NDN, Technical Report NDN-0043. http://named-data.net/techreports.html
Revision 2: March 29, 2017
Revision 1: July 16, 2016

The Design and Implementation of the NDN Protocol Stack for RIOT-OS

Wentao Shang, Yang Wen, Alex Afanasyev, and Lixia Zhang
Computer Science Department, UCLA
Email: {wentao,aa,lixia}@cs.ucla.edu, yangwenca@gmail.com

***Abstract*—The Named Data Networking (NDN) architecture has been proposed as a promising solution for supporting communications in IoT environments. An important class of IoT platform is the constrained devices that have limited computing resources and are connected by constrained networks. This paper presents the design and implementation of the NDN protocol stack for RIOT-OS, a popular operating system for constrained IoT platforms. We succeeded in integrating the core NDN packet forwarding logic into the RIOT-OS kernel together with a high-level application interface with data security support. Our results demonstrated the feasibility of using NDN protocol stack to support applications on constrained devices with only 10s of KB of RAM and flash memory.**

I. Introduction

Recent years have witnessed a rapid development of the Internet-of-Things (IoT) technologies, and the interest in this area continues to grow from both the industry and the academia. An IoT system may interconnect a large number of sensors and actuators and allow users to access various types of data and control functions remotely. Named Data Networking (NDN) [1], [2] has been proposed as a promising solution to support IoT semantics at the networking level [3], [4], simplifying operations over heterogeneous networks, making massive amounts of sensor data readily available to the applications, and enabling data-centric security for sensing data and actuation commands.

An important class of IoT devices is the constrained devices with limited memory, low processing capability, and low-power network interfaces. These features make IoT technologies available at low cost and work with battery power for years, but also raise the question of how well the NDN protocol stack may fit into such constrained environments. There is a perception that NDN's forwarding logic is too complex to be implemented on constrained devices, or that the data-centric security mechanism too expensive for constrained IoT applications.

This paper addresses the above issues by presenting the design and implementation of the NDN protocol stack on RIOT-OS [5], a popular operating system designed for constrained IoT platforms. RIOT-OS provides a uniform abstraction of the hardware details across multiple constrained platforms with a developer-friendly programming interface. It already supports a number of popular IoT development boards with the built-in drivers for many peripheral IoT sensors, including humidity and temperature sensor, light sensor, gas meter, accelerometer, etc. [6], that can be used to implement a lot of real-life applications.

We dubbed our implementation *NDN-RIOT*, which integrates the core NDN forwarding logic into the RIOT-OS kernel and supports NDN communication directly over IEEE 802.15.4 low-rate wireless network interface that is widely available on RIOT-OS-capable devices, as well as the traditional Ethernet. While having limitations imposed by the constrained environment, NDN-RIOT still provides the essential for IoT data-centric security feature of data authentication through a limited set of signature types (HMAC and ECDSA). Our demo application can run on a 32-bit microcontroller with 32 KB of RAM and require only ≈40 KB of flash memory for storing the whole program (including the application code and the OS kernel). We made the NDN-RIOT source code freely available on GitHub [7], and we highly welcome comments, feature requests, and help from the broad community.

The rest of the paper is organized as follows: Section II gives a brief review of the NDN architecture and the RIOT-OS system; Section III describes the software design of NDN-RIOT; Section IV presents evaluation results that focus on memory usage and performance of the system; Section V reviews related works on the network stack implementation for constrained devices; finally, Section VI concludes the paper and addresses future work.

II. Background

A. Named Data Networking

Named Data Networking (NDN) [1], [2] is a realization of the Information-Centric Network (ICN) paradigm that shifts the network communication model from host-centric (in TCP/IP) to data-centric. In NDN, every piece of data has a unique, hierarchically structured name that is used by the applications to identify and retrieve the data. A *consumer* requests desired data by sending an *Interest* packet carrying the data name (or its prefix). The network forwards the Interest packet according to the Interest name and using the *Interest Forwarding Strategy* which takes input from the *Forwarding Information Base (FIB)* about the potential locations of the data and utilizes data-plane performance measurement to adjust forwarding decisions [8]. As an Interest is forwarded, each router along the path keeps track of the interface the Interest came from in its *Pending Interest Table (PIT)*.

