RIOT-ROS2: Low-Cost Robots in IoT Controlled via Information-Centric Networking

Loïc Dauphin, Emmanuel Baccelli, Cédric Adjih
Inria, France
Email: <FirstName>.<FamilyName>@inria.fr

***Abstract*—In the future, IoT devices will be part of the robotics ecosystem, and the border between IoT and robotics will blur. Already today, we observe converging trends between low-end IoT devices and minibots (i.e. tiny, cheap robots) concerning their hardware, and open source software. In this paper, we explore the potential of programming minibots with the open source robotics software framework ROS2, running on top of the IoT operating system RIOT; we call the fruitful association of both elements *RIOT-ROS2*. In this article, the emphasis is particularly on the networking layer: using an *information-centric networking* (ICN) paradigm, we design and implement the communication primitives for RIOT-ROS2. We further evaluate the performance of our design on prototype minibots based on cheap, off-the-shelf hardware elements. We show that RIOT-ROS2 fits on low-end robotics hardware such as a System-on-Chip costing under $2, based on an ARM Cortex-M0+ microcontroller. Our experiments also show that the latency incurred with our information-centric approach is acceptable for minibot control, even on a low-throughput IEEE 802.15.4 radio.**

I. Introduction

In the future, IoT devices will be part of the robotics ecosystem and the border between IoT and robotics will blur. Over the last decade, a large variety of cheap, networked single-board computers have become available. Such devices, also known as low-end IoT devices (or constrained devices [1]) are based on microcontrollers with very limited CPU, memory, and power resources. For instance, compared to a RaspberryPi ($>$ 1 Watt [2]), low-end IoT devices consume 10^3 less power, embark 10^6 less memory and are orders of magnitude slower.

Meanwhile, *minibots* (miniature robots) have appeared on the market. A large community emerged, designing do-it-yourself minibots [3], and cheap, re-programmable minibots with communication capabilities are now available. For instance, small wheeled robots such as the Zooid [4] are based on a small microcontroller (8kB RAM, 64kB ROM) and communicating with a low-power radio in the 2.4 GHz ISM band. Other examples are cheap drones such as the Cheerson CX-10 [5], which has similar hardware characteristics, and which costs under 15$. Simple robotic arms and legged robots are also available, such as the MetaBot [6].

A current trend bases software embedded in minibots on open source frameworks. The Robot Operating System (ROS [7]) is a software framework for robot application development which has become a de facto standard for most areas in robotics. Other open source robotics frameworks include software suite tailored for drones [8], some of which provide compatibility with ROS [9], [10].

In fact, we observe that minibots have a number of characteristics in common with low-end devices found in the Internet of Things (IoT). Compared to low-end IoT devices [1], minibots are based on similar hardware and their software follows similar trends. For instance, an IoT-enabled actuator based on a System-on-Chip (SoC) embarking a small microcontroller, and a radio communicating with a remote server, is very similar to a simple radio-controlled robot. Low-end IoT devices use similar radio modules [11], and software embedded in IoT devices is more and more based on a variety of open source, lightweight operating systems such as RIOT [12], FreeRTOS [13] and NuttX [14], among others [11].

Similarly, as for IoT embedded systems, the network component of minibots represents by itself in important part of the software (in terms of features, code/memory size, and performance). In fact, a wide variety of radio modules and communication protocols are used on minibots. The protocols used by micro-robots for (internal or external) communication range from direct motor control (pulse width modulation PWM, pulse position modulation PPM, or PCM), to serial/bus protocols, and high level protocols such as Real-time Publish-Subscribe Protocol (RTPS) [15], [16].

The goal of this paper is thus to explore the potential of bundling open source robotics software frameworks with IoT software and network architectures, to program and control minibots. To do so, we extend [17] by designing ROS-ready technology for a minibot based on RIOT [12] and ROS2. We focus primarily on software and networking aspects, targeting ultra-lightweight robots based on a reprogrammable SoC with a microcontroller running at $\approx$50 MHz, with $\approx$10kB RAM, $\approx$100kB Flash, and a low-power radio. Using an information-centric networking paradigm extending NDN, we design and implement the communication primitives required by RIOT-ROS2. Our prototype is able to maintain full compatibility between ROS nodes running on the minibot(s) and ROS nodes running elsewhere on the network without the use of a bridge. We show that RIOT-ROS2 fits on low-end robotics hardware such as a System-on-Chip with an ARM Cortex-M0+ microcontroller [18]. On the software and network performance evaluation side, we illustrate that the latency incurred with our ICN approach is completely acceptable for minibot control, even on constrained radio, based on micro-benchmarks.

