UNIVERSITY OF CALIFORNIA SAN DIEGO

Interface Design and Resource Policies for Networking in Embedded Operating Systems

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science

by

Tyler Potyondy

Committee in charge:

Professor Pat Pannuto, Chair
Professor Deian Stefan
Professor Geoffrey Voelker

2023

The Thesis of Tyler Potyondy is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to acknowledge the mentorship and support of Pat Pannuto in my work.

I would like to thank Geoffrey Voelker and Deian Stefan for their willingness to serve on my committee.

ABSTRACT OF THE THESIS

Interface Design and Resource Policies for Networking in Embedded Operating Systems

by

Tyler Potyondy

Master of Science in Computer Science

University of California San Diego, 2023

Professor Pat Pannuto, Chair

Thread networking's emergence as the de facto low-power IP networking technology warrants investigating a Thread interface within embedded operating system network stacks. Although there exist many embedded software platforms providing Thread integration, these examples notably lack multi-tenancy and a principled policy for interface design and resource sharing. This work first provides a survey of leading embedded operating systems' networking and Thread capabilities followed by a proposed Thread networking interface design. To demonstrate the interface design, kernel Thread networking support was implemented within TockOS and provided functional interoperability for a Tock child device to attach to an OpenThread router.

Introduction

The need to communicate information across physically disjoint locations dates back millennia. The ancient Chinese are well known for the impressive magnitude and length of the Great Wall of China. Often forgotten, however, is their pronounced communication innovations allowing a message to pass 7,300 kilometers in only one hour. This feat was accomplished through the use of light to signal between beacon towers; information was encoded in colored smoke and the number of lanterns illuminated [1].

Modern sensor networks allow for the measurement and aggregation of data at an unprecedented scale. This scale and ability for widespread measurements is achieved through the use of inexpensive microcontrollers coupled with sensors to perform measurements and wireless radios to transmit this data. Being an inexpensive microcontroller, however, creates numerous deployment challenges as these inexpensive devices are severely resource constrained. The most scarce resources are the device's limited memory (order 64kB) and limited power budget (as sensor nodes are typically battery powered).

Novel network technologies, such as Thread networking, are purpose built for the constrained power budget of battery powered devices [2]. Because Thread is purpose built for low-power devices, it has gained immense popularity among smart home, and sensor network devices. These systems require wireless communication while also being able to primarily deep-sleep and cease radio communications [3]. Prior to Thread, these low-power devices all deployed custom wireless networks with costly and cumbersome supporting infrastructure. Thread solves this problem and allows devices to act as full-fledged IP endpoints while also providing the means for these devices to connect to the Internet. Given Thread's popularity, many embedded software

platforms have worked to provide Thread networking support. Thread, however, is unlike IEEE 802.15.4 or BLE as it is a networking policy that exists within, alongside, and above the UDP / IPv6 / 6LoWPAN / IEEE 802.15.4 networking stack. This presents potential multi-tenancy concerns and design challenges surrounding how best to interface a Thread implementation within an existing network stack that may also be used by other non-Thread clients.

Subsequently, we investigated a Thread networking interface and implemented the proposed design within the embedded operating system Tock. We evaluated the success of this implementation by confirming interoperability between a Tock child device and an OpenThread router node. Ultimately, we achieved consistent attachment of the Tock child device to the OpenThread router. This demonstrated the implementation's interoperability with a production grade Thread network.

Chapter 1

Background and Related Work

1.1 Thread Networking

1.1.1 Overview

Thread networking *(hence forth referred to as Thread)* is a wireless communication protocol purpose built for IoT devices. Thread offers IPv6 interoperability, a robust mesh topology, low-latency data transfer, and low-power operation. The Thread standard is governed by the Thread Group—a not-for-profit organization composed of Thread stakeholder companies and institutions that develop products utilizing Thread. Thread currently powers the wireless communications of Google Nest products, numerous Amazon smart home products, agriculture livestock monitoring, and various additional smart-home/smart-building products [2]. Notably, Thread has also been selected as the communication standard to work alongside WiFi to power Matter, a novel communication standard meant to unify smart devices and provide interoperability across manufacturers [4]. Currently, Google's OpenThread serves as the de facto implementation of the Thread standard.

1.1.2 Technical Description

Thread utilizes a UDP / IPv6 / 6LoWPAN / IEEE 802.15.4 networking stack to form, maintain, and send messages across a mesh topology. Thread's network topology, node types, security strategy, and use of the aforementioned networking layers are described next.

Network Topology and Node Types

A Thread network is composed of routers and end devices. Together, these devices form Thread's mesh network. Router nodes must track the network's state and store the addresses of both other network routers and connected end devices. Together, these routers communicate with each other and work to maintain the mesh network topology. Furthermore, these routers facilitate the communication of non-adjacent end devices. Figure 1.1 demonstrates a Thread network topology comprised of routers and end devices. Router devices are often referred to as "parents" and end devices are often referred to as "children".

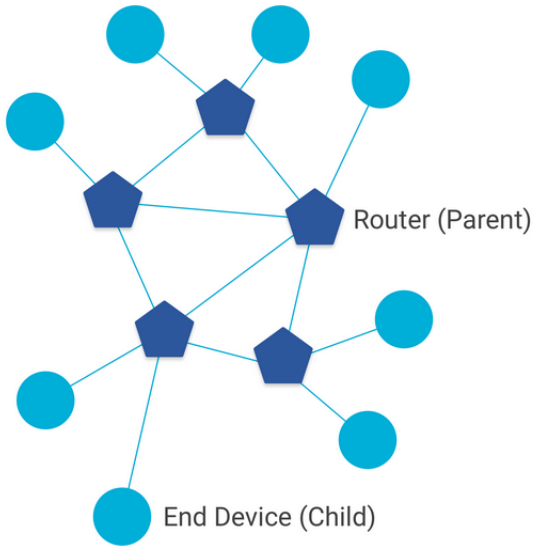

Figure 1.1. Depiction of Thread mesh network topology with routers and end devices—**Figure From** [5].

Thread's division of nodes into children and parents is crucial for providing low power operation. For many networked, power-constrained devices, the majority of the power budget is spent operating the radio. Of this, the vast majority of radio energy is spent not on useful data transmissions, but rather on trying (and failing) to receive just in case someone happens to send a packet—the "idle listening" problem. To operate in a low-power profile, devices must suspend radio operations. However, this creates challenges as the device will not receive packets sent

during this inactive interval. Although this can theoretically be mitigated through resending a packet until receiving an acknowledgment, this is a highly inefficient use of sender resources and may lead to these resend attempts saturating the RF band and colliding with other packets. Thread solves this with an asymmetric design. Thread routers take on the idle listening burden, which allows for tightly efficient child devices; a low-power device can register as a sleepy end device to a router, negotiate between the router and child a "sleeping" time period in which the radio is powered off, and the router will queue and then delay sending all messages destined for the sleepy end device until the sleepy end device is "awake" (e.g. radio is powered on).

Figure 1.2 depicts the available Thread node subtypes. Each Thread node can be configured as either a Full Thread Device (FTD) or Minimal Thread Device (MTD). Typically, FTD do not possess power concerns while MTD are often power constrained (i.e. battery powered). Depending on the current topology, Thread routers may promote a FTD child to a router to promote a healthy topology of routers.

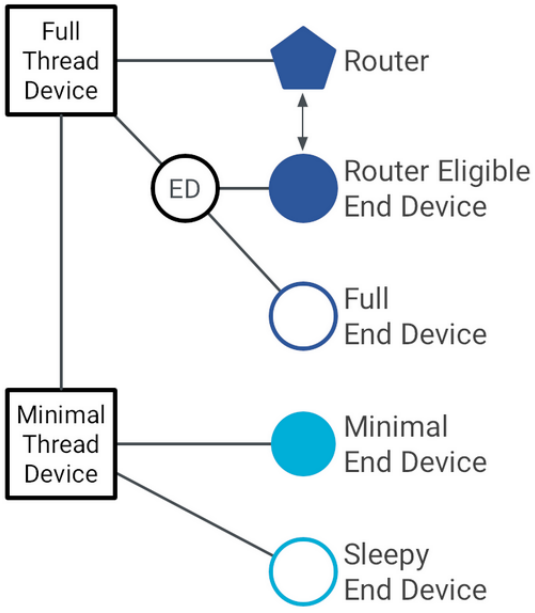

Figure 1.2. Depiction of Thread device types and their possible roles in the network—**Figure From** [5].

