Tock OS | Thread Protocol Implementation Status 8/31/2017

Hubert Teo, Paul Crews, Mateo Garcia

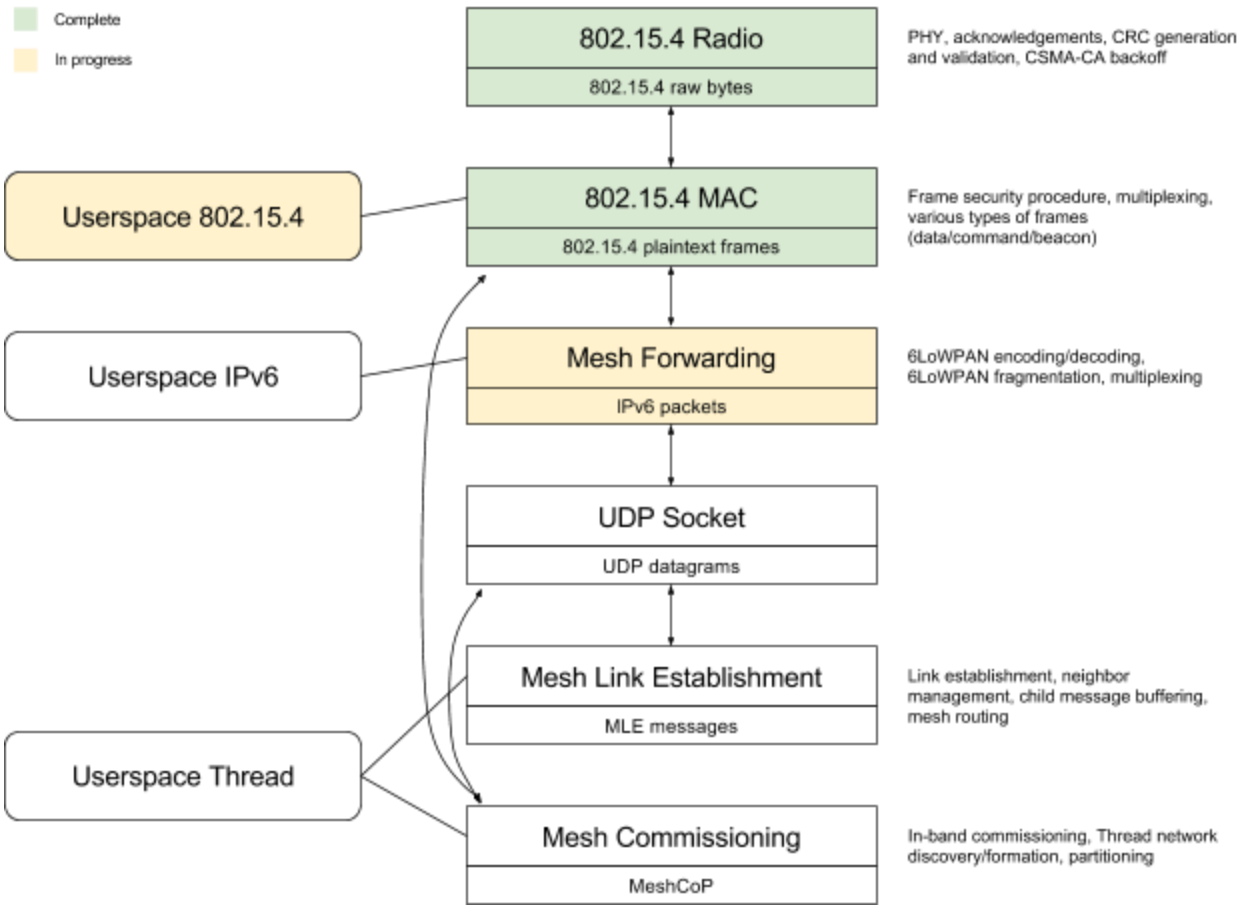
The original goal of our CURIS project this summer was to add low-power wireless networking support to Tock OS. We implemented IPv6 packet transmission over an IEEE 802.15.4 radio using the 6LoWPAN packet compression scheme. The end goal is for Tock to support the Thread protocol developed by Nest Labs, which defines a mesh networking protocol that operates over 802.15.4 and 6LoWPAN.

What follows is a documentation of the current state of the Thread protocol implementation in Tock. We describe modules and tests that have been written, issues yet to be solved, and functionality that is yet to be implemented.