Once an Interest finds the requested data, either at the original *producer* or a forwarder's *cache* (called *Content Store* or CS), the *Data* packet is returned to the consumer by reversing the Interest forwarding path, using the trace left in the routers' PIT. Each Data packet carries a cryptographic signature that securely binds the name and content, allowing the consumer to authenticate the data regardless of how and from where it is retrieved. Consequently, an NDN network can utilize various types of data storage (such as in-network cache and dedicated repos) to improve the efficiency of communication.

B. RIOT-OS

RIOT-OS [5] is a cross-platform operating system designed for constrained IoT devices with a minimum of 10s of KB of RAM and flash memory. It provides a clean and uniform abstraction over the details of various IoT hardware and a C (C99-compatible) programming environment for the application developers, with full support for the C standard library. There is also a limited support for C++ and STL library. The RIOT-OS micro-kernel provides core functionalities such as multi-threading, priority-based scheduling, interrupt handling, and Inter-Process Communication (IPC) interface. It currently includes drivers for Ethernet and IEEE 802.15.4 network interfaces, as well as various peripheral IoT sensors and actuators. It also has a built-in support for IoT-related network protocols such as IPv6, UDP, 6LoWPAN, RPL, CoAP, etc. Since its initial release in 2013, RIOT-OS has been ported to several IoT platforms with different CPU architectures, including ARMv7, ARM Cortex-M0+, MSP430, etc., and is quickly gaining popularity in the IoT community.

III. NDN-RIOT Design

A. Objectives

The main objective for NDN-RIOT implementation is to support the core NDN forwarding operations with minimum computing resources and provide compatibility with the current protocol specification [9]. We initially target devices with 10s of KB of RAM (for storing runtime data), 100s of KB of flash memory (for storing binary executable code), and low-power CPU running at a frequency of less than 100 MHz. This requires NDN-RIOT to implement simplified versions of the underlying data structures (PIT, FIB, CS) to fit within the constrained parameters, supporting most of the core requirements for the forwarding mechanisms. The second but not less important objective is to facilitate development of secure NDN-based applications by providing a high-level API with inherent support for data-centric security primitives.

B. Software Architecture

RIOT-OS uses a micro-kernel architecture where the network protocol modules (e.g., IPv6 and UDP) are implemented as kernel threads,[1] and packet passing across layers is achieved through IPC between different modules. The NDN-RIOT stack is implemented in the same fashion, as illustrated in Fig. 1. When the application wants to send an NDN packet, it passes the packet to the NDN thread in an IPC call; the NDN thread processes the packet and then passes it to the network device (NIC) driver thread for transmission. When the NIC driver receives a Layer-2 frame that contains an NDN packet, it passes the packet to the NDN thread, which then notifies the applications and/or further propagates the packet to remote NDN-RIOT nodes.

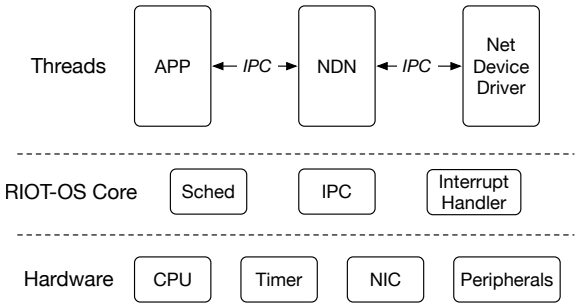

Fig. 1: Software architecture of NDN-RIOT

C. Packet Encoding and Decoding

In order to support constrained devices with extremely limited memory resources, we carefully designed the packet encoding and decoding routines to minimize the number of copies of the NDN packets in memory. An important design decision we made is to store the packet in the wire format form throughout its lifetime in the system and access the underlying elements (e.g., access the name field to decide where to forward the interest or access the signature field during the data packet authentication) through on-demand TLV packet parsing. This way we saved memory and CPU cycles by not using the deserialized packet, paying the prices of additional CPU processing every time a packet field is accessed. Fortunately, as we demonstrate in Section IV, parsing the TLV-formatted NDN packet is usually very efficient.

To simplify memory management and ownership tracking, we implemented a lightweight *shared memory block* structure (inspired by the shared pointer mechanism in C++11) for storing the TLV-encoded NDN name, Interest, and Data. It allows us to maximize the sharing of common data across system modules while avoiding memory leaks due to programming errors.