Child - Parent Attachment

Thread utilizes Mesh Link Establishment (MLE) messages to form, maintain, and join a Thread network. For a child device to attach / connect to a parent, a four part MLE handshake must occur. First, the joining device multicasts a Parent Request. In response to this parent request, routers and router eligible end devices (REEDs) that receive this parent request respond with a parent response. Next, the joining device parses the received parent responses and determines which responding router is optimal as prescribed in the Thread standard. After determining the optimal router to attach to, the joining child device sends to that router a child ID request. This router then completes the joining "handshake" by responding with a child ID response. Upon sending this child ID response, the router has officially added the joining device as a child. A visualization of this process is depicted in Figure 1.3. It is important to note that all Thread devices must join the network in this manner. They are then able to promote from a child to a router depending on the device specific configuration.

IEEE 802.15.4

IEEE 802.15.4 *(hence forth referred to as 15.4)* is a widely used networking standard. This section will not exhaustively describe 15.4, but instead will focus on how Thread uses this technology. 15.4 constitutes Thread's Physical (PHY) and medium access control (MAC) layers. Subsequently, any device implementing Thread must possess a 15.4 radio.

The primary components used in transmitting 15.4 messages are the sending of framed packets (i.e. data transmission) and acknowledgments of received frames. Acknowledgments serve an important purpose in notifying a sender if a recipient received a given packet. Additionally, Thread utilizes 15.4's security protocol for securing Thread packets. This will be discussed further in the security subsection.

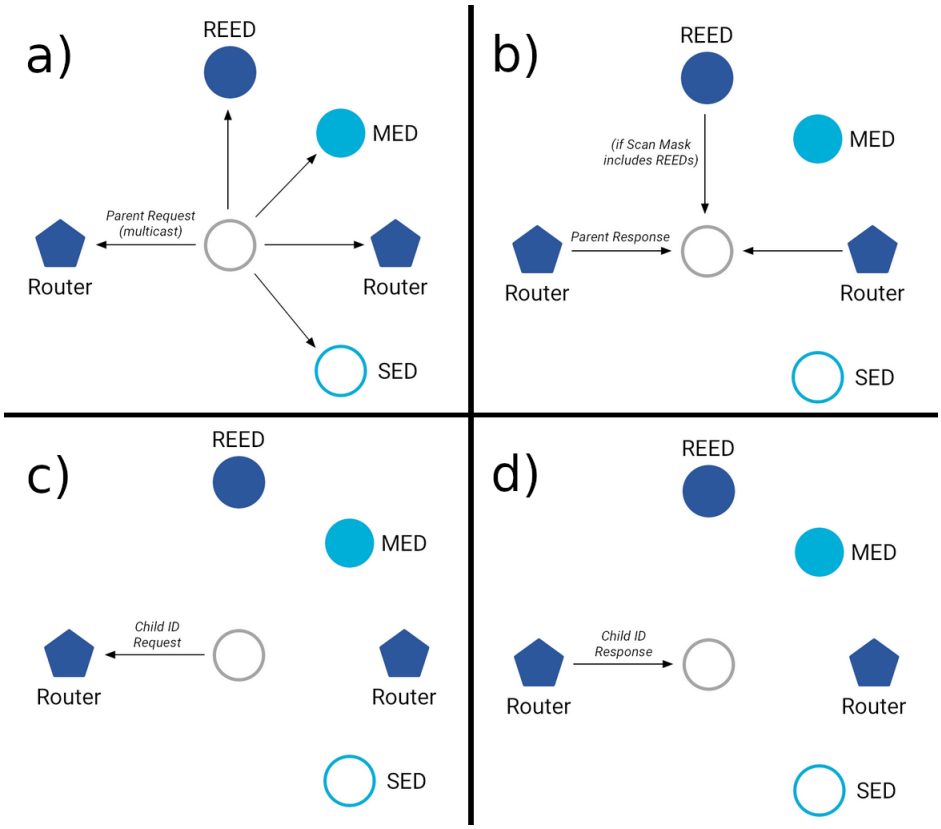

Figure 1.3. Steps required for a child to attach to a router and join a Thread network. **A)** Parent Request **B)** Parent Response **C)** Child ID Request **D)** Child ID Response—**Figure From** [6].

IPv6 / 6LoWPAN

IPv6 was instantiated to meet the growing demand for internet addresses in light of IPv4's limited supply. The expansive scale IPv6 addressing provides requires an increased number of bytes to encode addresses and subsequently creates 15.4 interoperability challenges. A 15.4 MAC layer frame holds at most 127 bytes—source and destination IPv6 addresses alone would consume 25% of every frame. 6LoWPAN was created to solve this challenge and provides an IPv6 compression scheme allowing IPv6 to be used efficiently with 15.4 PHY / MAC [7].

Thread utilizes both IPv6 and subsequently 6LoWPAN in order to be usable with 15.4. IPv6 allows each Thread node to be globally addressable by generating an IPv6 address tied to the devices' Extended Unique Identifier (EUI-64). In a Thread network, a node has three potential manners in which it can be addressed directly: link-local, mesh-local, and a global

address. These are depicted in Figure 1.4.

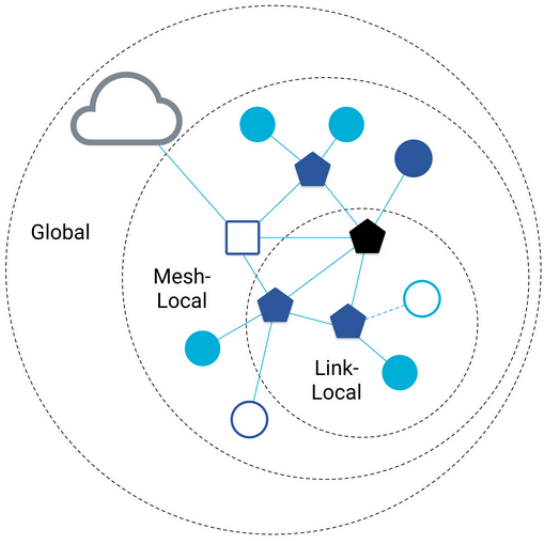

Figure 1.4. IPv6 addressing hierarchy—**Figure From** [8].

UDP

UDP is utilized to send MLE messages and can also be used to send data between nodes. MLE messages and data are encapsulated in the UDP packet's payload. Thread specifies the use of UDP port 19788 for all MLE messages. Thread UDP packets take one of two forms, dependent on if link layer encryption is used. In both cases the first payload byte denotes the encryption scheme. In the encrypted case, an auxillary security header provides the encryption details, and a message integrity code (MIC) is used to validate the received message. Both the encrypted and unencrypted cases are displayed in Figure 1.5. MLE messages are primarily organized by command type and subsequent Type-Length-Values (TLVs). Together, the specified command and encoded TLV data work to organize and control the Thread mesh network.

Security

Thread security operates in two modalities: MLE and MAC security. MAC security is provided using the prescribed 15.4 link layer security of AES in Counter with CBC-MAC Mode (CCM) [9]. In order to simplify MLE security, MAC security is reused and adapted with minor

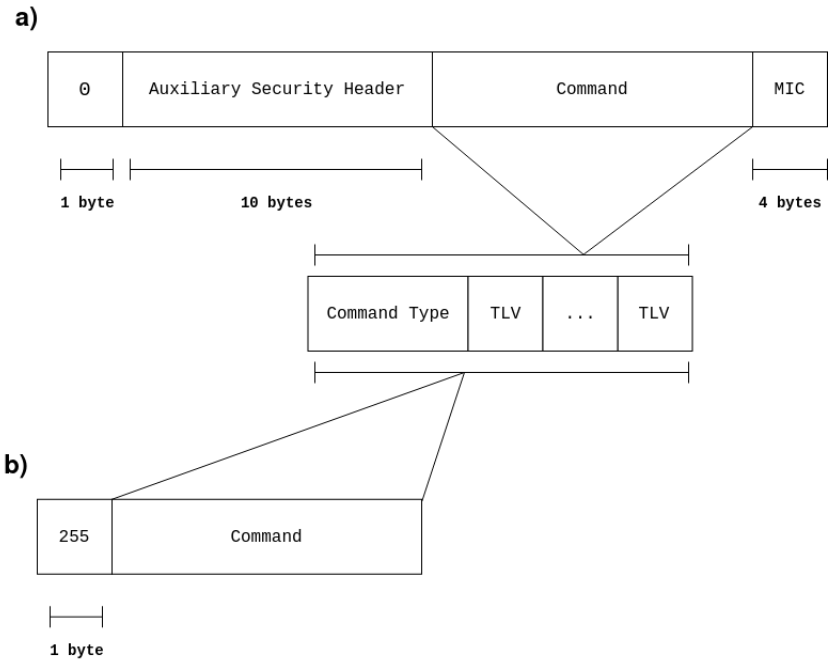

Figure 1.5. UDP Payload depiction. **A)** Encrypted UDP payload **B)** Unencrypted UDP payload. Both the unencrypted and encrypted UDP payload utilize the same format for the command.