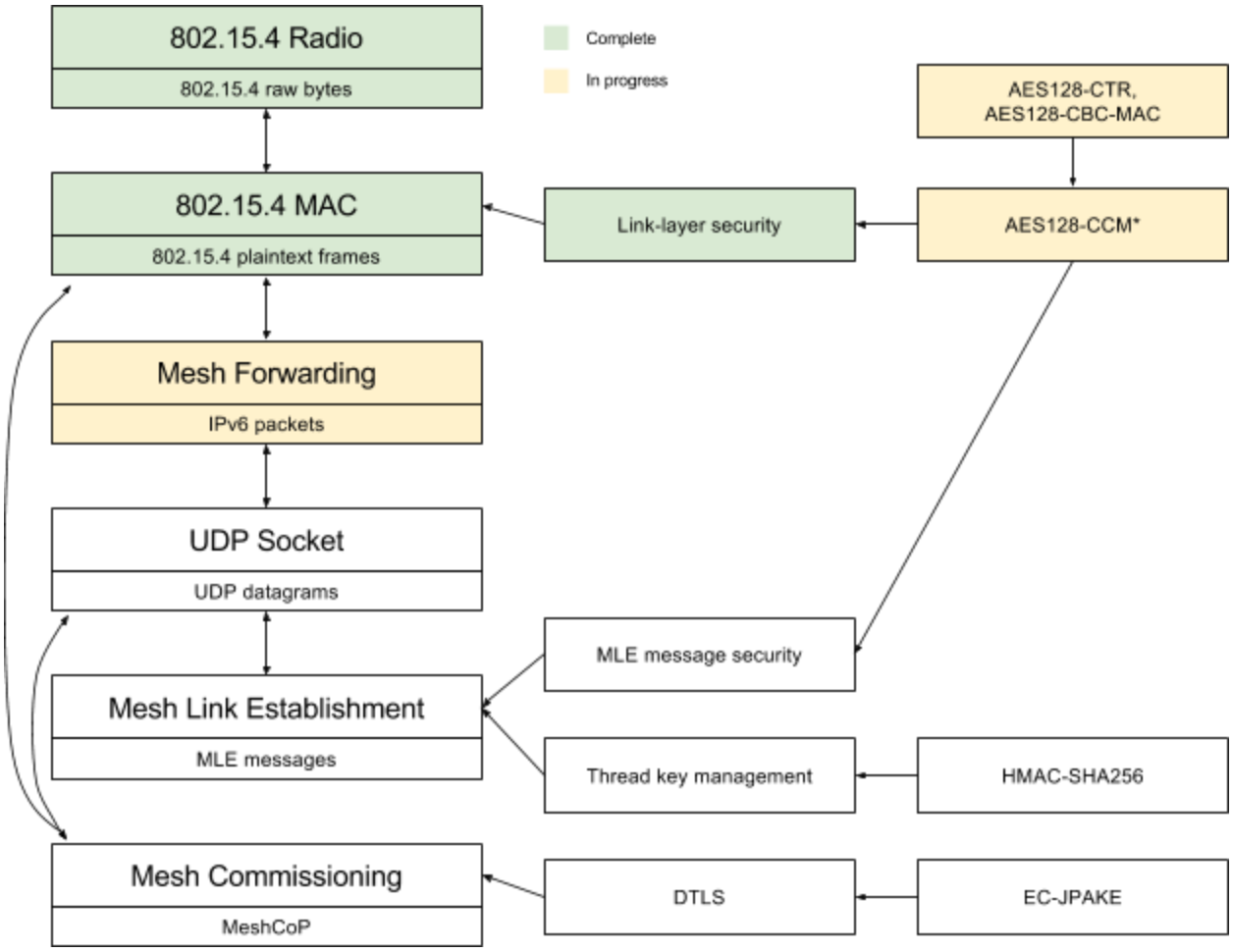
Overview

Thread is a mesh networking protocol that builds on a very large set of pre-existing technologies. These include networking layers, their corresponding userspace interfaces, cryptographic schemes, and frame/packet formats for encapsulating data and metadata. Here is an overview of the relationships between the subsystems required for full Thread support and a rough gauge of their status of completion.