D. Data Authentication

Data security is one of the key requirements for IoT applications. However, because of the limited CPU and memory resources, constrained IoT devices have limitations on which cryptographic operations they can support. Nevertheless, with the help of the hash API in RIOT-OS kernel and a third-party *micro-ecc* library [10], we were able to implement three practical data authentication methods that can be used by IoT applications: AES-CCM [11], HMAC [12], and ECDSA [13].

[1]RIOT-OS does not support virtual memory so there is no difference between processes and threads. Moreover, it does not provide separation of user-space and kernel-space, which gives kernel threads and application threads the same level of execution privilege.

Note that AES-CCM is an authenticated encryption algorithm, which provides both data authenticity and secrecy. Our implementation uses the standard `secp256r1` curve [14] for ECDSA with a signature length of 64 bytes. RSA signature algorithm is not supported due to its prohibitive computation cost.

One particular challenge we faced when implementing ECDSA support was that a lot of constrained IoT devices lack the hardware entropy source for generating cryptographically secure random numbers, which is a critical step in the ECDSA signing process. Using a pseudo-random number generator (PRNG) without a strong source of entropy significantly impairs the strength of the ECDSA signatures. As a current solution we adopted the use of deterministic ECDSA signing [15], which does not use true random numbers when generating the signatures. However, creating ECDSA key pairs still requires cryptographically strong random numbers. One way to achieve this is to let the applications on RIOT-OS use ECDSA key pairs created on other platforms and transported to the IoT devices. The detailed mechanism of exporting and configuring ECDSA keys is under development and will be described in NDN-RIOT documentation.

E. Packet Forwarding

To support the NDN packet forwarding logic, NDN-RIOT includes the simplified versions of PIT, FIB, and CS data structures. In order to minimize memory overhead, all three data structures are implemented as simple linked lists, as we expect low numbers of entries (under 100) on constrained devices. The current packet processing does not support Interest selectors and all lookups are performed only using Interest and Data names. The implementation also does not support retrieval of data using full names including the implicit digest component, as IoT applications are primarily used for retrieving data samples and processing commands, which typically do not require the use of implicit digest.

The PIT uses the exact match for the Interest name and any prefix match for the data name (i.e., the Data packet will match any PIT entry with name that is shorter or the same as the data name). The simplified PIT entries record only incoming faces for the Interests. During forwarding, PIT state is used to prevent potential loops without using the nonce mechanism.

The FIB table implements the longest-prefix match using Interest names and stores unranked arrays of faces, which can be configured either through static configuration or using a simple IPC-based mechanism at run time. The latter is currently available only for local application prefix registration. Propagation of the routing information (proactive or on-demand) to remote nodes, especially over wireless mesh networks, is left for our future work. Additional IPC-based mechanism for dynamic FIB configuration is also a part of our future plan.

The CS uses a pre-configured maximum size (current compile-time adjustable default is 24 KB) and implements a simple FIFO cache eviction policy. It benefits from the shared memory block design described in Section III-C and stores the shared pointer to the cached Data packet rather than making a deep copy.

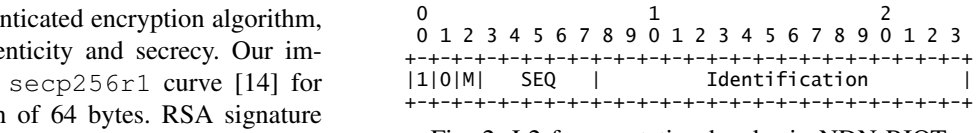

Fig. 2: L2 fragmentation header in NDN-RIOT

We also implemented a simplified version of the forwarding strategy framework described in the NFD developer guide [16]. The forwarding strategy inserts custom operations using callback functions at three critical points in the NDN packet processing pipeline: *after an Interest packet is received*, *before a PIT entry is satisfied by a received Data packet*, and *before an expired PIT entry is removed*. The strategy callback functions have full access to the PIT, FIB, and CS tables and can make forwarding decisions such as (re)transmitting a packet to a specific face or drop the current packet based on the logic of the strategy. When NDN-RIOT application is deployed, one of the currently available (*best-route unicast* and *multicast*) or a custom-defined strategy can be selected to handle forwarding of all Interests. Unlike full-featured NFD strategy framework, NDN-RIOT supports only one strategy at a time, which we expect to be sufficient for the expected uses of the platform.