A. Related Work

In the fields of robotics, we observe a trend towards **miniature and distributed software architectures** and open source. ROS2 [19] proposes an update of ROS aiming at better support of multi-robot system, smaller processors and unreliable network connectivity. Projects have been initiated, such as micro-ROS [20], which aims to adapt ROS2 to a variety of micro robots based on small microcontrollers. Typical examples and design trade-offs for minibots are offered by a variety of small drones which have recently appeared. Recent work [21] designed a comprehensive UAV platform using open-source components for a cost well under $1000. The most popular frameworks and flight stacks for micro aerial robots include a mechanism [9] for connecting the flight logic with ROS. However, efforts to bring ROS support directly on smaller robots [22] have so far left aside resource-constrained microcontrollers and minibots.

Practical **IoT robotics** has started to be explored in recent work such as [23] [24] which introduce building blocks for a cloud-controlled IoT robot based on small microcontrollers. Reliable IoT wireless protocols, TDMA-based, such as time slotted channel hopping over IEEE 802.15.4 radio have demonstrated ability to control robotic elements in real-time [25]. In [17], a prototype demonstrates a wheeled IoT robot running RIOT and ROS2 based on an ARM Cortex M0+ microcontroller and IEEE 802.15.4 communication.

In all IoT robotics software architectures, the network is essential: communication should build upon network stacks which specifically target the wireless context of IoT (embedded constrained devices) and with a tiny memory footprint. Examples of IoT network stacks include the standard 6LoWPAN network stack [26] based on IPv6 protocols, or experimental stacks based on novel paradigms such as **information-centric networking (ICN)** as described in [27]. Here, we focus on ICN, which fits well that the architecture we consider for robotics use cases (as detailed in Section IV-C). We have selected the Named-Data Network (NDN) protocol [28], for which there exists an implementation for IoT (and RIOT [29]). While a number of ICN publish/subscribe architectures have been proposed, such as COPSS [30] for instance, our emphasis is on the use of unmodified version of ICN protocols (to further allow for portability including between different implementations for microcontrollers and desktop computers/servers): some building blocks for the real-time streaming of information are presented in NDN-RTC [31], and some building blocks for discovery and spreading of information in synchronization protocols (such as VectorSync [32] or iRoundSync [33]), see the survey in [34]).

II. Design Choices

A. *Targeted Systems, Use-Cases and Requirements*

Our design choices should enable ultra-lightweight robots. As a reference, we design the system such that minibots roughly similar to the Zooid [4] or the Cheerson CX-10 [5] remain in scope. More generally, the low-end of robots we target are based on (re)programmable SoC, with a microcontroller running at $\approx$ 50 MHz, with $\approx$ 10kB RAM, $\approx$ 100kB Flash, and a low-power radio. In our actual experiments, such systems are namely SAM R21 Xplained Pro boards [18] with an Atmel ARM Cortex-M0+, 48 MHz, 32kB of RAM, 256kB of Flash and IEEE 802.15.4 radio (at 2.4 GHz, 250 kbps).

Typically, the software developed could (but not only) be used for a simple 2-wheeled robot wirelessly remote-controlled similar to the one demonstrated in our prior work such as [17] (video at https://www.youtube.com/watch?v=YvPssYSgLYY). In [17], the following simple system is used: one (ROS2) module produces measurements of the angle (through a gyroscope sensor) on one board held by the user, whereas on the robot, there is another module acting as a consumer and adapting the motion of the minibot to wireless received measurements (in the demo, the angle). A more general control would let the angle and the linear speed of the robot be controlled remotely by a trajectory generator. In general, to achieve an acceptable open-loop control on such robot, the required performance is on the order of transmitting 10 commands per second (and a few times the minimum would be comfortable). For a closed-loop control, an additional 10 position data per second should be sent to the computer.

B. *Modularity, Interoperability & Portability*

Our goal is to be able to reuse the same software framework, and further, the same API, in deeply embedded systems and in desktop/cloud computers. In our case, for the robotics framework, we chose ROS2, the latest generation of the popular platform ROS. For the embedded software platform, we chose a generic IoT operating system: RIOT. Both are actively developed in their respective communities: ROS2 as a new generation of ROS, and RIOT as integration platform for more and more IoT protocol stacks and hardware. The advantage of this approach is twofold: on one hand such frameworks offer a familiar environment for developers, and on the other hand their architecture favors application modularity, reusability and portability, which has numerous (interrelated) advantages in terms of software engineering:

- **Portability:** both ROS2 and RIOT are designed with portability in mind: this means for instance that a user is able to change the microcontroller board with minimal changes (if at all) in the code.
- **Configurability**: any ROS2 node that does not rely on the hardware is able to be executed on any connected hardware, preserving some flexibility in the final application design. For instance, if a ROS2 node turns out to be too computationally intensive for the microcontroller on the drone, it can be offloaded on a controlling desktop (assuming sufficient communication capabilities).
- **Interoperability**: the same system is ported on several platforms (embedded or not). This, implicitly, allows the nodes of different systems to be able to communicate together, independently of their location. The same concept applies to interoperability over different communication interfaces (e.g. wireless and wired) which has traditionally been a complicated and conflicting task in robotics when using ROS.