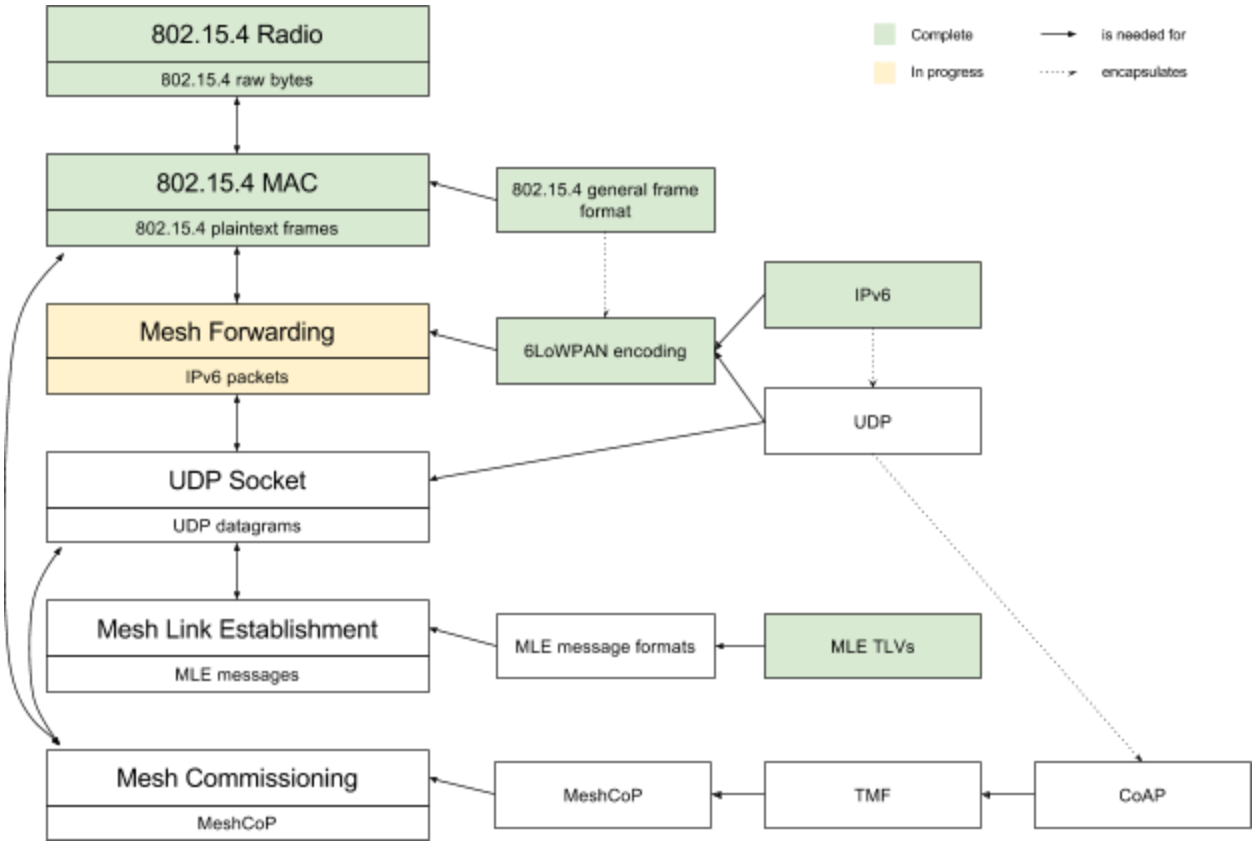
Networking Layers



Cryptographic Primitives



Header Formats



Summary

We have not been able to implement much of the Thread networking protocol itself, but our work covers a large part of the significant engineering effort required to write a networking stack from the ground up in Tock. Furthermore, our work paves the way for Thread in Tock by providing a clear reference point and implementation roadmap.

As is clear from the above diagrams, each networking layer relies on having access to implementations of two broad categories of subroutines: cryptography and header encoding/decoding. We suspect that this need for sufficiently flexible and extensible cryptography and header encoding/decoding functionality is not specific to Thread, but instead to networking in general. Hence, it would be ideal to build general-purpose frameworks for these two needs into Tock. We have already taken the first step of producing such a framework for header encoding/decoding using macros, but it must be further tested against more header formats before we can be sure it is sufficiently general. A similar framework is in the works for cryptography.