F. Layer-2 Communication

RIOT-OS currently supports two types of L2 protocols: Ethernet (used by the emulator) and IEEE 802.15.4 (on real devices). When sending NDN packets over Ethernet, the network device driver sets the destination MAC address to the broadcast address (FF:FF:FF:FF:FF:FF). The Ethernet header also carries the IEEE 802 protocol number for NDN so that the packet receiver can detect the packet type and dispatch the packet to the NDN thread. When operating over IEEE 802.15.4 links, the devices tune into the pre-selected wireless channel (identified by the channel ID number, e.g., 26) and Personal Area Network (PAN) (identified by the PAN ID, e.g., 0x23) and use broadcast destination address (FF:FF).

Most constrained networks have a very limited MTU size, typically less than 100 bytes. Although IoT applications optimized for constrained environments should try to avoid using big packets, it is too restrictive to require all applications to always send NDN packets that can fit into the network MTU. Therefore, we also designed a lightweight hop-by-hop L2 fragmentation and reassembly mechanism for NDN-RIOT. Fig. 2 shows the format of the 3-byte fragmentation header that is prepended to every fragment of an NDN packet if it needs L2 fragmentation. The header format is optimized for constrained environments by packing all information into 24 bits. The first bit of the header is set to 1 to indicate the packet is fragmented. Unfragmented packet will start with the type code for Interest (5) or Data (6) packet, whose highest-order bit is always 0. The More-Fragment (MF) bit indicates whether the current packet is the last fragment. The sequence

```
static ndn_app_t* handle = NULL;

static int on_data(ndn_block_t* interest,
                   ndn_block_t* data) {
  ndn_block_t content;
  ndn_data_get_content(data, &content);
  // do something with content...
  return NDN_APP_STOP;
}

static int on_timeout(ndn_block_t* interest) {
  ndn_block_t name;
  ndn_interest_get_name(interest, &name);
  ndn_name_print(&name);
  return NDN_APP_STOP;
}

static int send_interest(void* context) {
  const char* uri = (const char*)context;
  ndn_shared_block_t* sn =
    ndn_name_from_uri(uri, strlen(uri));
  ndn_app_express_interest(handle, &sn->block,
                           NULL, 4000,
                           on_data, on_timeout);
  ndn_shared_block_release(sn);
  return NDN_APP_CONTINUE;
}

void run_consumer(const char* uri) {
  handle = ndn_app_create();
  ndn_app_schedule(handle, send_interest,
                   (void*)uri, 1000000);
  ndn_app_run(handle);
  ndn_app_destroy(handle);
}
```

Listing 1: Skeleton NDN consumer app for RIOT-OS

```
static ndn_app_t* handle = NULL;

static int on_interest(ndn_block_t* interest) {
  ndn_block_t in;
  ndn_interest_get_name(interest, &in);
  ndn_shared_block_t* sdn =
    ndn_name_append_uint8(&in, 0);
  ndn_metainfo_t meta = {NDN_CONTENT_TYPE_BLOB, -1};
  uint8_t buf[20] = {0x23};
  ndn_block_t content = {buf, sizeof(buf)};
  ndn_shared_block_t* sd =
    ndn_data_create(&sdn->block, &meta, &content,
                    NDN_SIG_TYPE_ECDSA_SHA256, NULL,
                    ecc_key, sizeof(ecc_key));
  ndn_shared_block_release(sdn);
  ndn_app_put_data(handle, sd);
  return NDN_APP_CONTINUE;
}

void run_producer(const char* prefix) {
  handle = ndn_app_create();
  ndn_shared_block_t* sp =
    ndn_name_from_uri(prefix, strlen(prefix));
  ndn_app_register_prefix(handle, sp, on_interest)
  ndn_app_run(handle);
  ndn_app_destroy(handle);
}
```

Listing 2: Skeleton NDN producer app for RIOT-OS

number (SEQ) and identification fields provide ordering of the fragments, which are used by the receiver to reassemble the original NDN packet.

