappacher, E Schmitt, A Sikora, P Weber, A Yushev, A Flexible, Modular, *Open-Source Implementation of 6LoWPAN*, 8th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS2015), 24-26 September 2015, Warsaw, Poland, pp. 838-844.
[5] A Sikora, *Funknetzwerke für das Internet der Dinge: 6LoWPAN OpenSource-Projekt: emb6*, Elektronik Wireless 2016, (2016)
[6] Thread Group, Inc., *Thread Stack Fundamentals (whitepaper)*, Revision 2.0 (July 2015)
[7] B Curtis, S Ashon. *Thread Open House*. Thread Group, (May 2016)
[8] R K Kelsey, *Mesh Link Establishment*, (Oct. 2013)
[9] ARM mbed 6LoWPAN Stack Overview, https://docs.mbed.com/docs/arm-ipv66lowpan-stack/en/latest/02_N_arch/, (04.02.2017)

[10] Malkin, G and R Minnear, *RIPng for IPv6*, RFC 2080, (Jan. 1997)
[11] T Agami Reddy, *Applied data analysis and modeling for energy engineers and scientists,* in. Boston, MA: Springer US, pp. 253–288 (2011)
[12] Z Shelby, K Hartke, C Bormann, *The Constrained Application Protocol (CoAP),* RFC 7252, (2014)