Changes

- IEEE 802.15.4 (#607)
    - capsules/src/rf233.rs: 802.15.4 radio, talks to RF233 over SPI
    - capsules/src/ieee802154/mac.rs: 802.15.4 MAC layer on top of a radio
    - capsules/src/ieee802154/driver.rs: Userspace driver for 802.15.4
    - capsules/src/ieee802154/virtual_mac.rs: MAC virtualization/mux
    - userland/libtock/ieee802154.{c|h}: 802.15.4 userspace interface
    - userland/examples/tests/ieee802154/radio_{rx|tx|rxtx|ack}: Tests
- 6LoWPAN (#581)
    - capsules/src/net/lowpan.rs: 6LoWPAN compression/decompression
    - capsules/src/net/lowpan_fragment.rs: 6LoWPAN fragmentation
    - boards/imix/src/sixlowpan_dummy.rs: 6LoWPAN compression, decompression and transmission test
    - boards/imix/src/lowpan_frag_dummy.rs: 6LoWPAN fragmentation test
- Thread (merged)
    - capsules/src/net/thread/tlv.rs: MLE TLV support
- Header encoding/decoding
    - capsules/src/net/stream.rs: Generalized header encoding/decoding framework with Rust macros
    - capsules/src/net/ieee802154.rs: 802.15.4 header format support
    - capsules/src/net/ip.rs: IPv6 header support (does not use the above framework)
    - capsules/src/net/utils.rs: Utilities for IPv6 address prefix matching
- Kernel changes
    - kernel/src/common/list.rs: Some additional list manipulation methods
    - kernel/src/common/take_cell.rs: and_then() for TakeCell/MapCell
- Increasing ROM region to accommodate code size
    - boards/imix/chip_layout.ld: Increase size of ROM region
    - boards/imix/Makefile-app: Board-specific makefile to change app offset when calling tockloader
    - userland/tools/flash/imix.sh: Change app offset here too

IEEE 802.15.4

Thread operates with the 6LoWPAN standard over IEEE 802.15.4 links. While Tock already had a working implementation of an 802.15.4 radio (using the AT86RF233 radio transceiver chip), it turned out to be rather incomplete upon closer inspection. Namely, its header parsing and construction implementation (and hence, the frame-base interface it exposes) was insufficiently general to support the needs of Thread. Additionally, the IEEE 802.15.4 security procedures for incoming and outgoing frames were not at all supported. Link-layer security is mandatory in a Thread network. Hence, we implemented a MAC layer abstraction on top of the existing code that interfaces with the RF233 hardware, replacing the existing frame management code and exposing a cleaner interface.

General Frame Format

IEEE 802.15.4 frames all fall under a unified general frame format. However, there are several different types of frames (data, MAC command, beacon) that use different subsets of the general frame format. Beacon and command frames also have additional structure in the payload.

Our implementation of general frame format support is able to encode and decode byte buffers containing 802.15.4 frames into a Rust struct with easily accessible fields. There is one complication in this setup, which has to do with the frame format itself. 802.15.4 frames may contain any number of Information Elements (IEs), and some IEs are considered as part of the header and others as part of the payload. Together, they form a variable-length sequence of IEs straddling the header-payload boundary. We store this sequence of IEs as entries in a fixed-size array in the header struct, arbitrarily setting a limit on the maximum number of IEs in a header.

We choose to parse these IEs and include them in the header struct for convenience. However, since the payload is encrypted in secured frames, this means that we cannot attempt to parse payload IEs from secured frames, and must instead re-parse them when frames are unsecured. This logic is implemented as a flag into the frame header decoding process.

The resulting interface makes it easy to prepare and parse 802.15.4 data frames during transmission and reception. While there is no support yet for MAC beacon and command frames, the header parsing library can easily be extended to do so by just adding encoding/decoding support for superframe specifications and command IDs.

Link-layer Security

Almost all parts of the IEEE 802.15.4 link-layer frame security procedures are implemented in the MAC layer. The remaining parts are the device and key management that are to be determined by an upper layer (Thread), and encryption. Device management is necessary because that the frame security procedures rely on MAC extended addresses, which may not

always be contained in the frames. Instead, a “device” is identified by its MAC short address or other frame metadata, and a mapping between “devices” and their extended MAC addresses needs to be maintained. For now, a minimal device and key management mechanism is implemented in the userland interface. For encryption, support can easily be added once there is AES-CCM* support in Tock.

Virtualization and Userspace Interface

We wrote a sequencing multiplexer to virtualize the 802.15.4 interface (so that there can be both a userspace interface for sending raw frames, and a mesh forwarder that sends IP packets over the same interface). This multiplexer queues up transmission requests (one per client) arbitrarily, but exposes all received frames to all clients, as if in promiscuous mode. This is so that each client can implement its own filtering mechanism. In the future, we might need to extend the queueing system to have a notion of priority, since certain Thread network commissioning protocols take precedence over data transmission.

There is also a minimal userspace interface for sending raw 802.15.4 frames. It exposes a list-based mechanism for configuring device and key management, and replicates some frame inspection routines in userland so that Tock applications can work with 802.15.4 frames easily.