G. Application Interface

We implemented a set of high-level application interface that abstracts away the internal communication mechanism between the application thread and the NDN stack. Those APIs provide asynchronous communication model that is widely adopted in existing NDN client libraries such as NDN.JS [17] and ndn-cxx [18]. Under this model, an application runs an event loop that dispatches I/O events (e.g., packet received, timer expired, etc.) on a single thread and invokes the corresponding callbacks to handle those events. The application may also create multiple run loops in different threads to schedule different tasks. Table I shows the core APIs that are frequently used in NDN applications.

We illustrate the usage of the APIs in Listings 1 and 2, which show the code of skeleton consumer and producer applications, respectively. Due to space limit, we omit all the error checking code and the main function that starts the app. The complete source code can be found at [7].

IV. Evaluation

In this section we present the evaluation results we gathered from real IoT devices. We focus on three aspects, memory usage, execution speed, and power consumption, all of which are critical for software systems running on constrained devices. Our benchmark code is compiled with the GCC ARM Embedded toolchain on Ubuntu 16.04, which is based on GCC version 4.9.3. We follow the default GCC settings from the RIOT-OS codebase, which uses level-2 optimization (`-O2`).

We perform the evaluation on two different IoT platforms:

- **SAMR21-XPRO** [19]: an IoT evaluation board produced by Atmel, which has a 32-bit ARM Cortex-M0+ 48 MHz microcontroller (MCU) with 32KB embedded RAM and 256KB embedded flash, and a 2.4 GHz IEEE 802.15.4 compliant radio interface.
- **IoTLab-M3** [20]: an IoT board designed for the FIT IoTLab [21], which has a 32-bit ARM Cortex-M3 72 MHz MCU with 64KB embedded RAM and 512KB embedded flash, and the same type of radio interface as in SAMR21-XPRO.

A. Memory Usage

We measure the memory usage of the NDN-RIOT applications by inspecting the binary object code compiled for the two platforms. We use the GNU `readelf` tool to obtain the code size of each API function, and the GNU `size` tool to obtain the total code size and the static memory usage from the executables of the skeleton consumer and producer examples shown in Section III-G.

Table II lists the object code size of the core APIs in NDN-RIOT. The second column shows the size of the APIs compiled for the ARM Cortex-M0+ MCU, which is based on the ARMv6-M Instruction Set Architecture (ISA). The third column shows the size of the APIs compiled for the ARM

TABLE I: Application interface in NDN-RIOT

Function	Description
`ndn_app_create`	Create a handle for the NDN application and initialize resources
`ndn_app_run`	Start the application's run-loop and block until the application terminates
`ndn_app_destroy`	Release the application's handle and associated resources
`ndn_app_schedule`	Schedule a callback function to be executed after the specified time interval
`ndn_app_express_interest`	Send an Interest and register the callbacks for processing the retrieved Data and the Interest timeout event
`ndn_app_register_prefix`	Register the prefix in the local FIB and the callback for processing the received Interests that match the prefix
`ndn_app_put_data`	Send a Data packet to the NDN thread to consume previously received Interests

TABLE II: Object code size of the NDN-RIOT API (in bytes)

Function Name	ARMv6-M	ARMv7-M
`ndn_name_from_uri`	420	408
`ndn_name_append`	232	232
`ndn_name_get_size_from_block`	124	124
`ndn_name_get_component_from_block`	152	164
`ndn_interest_create`	196	192
`ndn_interest_get_name`	92	94
`ndn_data_create`	668	692
`ndn_data_get_name`	98	100
`ndn_data_get_content`	160	168
`ndn_data_verify_signature`	450	502
`ndn_app_run`	612	596
`ndn_app_schedule`	96	88
`ndn_app_express_interest`	160	168
`ndn_app_register_prefix`	180	180
`ndn_app_put_data`	60	56

TABLE III: Overall memory usage of the skeleton NDN consumer and producer apps for RIOT-OS (in bytes)

ISA	App	text	data	bss	Flash	RAM
ARMv6-M	Consumer	35,300	192	11,208	35,492	11,400
ARMv7-M	Consumer	33,900	192	11,208	34,092	11,400
ARMv6-M	Producer	35,212	192	11,208	35,404	11,400
ARMv7-M	Producer	33,800	192	11,208	33,992	11,400

Cortex-M3 MCU, which is based on the ARMv7-M ISA. Both architectures support the Thumb instruction set with 16-bit/32-bit encoding, which leads to similar code sizes.