modification to the authentication data. Moreover, different keys are used for MAC and MLE security. Thread's threat model primarily prevents eaves-dropping upon transmitted packets. MLE encryption and security fits within this model to primarily initialize link layer security. Furthermore, the Thread specification explicitly states that "data sent using UDP is not secured" and that DTLS must be utilized to secure UDP data [10].

Each Thread network has a unique network key that devices require in order to join the network. From this network key, the MLE and MAC keys are generated using a hashing protocol described in the Thread specification [10]. Together, MLE and MAC security ensure that only devices possessing the network key can decrypt and encrypt messages within a Thread network. However, this implies that if a Thread device is compromised and this network key is leaked, a malicious actor could observe all Thread traffic. Thread's threat model does not provide a solution to this fact, but instead recommends utilizing other forms of protection in addition to MLE and MAC encryption.

1.2 Embedded Software Platforms

1.2.1 FreeRTOS

General Overview

FreeRTOS is a real time operating system that provides a low-memory kernel capable of low-power operation. This minimal kernel is expanded upon with libraries that can be appended and configured based on application needs. This project provides a stable and widespread ecosystem with support for over 40 microcontroller architectures and long term support releases. FreeRTOS provides implementations of memory protection units on equipped microcontrollers. However, this must be implemented and configured on a per application basis and does not easily provide the granularity required for isolating individual processes. Subsequently, FreeRTOS does not provide a by default kernel-process or process-to-process isolation. [11].

Networking Overview

Given FreeRTOS's modular library design, the networking stacks are implemented outside the kernel as libraries. These libraries consist of capabilities ranging from UDP, TCP, BLE, WiFi, Cellular LTE-M, and ethernet (FreeRTOS-Plus-TCP, Amazon Web Services (AWS) IoT Greengrass, AWS IoT Core) [12, 13]. Of these, the TCP and UDP networking libraries are built upon a generic UDP/TCP POSIX socket interface. Furthermore, FreeRTOS provides packaged networking solutions providing management and updating capabilities through AWS IoT Over-the-air-updating and AWS IoT Jobs [12].

Thread Specific Overview

FreeRTOS has been used in conjunction with OpenThread in the OpenThread RTOS project. OpenThread RTOS "provides both system-level and application support for connecting a device to a Thread network and the internet" [14]. This provides an integration of FreeRTOS, LwIP, and OpenThread into a single software platform [14]. Despite, OpenThread RTOS being officially supported by OpenThread, the project seems to have less activity recently based on the

repository's most recent commit occurring in January of 2023 [14, 15]. Beyond OpenThread RTOS, FreeRTOS is independently also used in numerous Thread applications. Per Thread's device manufacturer page, Texas Instruments utilizes FreeRTOS as the embedded operating system for a number of chipsets providing Thread support. Furthermore, Bouffalo Lab and Espressif implement a FreeRTOS based Thread border router [16].

1.2.2 RIOT

General Overview

Riot is an embedded operating system purpose built for IoT applications and supports 32, 16, and 8 bit microcontroller architectures. Major features of this operating system include: a micro-kernel, tickless scheduling, soft real time capabilities, and low overhead multithreading [17, 18]. The RIOT kernel serves to provide functionality for multi-threading and separates other necessary functionality such as device drivers, networking stacks, and applications into separate libraries [18].

Networking Overview

The RIOT networking stack implements a modular design exposing two primary programming interfaces: *netdev* and *sock*. Respectively, *netdev* provides a generalized driver interface for the underlying radio hardware while *sock* serves a purpose similar to POSIX sockets. Notably, *sock* interfaces exclusively use static memory and provide a generalized API that is agnostic to the underlying networking stack [18]. Currently, RIOT provides networking support for application, link, and transport protocols including: IEEE 802.15.4, IPv4, IPv6, LoRa, UDP, TCP, BLE, 6LoWPAN, and WiFi [17].

Thread Specific Overview

RIOT provides a port of OpenThread that is currently deemed stable [17]. This port to the RIOT platform has not been officially certified by the Thread group, but is acknowledged by OpenThread [19].

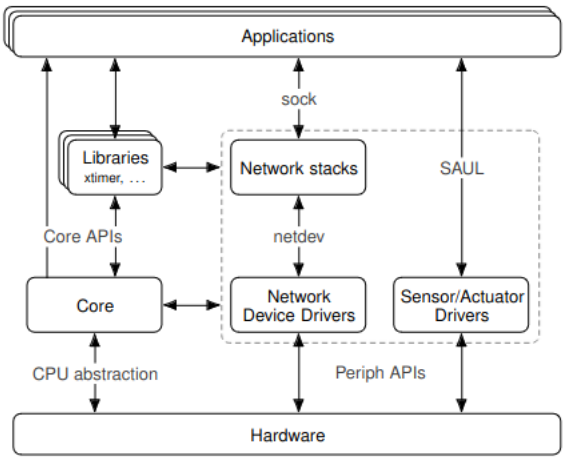

Figure 1.6. Depiction of RIOT programming APIs and networking stack integration—**Figure From** [18].

1.2.3 Contiki-NG

General Overview

Contiki-NG is an embedded operating system purpose built for "severely constrained wireless embedded devices" [20]. Contiki-NG originated as a fork from the Contiki operating system with the aim of robust IPv6 communication and supporting modern 32 bit IoT MCU platforms such as ARM Cortex M3 [21]. Contiki-NG implements a process abstraction using the Contiki-NG *protothread* construct. Process *protothreads* are scheduled cooperatively and non-preemptively through an event-driven model [21].

Networking Overview

Contiki-NG describes the projects networking as an "RFC-compliant, low-power IPv6 communication stack, enabling Internet connectivity" [21]. Because networking support is a primary goal of Contiki-NG, the networking stack provides support for a variety of networking protocols as depicted in Figure 1.7 [20].

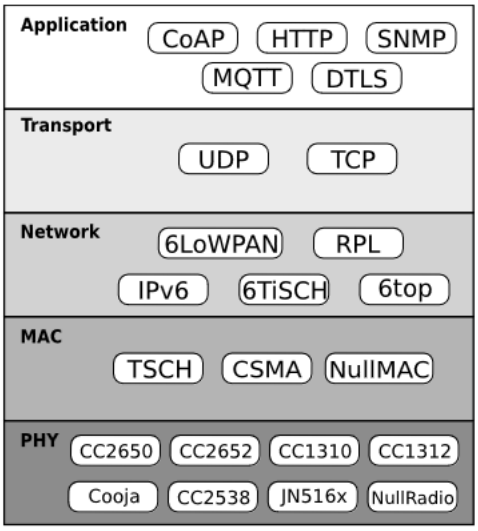

Figure 1.7. Depiction of Contiki-NG network stack—-**Figure From** [20].

Thread Specific Overview

Although Contiki-NG possesses the IEEE 802.15.4 / 6LoWPAN / IPV6 / UDP building blocks needed for a Thread network, Contiki-NG does not possess a Thread implementation.

1.2.4 Zephyr

General Overview

Zephyr is a Linux foundation real time operating system with an emphasis on portability, security, and connectivity. This RTOS incorporates a lightweight kernel with support for over 450 boards and powers many real world embedded applications ranging from wearables, industrial IoT, automotive, and healthcare [22]. Furthermore, Zephyr is capable of running on devices as small as 8kB and also provides thread-level memory protection [23]. Zephyr's architecture is depicted in Figure 1.8.