Tests

There are four sample Tock applications that test the combined functionality of the 802.15.4 MAC layer and radio: radio_tx, radio_rx, radio_rxtx and radio_ack. These respectively transmit, receive, echo, and verify acknowledgements frames.

6LoWPAN

6LoWPAN is a popular format for transmitting IPv6 packets over 802.15.4 links. The primary challenge of porting IPv6 to low power devices is the large minimum MTU required by the protocol, which conflicts with the relatively small MTU of 802.15.4 links. Further, IPv6 packet headers are large, and consume a substantial portion of the usable space in a 802.15.4 frame. 6LoWPAN solves these problems by using header compression and fragmentation, both of which we implemented for Tock. The full 6LoWPAN protocol is specified in RFC 4944 and the compression format is updated by RFC 6282. Our implementation uses the updated compression format given by RFC 6282, and additionally implements the fragmentation specification given in RFC 4944.

Compression and Decompression

In order to reduce the size of IPv6 headers, 6LoWPAN defines a fairly complex encoding and compression scheme. Although initially defined in RFC 4944, the header compression scheme was updated by RFC 6282, and the latter is used in the Thread protocol specification. For this reason, we implemented the header compression and decompression scheme specified in RFC 6282. The code for this implementation can be found in the `tock/capsules/src/net/lowpan.rs` file. Logically, the 6LoWPAN functionality is treated as a library instead of a well defined layer, with the transmit and receive paths conditionally compressing or decompressing headers before sending. This library primarily exposes the `compress` and `decompress` functions, but also exposes utility functions and ways to manage the list of contexts needed for some address compression schemes. The current 6LoWPAN compression/decompression routines are written without the header encoding/decoding framework used for 802.15.4 frames and MLE TLVs.

At this point, support for RFC 6282 header compression has been implemented in Tock. However, next header compression has not been extensively tested, and should be regarded as not fully reliable.

Fragmentation

Although 6LoWPAN header compression makes it easier to send IPv6 packets over 802.15.4 links, often the full packet will not fit in a single frame. Thus, 6LoWPAN defines a fragmentation and reassembly process, operating at a layer below IP. The fragmentation layer is defined in RFC 4944.

At this point, fragmentation has been implemented and tested for Tock. Although the required functionality is there and has been tested, some bugs or unimplemented features remain. For a full list of these issues, consult the Remaining Work section below.

Testing

We implemented and relied on several distinct testing strategies to confirm that the 6LoWPAN implementation functioned correctly.

Wireshark

The most direct way used to test the 6LoWPAN implementation is by using the TI CC2531 dongle with the tool found here: https://github.com/andrewdodd/ccsniffpiper. This piped all packets from the TI dongle to Wireshark, which is able to decode 6LoWPAN and 6LoWPAN fragmentation packets. This served as a quick and efficient way to determine whether the packets were being sent correctly. Note however, that Wireshark is unable to understand the 2015 802.15.4 frame format, and the 2006 format must be used instead.

Linux Kernel Module

Another means by which we tested the interoperability of our 6LoWPAN implementation was by using the Linux kernel. A 6LoWPAN implementation exists inside the Linux kernel (4.10+), and there exists a dummy loopback module (`fakelb`, installed via `modprobe fakelb`) which allows us to inject IEEE 802.15.4 frames into the kernel by creating two layer 2 802.15.4 interfaces (wpan0, wpan1) that transmit packets between themselves. To inject the frames, we relied on a TI CC2531 sniffer, which gave us raw 802.15.4 frames that we could then send to the layer 2 IEEE 802.15.4 interface (wpan0). We can then create a layer 3 IPv6 interface (lowpan1) on top of the other side of the loopback driver (wpan1), and receive the decompressed IPv6 packets. This allows us to quickly verify against a reference implementation that our compression scheme functions correctly. The full code for this can be found here: https://github.com/ptcrews/pyCCSniffer.

Executing the following lines will initialize the required interfaces:

```
$ modprobe fakelb  // Adds 802.15.4 interfaces wpan0, wpan1
$ iwpan dev wpan0 set pan_id [PAN ID]
$ iwpan dev wpan1 set pan_id [PAN ID]
$ ip link add link wpan0 name lowpan0 type lowpan  // Adds IPv6 interface lowpan0
$ ip link add link wpan1 name lowpan1 type lowpan  // Adds IPv6 interface lowpan1
$ ip link set wpan0 up
$ ip link set wpan1 up
$ ip link set lowpan0 up
$ ip link set lowpan1 up
```

We can then ping via the link local address (found via `ip addr`) as follows:

```
$ ping6 [link local address]%[interface to send from]
```

Note that both the destination IP and MAC address must refer to the lowpan1 interface. This is because the wpan0 and wpan1 interfaces do no processing on the packets they receive

whatsoever, and so the lowpan1 interface receives the original single IEEE 802.15.4 frame injected.