Table III shows the output from the GNU `size` command for the skeleton consumer and producer examples sketched out in Listings 1 and 2. The source code of both examples have roughly the same structure, which therefore results in similar code sizes after compilation. The last two columns in the table show the total amount of static memory residing in flash and RAM, respectively. On a device with 32KB RAM, the static data occupies ≈11KB of the embedded RAM,[2] leaving 21KB for dynamic allocation, such as creating PIT, FIB and CS entries, creating shared memory blocks for NDN packets, and storing dynamic user data generated at run-time.

B. Performance

To gauge the run-time performance of the system, we first analyze the execution speed of individual APIs through a set of benchmarks, and then show the application-level RTT between two directly-connected nodes as an indicator for the packet processing speed of NDN-RIOT. We did not measure the maximum network throughput since most IoT applications running on constrained devices do not require high throughput data transmission.

TABLE IV: Execution time of the NDN-RIOT APIs

Test Case	SAMR21-XPRO		IoTLab-M3	
	Time (μs)	Cycles	Time (μs)	Cycles
URI to Name	184	8,832	282	20,304
Get Name size	13	624	11	792
Get Name component	8	384	7	504
Append to Name	28	1,344	29	2,088
Create Interest	25	1,200	23	1,656
Get Interest Name	2	96	2	144
Create Data (HMAC)	1,806	86,688	1,333	95,976
Create Data (ECDSA)	451,215	21,658,320	269,314	19,390,608
Verify Data (ECDSA)	500,115	24,005,520	294,225	21,184,200
Get Data Name	3	144	2	144
Get Data Content	4	192	4	288

1) API execution speed: We measure the execution time of the NDN-RIOT APIs by calling the functions repeatedly and dividing the total running time by the number of iterations. The results are presented in Table IV in both real time and MCU cycles for comparison across MCUs. Since the benchmark suites run as a single-threaded application that takes over the whole MCU, the measurement results are quite stable over different runs.

To summarize, on SAMR21-XPRO, NDN-RIOT is able to create 5,434 NDN names (from URI strings), 40,000 Interests (using pre-generated name objects), or 553 HMAC-signed Data per second. The most expensive operations are creating and verifying ECDSA-signed Data packets. SAMR21-XPRO can create and verify ≈2 Data packets with ECDSA signatures per second. IoTLab-M3 performs the same operation at 3.5–3.7 Data packets per second,[3] but it is still ≈150 times slower than using HMAC.

2) Packet processing speed: Our final evaluation measures the packet processing delay of the NDN-RIOT implementation by the application-level RTT of Interest-Data exchange between two directly-connected NDN-RIOT nodes. The experiments are carried out on the FIT IoTLab testbed [21] in Paris with two IoTLab-M3 nodes communicating directly with

[2]The static data in the .data and .bss sections includes fixed-size stack for each thread and other pre-allocated global objects.

[3]We noticed that the load/store instructions execute slower than expected on IoTLab-M3, causing several "memory-bound" test cases to run much slower than on SAMR21-XPRO. The reason for the slow memory access is unclear.

TABLE V: Packet processing delay in Interest-Data exchange

Data Size	Cached?	Fragmented?	RTT (ms)
100 bytes	No	No	280
	Remote	No	11
	Local	No	<1
196 bytes	No	Yes	286
	Remote	Yes	16
	Local	No	<1

each other over IEEE 802.15.4 radio without interference from neighboring nodes. The testbed network has an MTU of 102 bytes and a fixed data rate of 250 Kbps.

The measurement is performed under two scenarios with the Data packet size of 100 bytes and 196 bytes. In each scenario, we measure the RTTs of *fetching new Data generated by the producer upon request*, *fetching pre-generated Data from the cache of the remote node (i.e., the producer)*, and *fetching pre-generated Data from the consumer's local cache*. Each experiment performs 100 Interest-Data exchange without pipelining and the RTT is calculated as the total running time divided by 100. Since the experiments are executed in an isolated environment, there is no packet loss during the measurement. All Data packets are signed by ECDSA.

Table V shows the RTT measurement results. When the consumer fetches newly created data, the RTT is dominated by the ECDSA signing delay (which takes ≈270 ms) for both packet sizes. When the consumer fetches 196-byte Data, the packet is fragmented into two pieces at the producer and reassembled at the consumer node, and the RTT shows ≈6 ms additional delay compared to the results without fragmentation. When the consumer fetches data from its local cache, no fragmentation is required for either packet size and the response time is less than 1ms.