Networking Overview

Zephyr has expansive networking support given the project's focus on connectivity. This includes support for the following technologies: IEEE 802.15.4, BLE, CAN, Cellular, Ethernet,

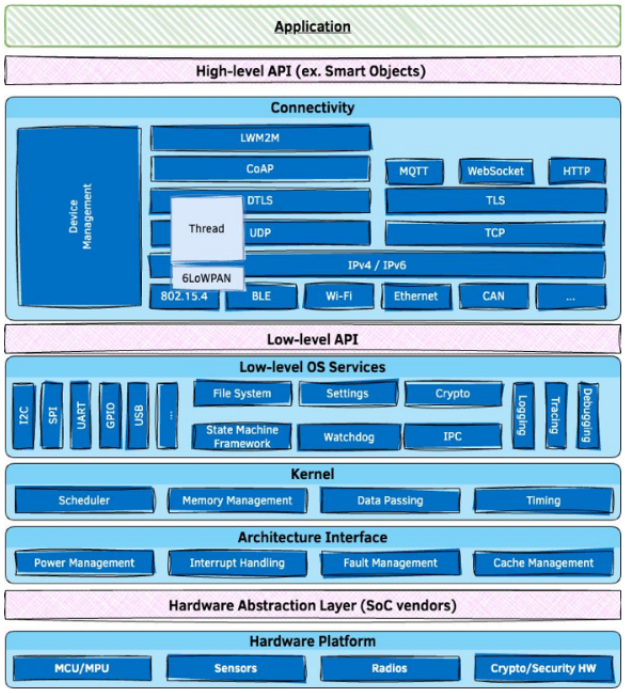

Figure 1.8. Zephyr architecture diagram—**Figure From** [22].

LoRaWAN, Thread, USB, and WiFi. Furthermore, Zephyr's networking stack also supports many standards such as 6LoWPAN, CoAP, HTTP, IPv4, IPv6, and MQTT [23].

Thread Specific Overview

Zephyr provides a robust and actively maintained integration for OpenThread [23]. This OpenThread integration is officially acknowledged by OpenThread [24].

1.2.5 Tock

General Overview

Tock is an embedded operating system, written in Rust, that utilizes Rust's memory safety and the microcontroller memory protection unit to provide strict kernel process isolation, userspace fault tolerance, dynamic memory, and support for multi-tenant applications. Tock is purpose built for low-memory resource constrained devices [25]. To date, Tock has primarily been embraced in security and secure root-of-trust applications due to its strict security guarantees

[26].

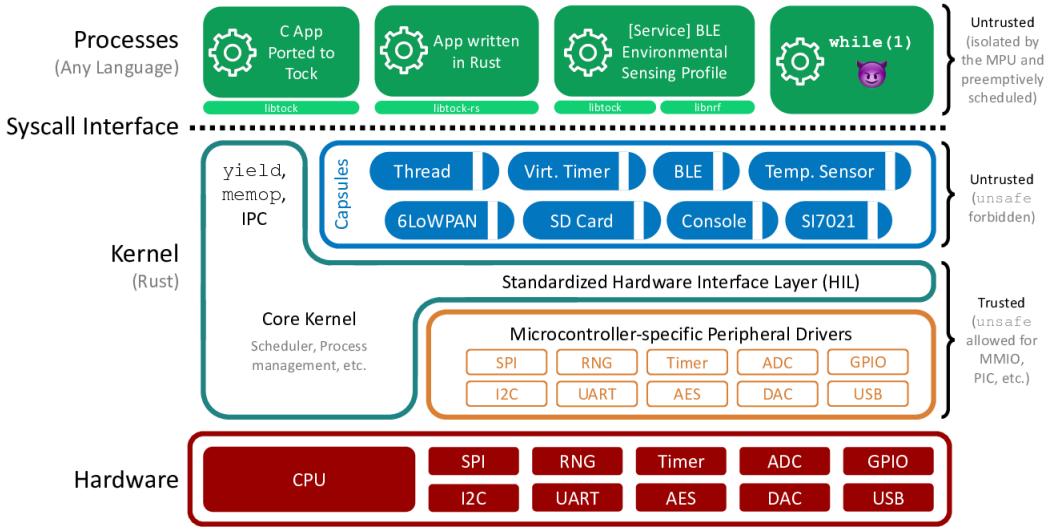

Figure 1.9. Tock architecture diagram demonstrating Tock's untrusted application and capsule design—**Figure From** [26].

Networking Overview

Although Tock possess a networking stack, the networking stack remains primarily in development and has limited industry adoption. Currently, Tock possesses a UDP / IPv6 / 6LoWPAN / IEEE 802.15.4 network stack in addition to the capability of sending BLE advertisements and ethernet packets [26].

Thread Specific Overview

Because Tock possesses a UDP / IPv6 / 6LoWPAN / IEEE 802.15.4 network stack, it possess all necessary infrastructure to support Thread. However, Tock, prior to this project did not have Thread support. Tock now supports a limited subset of Thread capabilities to attach as a child to a Thread router device.

Chapter 2

Networking Interface

2.1 Challenges

Designing a Thread networking interface faces two primary challenges: how to best incorporate Thread into existing network stacks and whether to place Thread specific implementation details in the kernel or a user process.

2.1.1 Thread / Network Stack Interface

Standard UDP / IPv6 / 6LoWPAN / IEEE 802.15.4 network stacks (Figure 2.1) allow for clean layer isolation. Each layer possesses a clear role and domain of control: the UDP layer accepts payload data, the IPv6 layer provides packet routing, the 6LoWPAN layer performs compression, and the 15.4 layer handles encryption and radio control. This modularity improves the implementation's readability and lends well to limiting a given layer's permissions to what is strictly necessary. For instance, only the 15.4 layer needs permission to access and alter the physical radio's state.

Thread is built upon and utilizes the aforementioned network stack. However, Thread does not act as a "user application" that interacts exclusively with the UDP layer, but instead exerts control over multiple layers. This tangibly occurs as Thread "hops" through 15.4 channels and must change the state of the physical radio. Thus, Thread potentially violates the clean isolations typically present in a UDP / IPv6 / 6LoWPAN / IEEE 802.15.4 network stack. Furthermore,

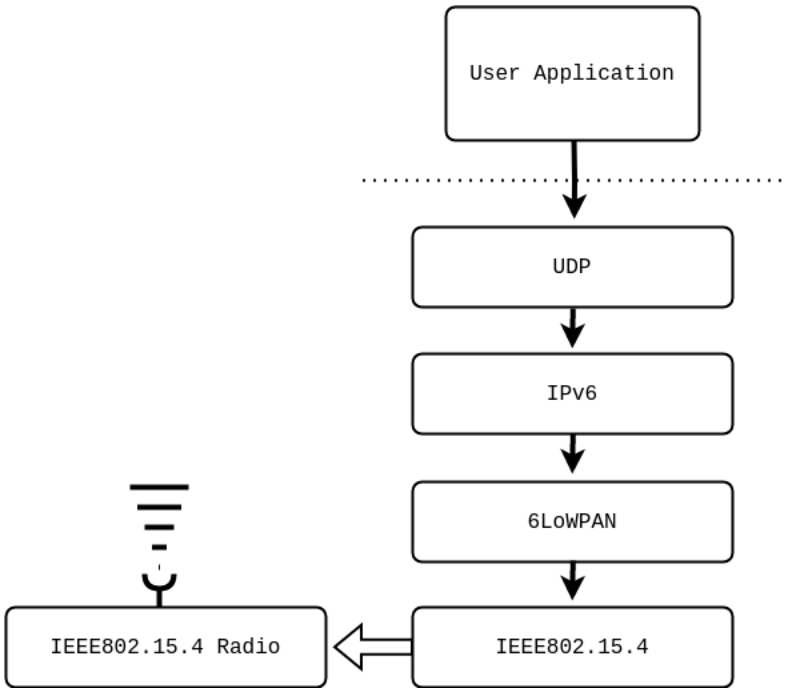

Figure 2.1. Standard UDP / IPv6 / 6LoWPAN / IEEE 802.15.4 Transmission Network Stack. Each layer appends necessary headers and passes the packet downwards until the physical radio transmits the packet. The receiving case occurs in the opposite direction.

alterations to the network stack to support Thread may lead to issues implementing a standard UDP application in addition to Thread. This concern is particularly pronounced in a multi-tenant embedded operating system; a given application may wish to send standard UDP packets using the 15.4 radio while a separate application may have initiated a Thread network. In such an instance, the kernel would need to either appropriately share resources or inform either application that the given request failed. This implies that a Thread interface can not be added to the network stack by merely altering the implementation of each layer. Rather, Thread necessitates a principled design that acknowledges and handles nuances exposed in a multi-tenant operating system.