The above features are building blocks to construct more sophisticated abilities, such dynamic offloading from one entity to another. For instance, dynamic delegation of the high-level control from the microcontroller to an Internet application – and conversely, should communication be disrupted.

C. Network Layer Flexibility

Our design choices should be agnostic to communication protocols – as much as possible. Concretely, this means that it should be able to run on top of different protocol stacks. ROS2 and RIOT are designed with such modularity in mind. As shown in Fig. 1, ROS2 defines a low-level API to plug in various communication layers, called ROS MiddleWare (RMW). On the other hand, RIOT offers several types of IoT communication protocols stacks. Based on a dedicated RMW, we were able to provide the communication primitives needed by ROS2 by combining the NDN protocol stack provided by RIOT [35] and serialization of data using CBOR [36], also supported by RIOT. Notice that with minimal changes, we were able to provide the same communication primitives using another stack based on MQTT and 6LoWPAN, also supported by RIOT (although we do not present experimental measurements for that implementation in this article).

III. ROS2 Basics

From the point of view of ROS2, a robotic application is a set of abstract software modules, called *Nodes*, that are distributed on a set of computing devices communicating via a common network.

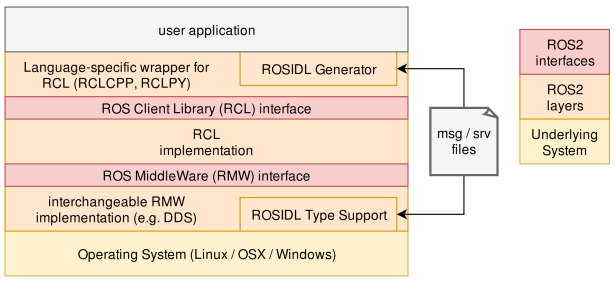
user application
Language-specific wrapper for RCL (RCLCPP, RCLPY)
ROSIDL Generator
ROS Client Library (RCL) interface
RCL implementation
ROS MiddleWare (RMW) interface
interchangeable RMW implementation (e.g. DDS)
ROSIDL Type Support
Operating System (Linux / OSX / Windows)
msg / srv files
ROS2 interfaces
ROS2 layers
Underlying System

Fig. 1: ROS2 architecture

The concepts for communications of ROS2 are:

- Nodes: they correspond to one module (of the user application, typically run in one thread or one process).
- Topics: communication channels used to distribute data from producers (Publishers) to consumers (Subscribers).
- Service: equivalent to a client/server communication in a request/response pattern (or remote procedure calls).

Nodes communicate with each other with well-defined interfaces, which are basically named channels of communication using structured data. In the example [17], one node publishes gyroscope measurements in the topic `/gyro/angle`, while another consumes them and sets the angle of the robot.

As shown in Fig. 1, ROS2 is designed as a stack separated into 3 layers. ROS2 separates managing the networking channels (left), and managing the data structure (right).

The upper layer is the user API: language-specific wrappers for ROS Client Library expose the ROS features for a supported programming language (e.g. Python with rclpy, or C++ with rclcpp). An implementation of this upper layer must wrap a C interface called RCL (ROS Client Library), which defines the basic features to be exposed. Since most languages have a compatibility layer with C, this potentially enables ROS2 extensions to a variety of languages.

The middle layer is a library that provides the glue between the RCL and RMW interfaces. The lower layer is the ROS MiddleWare implementation, which provides the networking primitives enabling ROS nodes to communicate with one another. This library leverages the underlying system's middleware (typically Data Distribution Service or DDS [37]) to implement the RMW (ROS MiddleWare) interface.

ROS2 defines the interface between nodes via `msg` and `srv` files. These files are basically defining C-like structs, but need to be transformed into actual code to be compiled and executed. The ROS2 tool which generates code from these files is ROSIDL. Similarly to the rest of the ROS2 stack, the ROSIDL-generated code must expose 2 faces. On one hand, `rosidl_generator` is exposed to users, so that they can manipulate the data as if it were a native structure in the language used. On the other hand, `rosidl_typesupport` is used by the RMW layer to serialize data sent on the network.