The average RTT of IP packets between two RIOT-OS devices in the testbed environment ranges from 5 to 9 ms as measured by the `ping` utility in the RIOT-OS kernel. Thus, the Interest-Data RTT of NDN-RIOT, which is yet to be fine tuned, is comparable to that of the IP stack (without the data signing delay). However, we note that the former can express a higher level semantics and directly feed data to applications, while an IP packet must go through additional layers of protocol processing, plus extra round trips for naming resolution and secured session setup, before reaching the application layer as explained in [22].

C. Power Usage

We measure the power usage of the NDN-RIOT APIs using the IoTLab-M3 node, which is running on a 3.7V LiPo battery with 650 mAh capacity[4]. The IoTLab testbed [21] conveniently provides the profiling tools for the users to collect the power consumption data at certain sampling rate from the testbed nodes. We run the same test suite used in Section IV-B1 but increase the running time of each test case to 30 seconds in order to minimize the noise in the power data sampling. We first measure the baseline power consumption of the M3 node when it is idle, which is about 0.14 W. The power usage of each test case is then calculated by subtracting the baseline power from the power when the test case is running. The average running time of each API is calculated as the total run-time divided by the number of repeated runs. Finally, the energy consumption is calculated as power times the average run-time.

[4]https://www.iot-lab.info/hardware/m3/

TABLE VI: Power usage of the NDN-RIOT APIs

Test Case	Power (mW)	Time (μs)	Energy (nJ)
URI to Name	5	273	1,365
Get Name size	7.5	12	90
Get Name component	8.5	8	68
Append to Name	8.5	30	255
Create Interest	9.5	23	218.5
Get Interest Name	9.5	2	19
Create Data (HMAC)	12	1,333	15,996
Create Data (ECDSA)	17	269,325	4,578,525
Verify Data (ECDSA)	17	294,237	5,002,029
Get Data Name	11	3	33
Get Data Content	9.5	4	38

The measurement results are listed in Table VI.[5] To put these numbers into perspective, consider a simple sensing application running on M3 that wakes up and reads the sensor measurement every 1 minute, packages the data point in an NDN data packet signed by ECDSA, immediately transmits the packet over IEEE 802.15.4 interface, then goes back to sleep. The Data packet generation will take about 300 ms using 0.157 W of power (including the baseline), which results in a total of 47.1 mJ. The IEEE 802.15.4 transceiver on the M3 node running at maximum transmit power consumes 14 mA at 3.6 V by itself,[6] and takes less than 4 ms to transmit a full-MTU-size packet, which costs a total energy of about 0.2 mJ. With the 650 mAh battery, the M3 node could (in theory) run the application for 127 $(= 650 * 3.7 * 3600/47.3/60/24)$ days without recharging. Note that this estimation does not cover the energy consumed by the sensor motes or during the transition between hibernation and wake-up.

V. Related Works

Traditional lightweight network stack implementations for embedded systems are based on the TCP/IP architecture. One of the most popular implementations is the lwIP [23] package that provides full-featured standard-compliant TCP/IP functionalities on devices with 10s of KB of RAM and flash memory. lwIP can run on different platforms, including RIOT-OS-based IoT systems. RIOT-OS also has its own IPv6-based network stack that supports IoT applications by incorporating IoT-related protocols and frameworks such as CoAP, RPL, and 6LoWPAN. NDN-RIOT follows the same software architecture of the IP stack in RIOT-OS in order to achieve good integration with the RIOT-OS kernel.

[5]The results of some test cases are sightly different from those in Table IV because they were collected in different runs.

[6]http://www.atmel.com/images/doc8111.pdf

CCN-lite [24] is a generic lightweight implementation of CCN [1], which has been ported to RIOT-OS as a third-party package [25]. CCN-lite is designed as a generic CCN stack that can run on different platforms and support multiple ICN protocol formats. In contrast, NDN-RIOT is an optimized version of the NDN stack for IoT applications and the RIOT-OS platform. Consequently, our implementation can fully utilize the RIOT-OS internal APIs and remove the redundant functionalities. More importantly, our NDN APIs provide data security support which is critical for IoT applications but currently missing in the CCN-lite implementation.