2.1.2 Kernel / Process

A Thread implementation can either be placed in the kernel or as an application process. The benefits of each are respectively the challenges of the counterpart. Namely, an application

process trades performance for enhanced and simplified isolation while a kernel implementation removes overhead while introducing greater security risks.

To access needed radio and cryptographic resources, a Thread process would need to perform a system call and context switch to enter the kernel. Such a context switch and system call incurs computational overhead and may lead to latency. In a timing sensitive application, such as Thread networking, this latency may result in unreliable performance. Despite this overhead, an application process benefits from the strict isolation enforced by an operating system possessing kernel-process isolation; Thread implementation errors will not cause errors in other processes.

A kernel based Thread implementation removes substantial overhead related to context switching, interprocess communication (IPC), and scheduling. These gains are obtained through the kernel code executing in a privileged state. This subsequently implies, however, that a Thread related implementation error could result in the entire system being compromised. Moreover, using a third-party Thread implementation would require importing external dependencies into the kernel. Such external dependencies linked to core kernel code could expose the system to supply chain threats and expose the kernel to a greater attack surface.

2.2 Design Goals

Primary Goal: Thread interface design that does not alter the existing networking interface but instead exerts *petitioned* control over each layer.

1. **Maintain network layer divisions / separations**
2. **Provide functionality and control to satisfy all Thread requirements**
    - IEEE 802.15.4 send / receive / acknowledgement
    - AES-128 CCM encryption
3. **Hardware agnostic**: Meet goal two across non-homogenous hardware platforms.

4. **Meet timing requirements**: Thread and underlying protocols require that messages be acknowledged and/or responded to within a certain time bound.

5. **Provide multi-tenancy for Thread and UDP application**

2.3 Proposed Design

We propose a design in which a Thread implementation does not alter the existing UDP / IPv6 / 6LoWPAN / IEEE 802.15.4 stack but instead serves as a multi-layer client. Elaborating further, Thread will register as a client to the UDP and 15.4 mux/virtualizer respectively. Once registered, Thread will then utilize the existing UDP and 15.4 function call API. This design satisfies multi-tenancy requirements through the interface with the mux/virtualizer; Thread requests will be queued and selected for transmission in the same manner as other packets sent through the network stack. Furthermore, by interfacing with instead of controlling these distinct layers, Thread can only *petition* the 15.4 layer for a state change. This ensures each layer can implement a respective policy to determine how and when Thread, or any other client, should alter radio state.

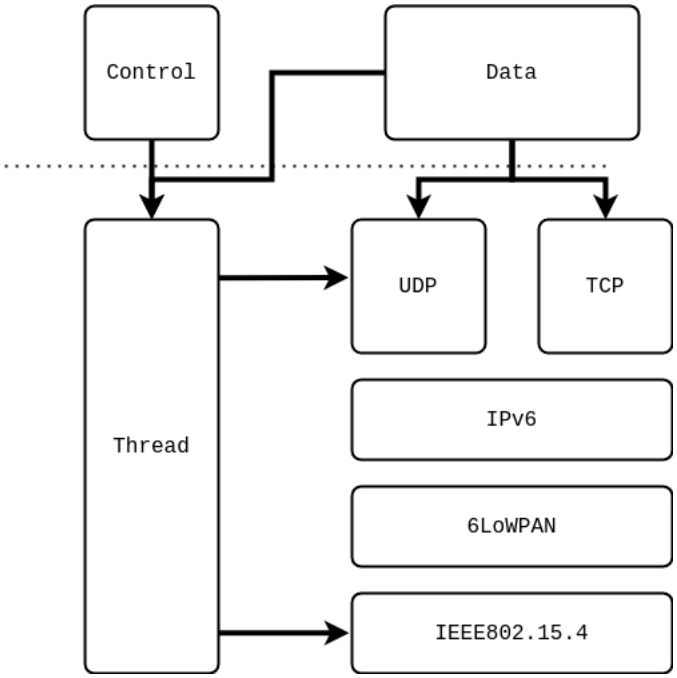

Figure 2.2. Proposed Thread Networking Design. The kernel / process boundary is shown by the dotted line. The control and data boxes in userspace depict two types of requests related to Thread (control requests update network state, data requests are the sending of data).

Chapter 3

Implementation and Evaluation

To implement and explore the proposed design, TockOS was selected as an embedded operating system to implement Thread networking support [25]. TockOS served as an ideal candidate for this exploration as it previously did not support Thread, but in theory possessed the necessary building blocks. Namely, Tock provided support for a UDP / IPv6 / 6LoWPAN / IEEE 802.15.4 stack and supported AES128-CCM encryption on the nRF52840DK development platform [26].

A substantial portion of investigating an optimal Thread interface design centered around first creating a functional Thread implementation. OpenThread was utilized to test the validity of this implementation. More specifically, a successful preliminary Thread implementation would consist of a Tock device attaching as a child to an OpenThread router device. In pursuing this goal, numerous design and Tock specific challenges arose; together these helped to hone what precisely an optimal Thread interface entails and also provide improvements to the open source Tock project. The discussion below will be centered around the challenges encountered and solutions found. Together, these challenges provide a roadmap to the implementation specific details of a proposed Thread network interface.

3.1 MLE Encryption

For the initial minimal implementation, MLE encryption was intentionally excluded for simplicity. However, this immediately created complications as we observed that OpenThread will ignore all unencrypted MLE messages. Subsequently, this resulted in correctly configuring the encryption specified for Thread MLE.

Thread's encryption scheme depends upon a number of minor details and required immense effort to successfully encrypt and decrypt MLE messages. Thread reuses 15.4's AES128-CCM encryption scheme. At a high level, AES-128 CCM requires a 16 byte encryption key and an input message divided into authentication and message data. The authentication data is used to generate a message integrity code (MIC). The authentication data is defined as the concatenation of the source IPv6 address, destination IPv6 address, and auxillary security header. The message data for a MLE message is then the MLE command field followed by all included Type-Length-Values (TLVs). To encrypt this message, the MLE encryption key is utilized. This is generated using Thread's described hashing function in conjunction with the Thread network key [10].

Thread also utilizes link layer encryption. Link layer encryption is accomplished within the 15.4 layer using an AES-128 CCM encryption scheme using the MAC encryption key. This MAC encryption key is generated using a Thread hashing protocol involving the network key [10]. Although Tock implemented a 15.4 link layer encryption, the implementation did not initially work properly and required a minor refactoring. MLE and link layer encryption are depicted within the context of the entire networking stack in Figure 3.1.

Once successfully encrypting and decrypting individual packets, the Thread interface implementation encountered another encryption related complication. Tock, being written in Rust, incorporates the "Rustism" of ownership; this in turn results in strict enforcement of how and when a data structure can be modified. Tock's cryptographic functionality subsequently accepts a buffer that is to be encrypted / decrypted and assumes ownership of the provided buffer.