It appears that Linux kernel NHC and IPHC for 6LoWPAN may be different from how it is written in Tock. This could explain why fragmented IPv6 packets are dropped by Linux.

It should be possible to test decompression by crafting an IPv6 packet, sending it via the layer 3 interface (lowpan1), reading out the data sent to the layer 2 interface (wpan0), then transmitting it to the Imix for decompression. We did not implement this because we chose to wait until Tock had userspace support for writing and receiving over USB.

On-Chip Code

Once minimal compression/decompression functionality was tested, we moved to on-chip testing between Imix boards. We wrote two separate test files, `boards/imix/src/sixlowpan_dummy.rs` and `boards/imix/src/lowpan_frag_dummy.rs` to test 6LoWPAN compression/decompression and 6LoWPAN fragmentation respectively. The first test, `sixlowpan_dummy.rs`, constructs a number of different IPv6 packets, compresses them, then decompresses them and checks that they are the same. If they are, then the compressed version is sent. No additional testing on the receive side is required, but we can confirm interoperability by sniffing via Wireshark. For the other test suite, `lowpan_frag_dummy.rs`, we again craft a number of IPv6 packets deterministically, then call the fragmentation send function. On the receive side, we craft the same IPv6 packet, then verify that the received/reassembled packet matches. Note that this test suite requires two Imix boards, one running the transmit version and the other running the receive version.

Remaining Work

There remains some additional testing to be done and features to be added. Most of the remaining work relates either to testing or to additional features required for full correctness.

Testing

Although there currently exists code for next header compression and decompression, we have not substantially tested this functionality. Furthermore, IPv6 encapsulation and UDP header compression has not been tested at all, and compression/decompression relying on shared context (context-dependent) has only been minimally tested. In order to correctly interoperate with arbitrary devices, these untested features should be tested further.

Features

Specific to 6LoWPAN, there are a number of minor features that have not been fully implemented. One of the most critical is ensuring that the fully compressed header fits within a single IEEE 802.15.4 MTU. This is implied by the specification, but we do not check conformance. Additionally, we need to correctly use the right address(es) for compressing and

decompressing encapsulated IPv6 headers. Finally, for compression performance, we should also modify the compression code to use the better of context-dependent and context-free address encoding schemes - currently, we always use link-local compression even if context-dependent encoding gives better compression.

Known Bugs

There exists only one substantial, known bug in the current code. In running the `lowpan_frag_test.rs` test suite, the middle fragment's payload becomes corrupted in some of the tests (between 2 and 3 tests out of 25). The end few bytes of the payload (anywhere between 1 and 8 bytes) are overwritten with what appears to be radio information (signal strength, etc.). Although the failing tests are somewhat deterministic, when changing which test executes first, which tests fail also change. The bug does not appear to be in the 6LoWPAN or MAC code, as the RF233 callback passes back the corrupted frame. Currently, we believe the bug is in either the SPI code or the RF233 code. Note that although the bug seems to be some kind of race condition, it happens on the same tests with almost identical bytes (given the same starting conditions), and thus does not appear to be greatly influenced by execution order.

Thread

MLE

Our intended deliverable was the ability to connect a Sleepy End Device (SED), as defined in Thread, to a Thread network. In Thread this is done using mesh link establishment (MLE). MLE for network attaching is a four-step handshake that works as follows:

1. A child device, in our case an SED, multicasts a Parent Request.
2. Each potential parent device on the network, classified as a Router in Thread, unicasts a Parent Response.
3. The child device selects a parent based on a hierarchy of connectivity metrics, and unicasts a Child ID Request.
4. The selected parent unicasts a Child ID Response.

MLE messages consist of an MLE command type followed by a series of Type-Length-Value (TLV) parameters.

TLVs

TLVs are used to serialize the configuration values exchanged during MLE, including link-layer (Layer 2) addresses, transmit and receive modes, and security parameters. TLVs can be nested within TLVs. The capsules/src/net/tlv.rs module, as it stands, supports the encoding and decoding of the minimum subset of TLV types used in the MLE process for an SED.