VI. Conclusion and Future Work

Our implementation of NDN-RIOT, a lightweight NDN protocol stack for RIOT-OS demonstrates the feasibility of bringing NDN's data-centric communication and security model to constrained IoT platforms, providing a solid foundation for developing more comprehensive IoT applications than TCP/IP protocol stack. However it is only a first step toward NDN-enabled IoT applications, with several important areas yet to be explored.

In the future, we plan to carry out multiple improvements to the current implementation that have been identified, including extending the APIs with transport-level functionalities such as Interest pipelining and automatic retransmission, adding data encryption support, and tuning the performance through micro-benchmark. We also plan to design high-level frameworks, such as auto-configuration and discovery, on top of NDN-RIOT to facilitate the IoT applications. We invite the broader community to join us in exploring this exciting area of IoT research.

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking Named Content," in *Proc. of CoNEXT*, 2009.

[2] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named Data Networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, Jul. 2014.

[3] W. Shang, A. Bannis, T. Liang, Z. Wang, Y. Yu, A. Afanasyev, J. Thompson, J. Burke, B. Zhang, and L. Zhang, "Named Data Networking of Things," in *Proc. of IoTDI*, 2016.

[4] Y. Zhang, D. Raychadhuri, L. A. Grieco, E. Baccelli, J. Burke, R. Ravindran, G. Wang, B. Ahlgren, and O. Schelen, "Requirements and Challenges for IoT over ICN," Internet-Draft draft-zhang-icnrg-icniot-requirements-01, Apr. 2016.

[5] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. of INFOCOM Comp. Comm. Workshops*, 2013.

[6] RIOT-OS Project Team, "RIOT-OS homepage," https://www.riot-os.org/.

[7] W. Shang and A. Afanasyev, "NDN protocol stack for RIOT-OS," https://github.com/named-data-iot.

[8] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, "A case for stateful forwarding plane," *Computer Communications*, vol. 36, no. 7, pp. 779–791, 2013.

[9] NDN Project Team, "NDN Packet Format Specification," http://named-data.net/doc/ndn-tlv/.

[10] K. MacKay, "micro-ecc: ECDH and ECDSA for 8-bit, 32-bit, and 64-bit processors." https://github.com/kmackay/micro-ecc.

[11] D. Whiting, R. Housley, and N. Ferguson, "Counter with CBC-MAC (CCM)," RFC 3610, Sep. 2003.

[12] D. H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997.

[13] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)," ANSI X9.62-2005, November 2005.

[14] Certicom Research, "SEC 2: Recommended Elliptic Curve Domain Parameters (Version 2.0)," http://www.secg.org/sec2-v2.pdf, January 2010.

[15] T. Pornin, "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," RFC 6979, Oct. 2015.

[16] A. Afanasyev *et al.*, "NFD Developers Guide," NDN Project, Tech. Rep. NDN-0021, Revision 6, mar 2016.

[17] W. Shang, J. Thompson, M. Cherkaoui, J. Burkey, and L. Zhang, "NDN.JS: A JavaScript Client Library for Named Data Networking," in *Proc. of INFOCOM Comp. Comm. Workshops*, 2013.

[18] NDN Project Team, "ndn-cxx overview," http://named-data.net/doc/ndn-cxx/current/README.html.

[19] Atmel Corporation, "SAM R21 Xplained Pro Evaluation Kit," http://www.atmel.com/tools/ATSAMR21-XPRO.aspx.

[20] FIT Consortium, "M3 Open Node," https://www.iot-lab.info/hardware/m3/.

[21] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed," in *Proc. of IEEE WF-IoT*, Dec 2015.

[22] W. Shang, Y. Yu, R. Droms, and L. Zhang, "Challenges in IoT Networking via TCP/IP Architecture," NDN, Technical Report NDN-0038, 2016.

[23] A. Dunkels, "Design and Implementation of the lwIP TCP/IP Stack," Feburary 2001.

[24] CCN-lite, "CCN-lite homepage," http://www.ccn-lite.net/.

[25] E. Baccelli, C. Mehlis, O. Hahm, T. C. Schmidt, and M. Wählisch, "Information Centric Networking in the IoT: Experiments with NDN in the Wild," in *Proce. of ICN*, 2014.