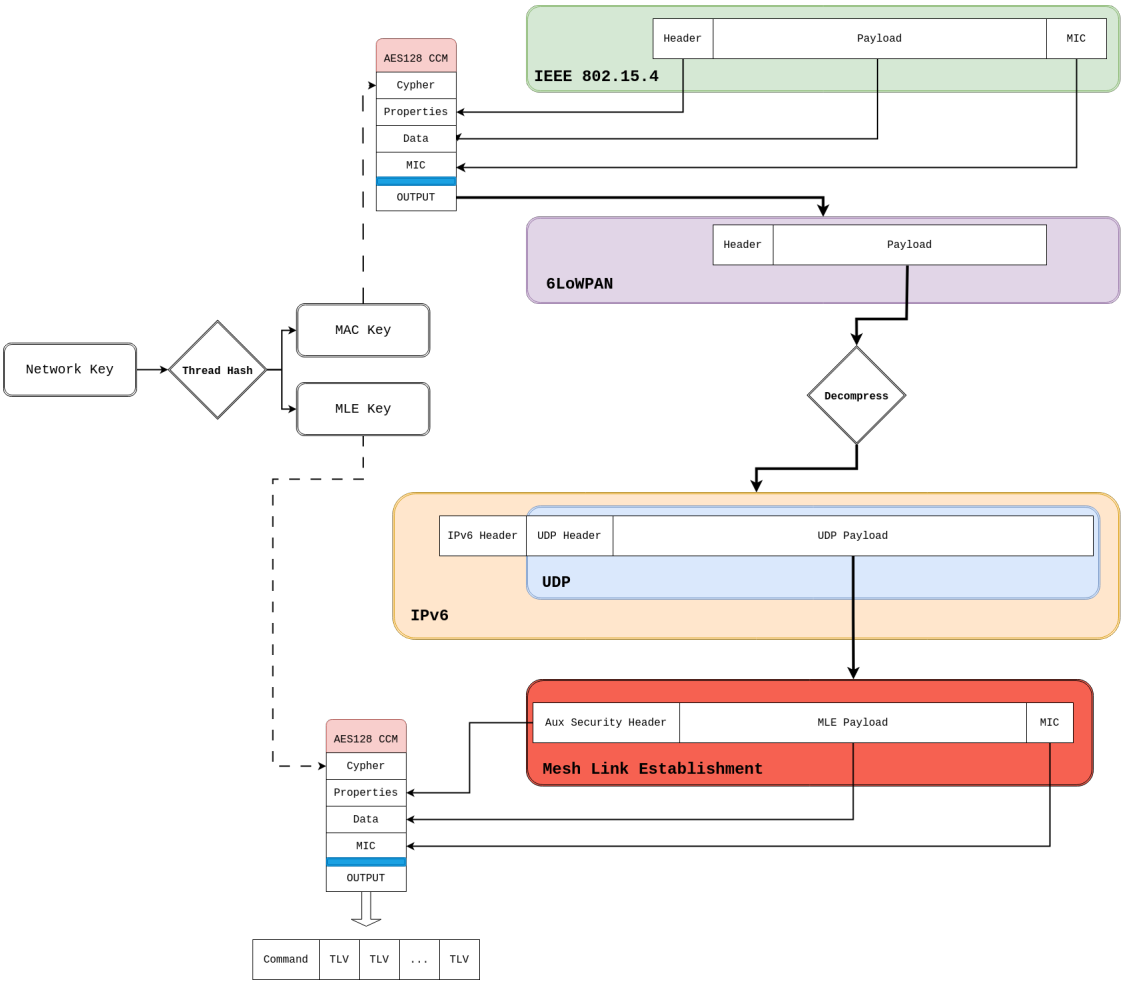

Figure 3.1. Network Stack Framing and Encryption for received IEEE 802.15.4 packet. The upper AES128 CCM decryption represents link layer encryption while the lower AES128 CCM represents MLE encryption.

The cryptographic operation is asynchronous and executes a registered callback function when completed. The originally provided buffer is passed as an argument to this callback, containing the now encrypted or decrypted data. This provides a mechanism to return ownership of this buffer to the original entity that previously held this buffer (i.e. send or receive buffer for the Thread interface). This process is depicted in Figure 3.2.

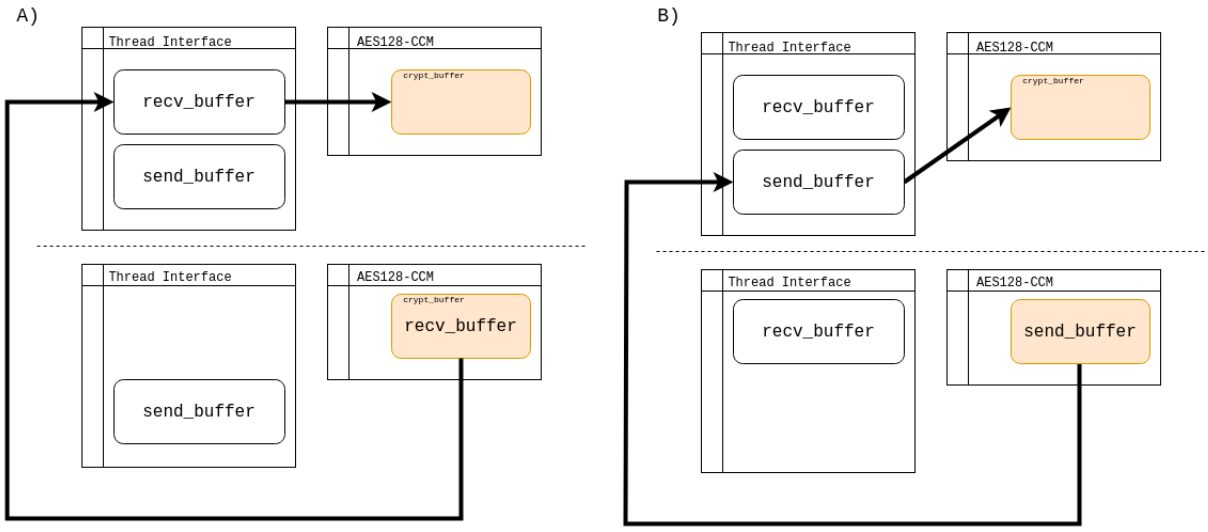

Figure 3.2. Thread interface interaction and ownership model with Tock cryptographic library. The respective buffer that is to be encrypted or decrypted is passed to the AES128-CCM library through a function which obtains ownership of the buffer. The dotted line signifies a later time point at which the cryptographic operation is completed and a callback function returns ownership of the respective buffer to the Thread Interface. **A** and **B** each depict operation for the send / receive buffer.

3.2 Acknowledgments

IEEE 802.15.4 incorporates acknowledgments in which a transmitted packet can specify if the receiver should respond and acknowledge the receipt of a packet. This is determined using an Acknowledge Request bit flag in the 15.4 header frame control field. If a received packet requests to be acknowledged, the receiver must send a response acknowledgement (ACK) within the IEEE 802.15.4 standard's prescribed time window, t, such that: $192\mu s < t < 512\mu s$ [9].

Many 15.4 radios implement ACK generation in hardware (auto-ACK). However, the nrf52840 radio used for developing and testing a Tock Thread network did not possess auto-ACK capabilities. The lack of acknowledgements for the nrf52840 hardware platform created numerous Thread related complications as 15.4 prescribes resending unacknowledged packets. In practice, this resulted in Thread repeatedly sending parent requests and repeatedly timing out the requests as the sent requests were not acknowledged. The state machine for how Thread sends a parent request when attaching a child is depicted in Figure 3.3.

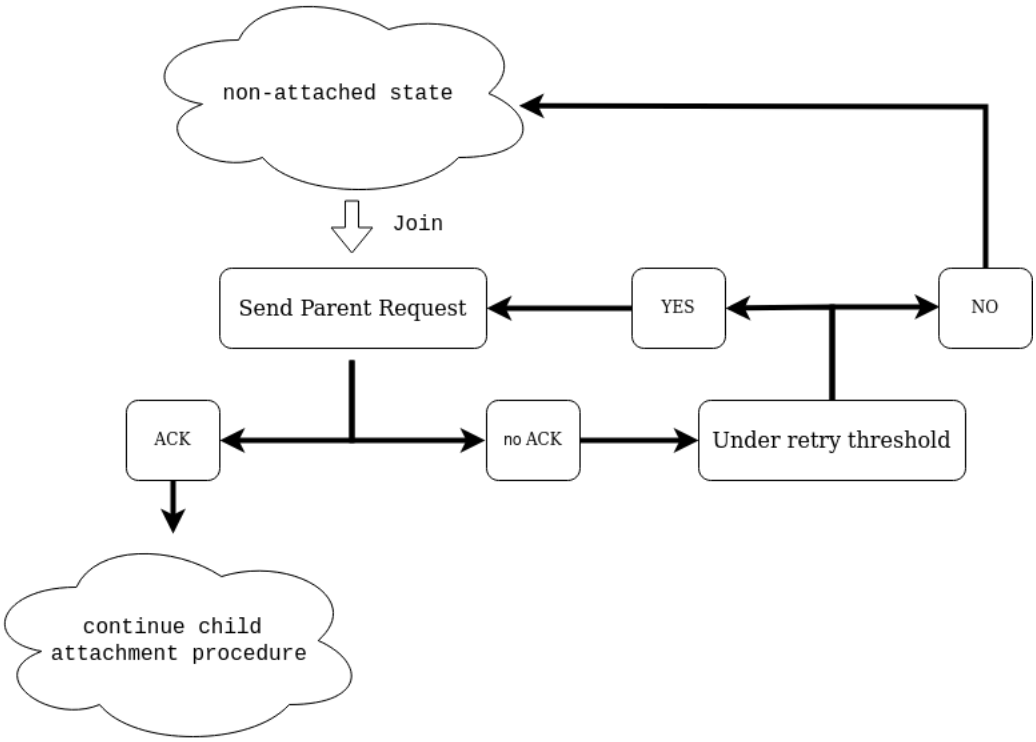

Figure 3.3. Visualization of Thread state machine for sending parent request.

Because not sending acknowledgements resulted in failed parent requests, implementing ACK support for the nrf52840 radio was a requirement for Tock to support Thread. Given the nrf52840's lack of hardware auto-ACK, adding ACK support to the nrf52840 required the implementation of a software auto-ACK. Tock utilizes a generic hardware abstraction such that the radio object the kernel interacts with appears uniform despite underlying hardware inhomogeneities. Subsequently, any software auto-ACK must be implemented in a radio specific driver (as other radios support hardware auto-ACK).

The narrow IEEE 802.15.4 ACK timing window created immediate implementation challenges to transmit an ACK within the 512µs upper bound. A software implementation must complete the following steps in 512µs: receive the packet, parse the input (to check if the ACK bit flag is set), transition the radio from receive to transmit mode, and then send an ACK packet. Accomplishing this timing required a major optimization of the nrf52840 radio driver, but ultimately resulted in meeting the timing requirement for sending ACKs. This was primarily

achieved by waiting to pass the received packet to the networking stack until after the ACK was transmitted and optimizing the radio's state machine.

To fit into Tock's execution model, the auto-ACK was implemented in the bottom half interrupt handler. This creates the obvious complication that an unrelated interrupt may assume control and block, leading to the auto-ACK missing the time bound. However, this potential failure mode was deemed acceptable as 15.4 is inherently designed to handle failed transmission (i.e. not receiving ACK). Furthermore, this case has not presented complications across extensive testing of joining a Thread network and sending 15.4 packets.

3.3 Thread Capsule

Implementing MLE encryption, link layer encryption, and ACKs provides all necessary components for implementing a Thread capsule. The Thread capsule was generally structured to possess a receiving buffer, sending buffer, a reference to the cryptographic tooling, and an interface to the UDPMux. This allows a Thread capsule to register as both a receiver and client and encrypt the necessary packets. Notably, this capsule implementation does not provide control over 15.4 properties such as the radio channel. This is a drawback of the current implementation.

The current Thread capsule is capable of successfully attaching a Tock child device to a Thread router. This is coordinated and controlled through a Thread state machine depicted in Figure 3.4. The Thread state machine works in conjunction with alarms to time out stale requests and prevent race conditions. As a whole, the Thread capsule generates, encrypts, and sends a parent request, followed by handling a receive callback from the UDPMux receiver. The Thread capsule driver currently exposes only one system call command to userspace: initiating a Thread parent request. [1]

[1]The entirety of the Tock Thread capsule has been merged into upstream Tock and can be viewed in the Tock github repository (https://github.com/tock/tock). Within the repository, the source code for this capsule is found within /capsules/extra/src/net/thread.

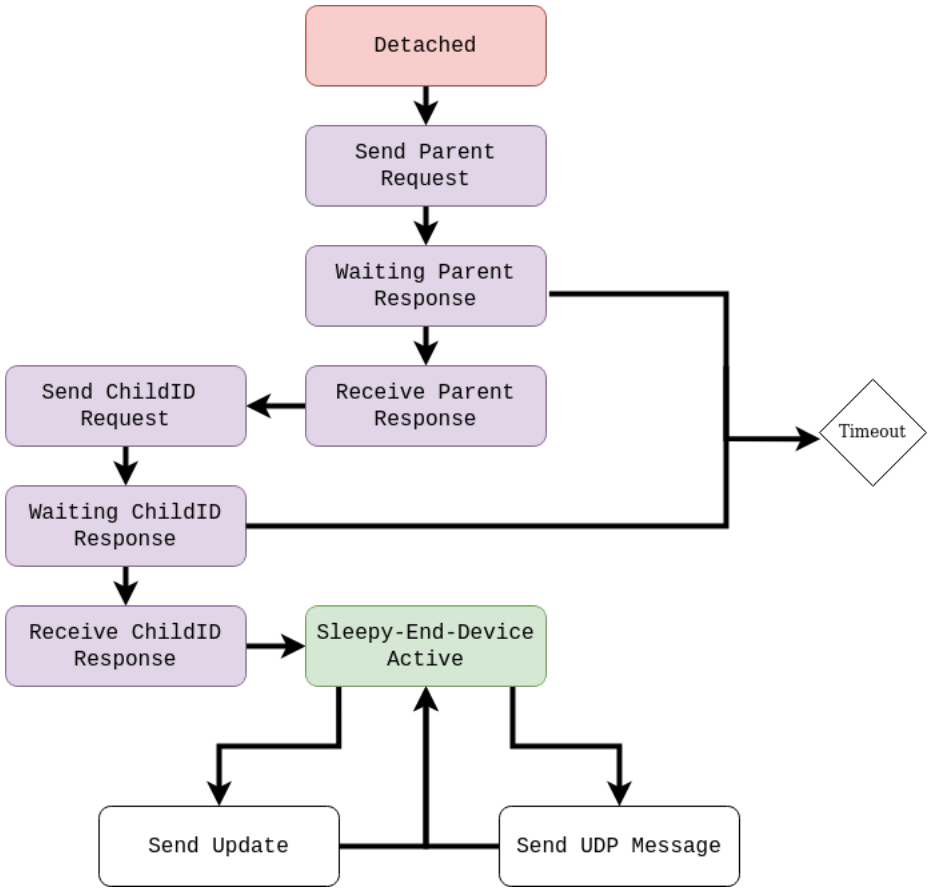

Figure 3.4. Visualization of the Thread state machine implemented in the Thread capsule.

3.4 Evaluation

The Tock Thread network implementation was evaluated against an OpenThread network. This testing served to confirm interoperability between the Tock Thread device and a production-grade, certified Thread device. To confirm this interoperability, we attempted to attach the Tock device to an OpenThread router and evaluate if the router device recognized the Tock device as an attached child. This was first measured using Wireshark and a dongle to sniff Thread packets. Through this, we observed a child attachment between the Tock child and OpenThread router that matched the behavior described in the Thread specification. The Wireshark captured traffic is displayed in Figure 3.5.

Although observing transmitted packets in joining the Thread network provides proof of

Source	Destination	Protocol	Length	Info
fe80::a0b5:a691:ee42:5636	ff02::2	MLE	61	Parent Request
		IEEE 802.15.4	3	Ack
fe80::a4c1:d479:21cf:8c6	fe80::a0b5:a691:ee42:5636	MLE	111	Parent Response
		IEEE 802.15.4	3	Ack
fe80::a0b5:a691:ee42:5636	fe80::a4c1:d479:21cf:8c6	MLE	84	Child ID Request
		IEEE 802.15.4	3	Ack
a6:c1:d4:79:21:cf:08:c6	a2:b5:a6:91:ee:42:56:36	6LoWPAN	124	Data, Dst: a2:b5:
		IEEE 802.15.4	3	Ack
fe80::a4c1:d479:21cf:8c6	fe80::a0b5:a691:ee42:5636	MLE	122	Child ID Response
		IEEE 802.15.4	3	Ack

Figure 3.5. Wireshark capture depicting Thread packets exchanged between Tock child and OpenThread router. The Tock child has the address *fe80::a0b5:a691:ee42:5636*. The OpenThread router has the address *fe80::a4c1:d479:21cf:8c6*.

interoperability, we can better evaluate the success of the child attaching to the router through the router's child table. OpenThread provides a command line interface for interacting with a device's OpenThread network. As part of the command line interface, there exists a command to display all children connected to the router. The OpenThread's child table is shown in Figure 3.6. In this child table, we observe that the Tock child device has successfully attached to the OpenThread router.

```
> child table
| ID  | RLOC16 | Timeout    | Age        | LQ In | C_VN |R|D|N|Ver|CSL|QMsgCnt|Suprvsn| Extended MAC     |
+-----+--------+------------+------------+-------+------+-+-+-+---+---+-------+-------+------------------+

Done
> child table
| ID  | RLOC16 | Timeout    | Age        | LQ In | C_VN |R|D|N|Ver|CSL|QMsgCnt|Suprvsn| Extended MAC     |
+-----+--------+------------+------------+-------+------+-+-+-+---+---+-------+-------+------------------+
|   1 | 0xbc01 |        240 |          7 |     3 |   52 |1|1|1|  4| 0 |     0 |   129 | a2b5a691ee425636 |
```

Figure 3.6. OpenThread Router child table depicting the empty child table prior to the Tock device attaching and the populated child table after the Tock child device's attachment.

Together, Figures 3.5 and 3.6 show the Tock Thread network's ability to attach as a child to the OpenThread router. These results demonstrate Tock's Thread compliance to attach to a parent device; subsequently we demonstrate that Tock possesses Thread interoperability for child-parent attachment.

Chapter 4

Future Work and Conclusions

From this work, Tock now supports a minimal Thread implementation to successfully attach a Tock child device to an OpenThread router. Furthermore, this work provides an interface design to maintain network layer abstractions while simultaneously providing the needed control. Using these achievements and gleaned insights, two primary paths for future work are detailed below.

4.1 Permission Model

A multi-tenant network stack introduces numerous permission challenges as varied applications may wish to configure the radio differently. The current Thread implementation does not account for these concerns. Furthermore, Tock currently ignores radio channel permission concerns and does not expose this functionality.

Future work should investigate how best to allocate resources that are inherently unable to be shared (i.e. the channel the radio uses). Tock provides the *capability* mechanism for controlling elevated permissions. Tock *capabilities* are a unique data structure that are enforced using type checking and prevented from being created by using the Rust *unsafe* label. Subsequently, *capabilities* can only be created and assigned within kernel compilation. Although *capabilities* provide an ideal mechanism for handling kernel / capsule elevated permissions, future investigation will be required for distinguishing and allocating elevated permission to user

applications.

4.2 OpenThread User Process

The described Tock Thread implementation was completed using a kernel capsule. This was a simpler initial implementation as it removed the risk of timing latency due to system calls. This timing concern is particularly pronounced in light of Thread's strict timing requirements and Tock's design fundamentally not providing timing guarantees. Furthermore, a user process Thread implementation would require interprocess communication to allow multiple processes access to the Thread network, adding yet another layer of complexity. In total, these complexities justified an initial proof of concept Thread implementation and network interface existing in the kernel.

Despite the above concerns and challenges, there exists ample benefits for a user process Thread implementation; namely the ability to use OpenThread. Tock, emphasizing security, disallows kernel imports of external dependencies. In turn, this implies that any kernel Thread implementation would require a bespoke implementation of Thread based upon the the Thread standard. Fully completing such an implementation for the more complicated router node type would constitute a herculean effort. This effort, however, can be potentially avoided through porting OpenThread to Tock as a user process (user processes are entirely untrusted in Tock and subsequently fully allow external dependencies). Regardless of this security policy, the Tock kernel is implemented in Rust while OpenThread is implemented in C. An OpenThread user process also avoids the complication of translating OpenThread to Rust as Tock supports C applications.

The steps required to port OpenThread to a new platform are as follows [27]:

- IEEE 802.15.4-2006 2.4 GHz radio
  - Send and receive IEEE 802.15.4 frames

  - Generate IEEE 802.15.4 Acknowledgment frames
  - Provide Received Signal Strength Indicator (RSSI) measurements on received frames
- A millisecond-resolution free-running timer with alarm
- Non-volatile storage for storing network configuration settings
- A true random number generator (TRNG)

The current kernel Thread implementation demonstrates the viability of Tock's 15.4 capabilities. The three remaining requirements would necessitate additional testing and development, but are achievable. Providing Tock with an OpenThread port would provide a unique combination of Tock's security guarantees in addition to the production grade Thread stack OpenThread implements.

4.3 Conclusions

Through this work, Tock now offers limited Thread support. Implementing and interfacing Thread to Tock's existing network stack provided insights that an "alongside" and "petitioned control" Thread interface functions well for integrating Thread into an existing network stack. However, this conclusion must be caveated for network stacks under load and in resource demanding applications as this remains untested.

Beyond these primary conclusions, this work demonstrated that a software platform's presence of building blocks does not necessarily equate with these building blocks being usable. For instance, Tock in theory appeared to support IEEE 802.15.4 acknowledgments and link layer encryption. In practice, however, these features in Tock were not usable "as is" for a robust Thread application. This discrepancy demonstrates the need for software platform studies to assess the maturity of components rather than simply investigating for the presence of components.

Bibliography

[1] Ivan Djordjevic, William Ryan, and Bane Vasic. *Coding for Optical Channels*. Springer, 2010.

[2] What Is Thread. https://www.threadgroup.org/What-is-Thread/Thread-Benefits. Accessed: 2023-11-13.

[3] Thread is Matter's secret sauce for a better smart home. https://www.theverge.com/23165855/thread-smart-home-protocol-matter-apple-google-interview. Accessed: 2023-11-17.

[4] Here's What the 'Matter' Smart Home Standard Is All About. https://www.wired.com/story/what-is-matter/. Accessed: 2023-11-13.

[5] OpenThread - Node Roles and Types. https://openthread.io/guides/thread-primer/node-roles-and-types. Accessed: 2023-11-13.

[6] OpenThread - Network Discovery and Formation. https://openthread.io/guides/thread-primer/network-discovery. Accessed: 2023-11-13.

[7] Jonathan W. Hui and David E. Culler. Ip is dead, long live ip for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, page 15–28, New York, NY, USA, 2008. Association for Computing Machinery.

[8] OpenThread - IPv6 Addressing. https://openthread.io/guides/thread-primer/ipv6-addressing. Accessed: 2023-11-13.

[9] Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pages 1–800, 2020.

[10] Thread 1.3.0 Specification. https://www.threadgroup.org/ThreadSpec. Accessed: 2023-04-11.

[11] FreeRTOS. https://www.freertos.org/. Accessed: 2023-11-13.

[12] FreeRTOS Features. https://www.freertos.org/. Accessed: 2023-11-07.

[13] FreeRTOS Features. https://aws.amazon.com/freertos/features/. Accessed: 2023-11-08.

[14] OpenThread RTOS. https://openthread.io/platforms/rtos/ot-rtos. Accessed: 2023-11-07.

[15] ot-rtos. https://github.com/openthread/ot-rtos. Accessed: 2023-11-07.

[16] Thread Developers. https://www.threadgroup.org/What-is-Thread/Developers. Accessed: 2023-11-07.

[17] RIOT. https://www.riot-os.org/. Accessed: 2023-11-08.

[18] Emmanuel Baccelli, Cenk, Gündoğan, Oliver Hahm, Peter Kietzmann, Martine Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, 5(6), 2018.

[19] OpenThread - RIOT. https://openthread.io/platforms/rtos/riot. Accessed: 2023-11-08.

[20] George Oikonomou, Simon Duquennoy, Atis Elsts, Joakim Eriksson, Yasuyuki Tanaka, and Nicolas Tsiftes. The Contiki-NG open source operating system for next generation IoT devices. *SoftwareX*, 18, 2022.

[21] Contiki-ng Repository. https://github.com/contiki-ng/. Accessed: 2023-11-07.

[22] Zephyr Project. https://www.zephyrproject.org. Accessed: 2023-11-15.

[23] Zephyr Datasheet. https://www.zephyrproject.org/wp-content/uploads/sites/38/2023/11/Zephyr_Datasheet_2023_110123-1.pdf. Accessed: 2023-11-15.

[24] OpenThread - Zephyr. https://openthread.io/platforms/rtos/zephyr. Accessed: 2023-11-15.

[25] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 234–251, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Tock Repository. https://github.com/tock/tock. Accessed: 2023-11-13.

[27] OpenThread - Porting Guide. https://openthread.io/guides/porting. Accessed: 2023-11-13.