IV. IoT Communication Primitives for ROS2

A. Required Features for RMW

As described previously, ROS2 relies entirely on the RMW to encapsulate communications. ROS2 operates by sending and receiving messages on named channels, instead of IP addresses. The messages are typed messages with a well defined structure (such as in CDR) and are serialized typically by the RMW. The default communication middleware of ROS2 is an implementation of DDS [38], an OMG standard for machine-to-machine communication using a publish/subscribe pattern based on the RTPS [15] protocol.

Although DDS as a protocol was thought as being an integral part of the design of ROS2, the middleware interface of ROS2 is *agnostic* to DDS and as such different middleware implementations are possible.

B. Current Implementations and Discussion

The main benefits of RTPS [15], compared to the initial protocols in ROS are:

- abstraction of the IP addresses, using only names for endpoints
- multicast can be used to deliver data to several subscribers
- no requirements for a central server for discovery

Previous work attempted to port ROS2 middleware on an OS targeting microcontrollers, by also porting the network stack DDS to embedded systems. An example is based on NuttX: Nuttx-ROS2 [39] used a fork of the Tinq open source project (based on the Qeo publish/subcribe framework). Nuttx-ROS2's middleware is a light DDS implementation (possibly unmaintained), and actually does not claim to be DDS compliant. Nuttx-ROS2 was able to run on Cortex-M4 microcontroller with 1 MB flash and 192 kB RAM, while interoperating with complete ROS2 systems. However there are apparent scalability limits (especially low-end μC).

Another attempt to port ROS2 on microcontroller is through freertps [16], which is an RTPS implementation aiming to run bare-metal, without dependencies. The project managed to get rid of the RTOS altogether, and runs on microcontrollers of the same size than Nuttx-ROS2. Yet, it remains under development, and no visible progress has been observed recently.

In a nutshell: all previous attempts were focusing on porting a DDS-like, IP-based middleware while squashing the ROS2 architecture into a single layer, preventing modularity, while not resolving the memory challenge (as we are targeting much smaller microcontrollers than Cortex-M4).

A significant part of the overhead comes from using an IP stack. Since RTPS discovers nodes/topics by their names, we tried to get rid of the IP stack entirely, and instead used an information-centric communication protocol: NDN (Named Data Networking [35]). The main idea of NDN is to directly use names (for instance `/ndn/local/ros2/motor/4`) for accessing data, instead on focusing on addresses for connecting to machines (such as with IP protocols). Based on the work described [17], we could obtain a simple and very light-weight Publish/Subscribe primitive usable by ROS2.

C. NDN-based ROS2 Communication Middleware

NDN [28] is an Information-Centric networking architecture that enables to address data instead of hosts. NDN communication is driven by data consumers: instead of sending an IP packet (with an IP destination address, etc.), in NDN, a data consumer will send an "Interest" packet to the network, containing a structured name corresponding to the desired content (within some naming hierarchy, such as `/ndn/local/ros2/motor/4`). Routers use this name (instead of the IP address) to decide how to forward the Interest towards the data producer(s). Once the Interest is received by a data producer that has the requested data, a "Data" packet is sent by the producer (with also the name of the data). This Data packet is then forwarded back to the consumer, using the same path as the Interest packet. Selectors and longest prefix matching mechanisms for interest names can be used to discover resources without having their complete name.

A more general overview of NDN functioning is in [28], while the detailed semantics of NDN (focusing on routing) are in [40]. In our case, we are using NDN without modification (through its RIOT implementation [29]). In the rest of this section, we describe the full design of adapting ROS2 (RMW) on top of NDN, even for features not yet fully implemented.

D. Publish/Subscribe for ROS2 with NDN-RIOT

We present our solution for publish/subscribe on top of NDN, for the case of one publisher and several subscribers, that was implemented in the demo [17], and further developed in the article. The general case can be derived for several publishers on the same channel. We adopt the similar concepts to those of NDN-RTC [31]: NDN-RTC is a version of video streaming (same function as WEB-RTC) based on NDN.

1) Streaming data: Since ROS2 topic names have the same URL-like format as NDN names, (e.g. `/robot/control`), a direct mapping can be established. However NDN is a pull-based protocol (through interests), instead of push-based (e.g. publish), a mechanism should be designed to retrieve successive pieces of data. As in NDN-RTC, we extend the topic names with a sequence number as suffix. Here for instance: `/robot/control/1`, `/robot/control/2`, `/robot/control/3`, ... For the producer, they correspond to the names of the sequentially generated data.

For the consumers: a node which has received data up to sequence number `2`, will send an Interest with a name `/robot/control/3` (the next sequence number). Upon receiving this message, the producer (or the cache of an NDN router between them) will either reply immediately with a Data message including the latest generated content; or will deliver it whenever it is available.

Notice that potential caching of NDN in intermediate router will only help this mechanism. But in the case of memory constrained hardware, keeping the history of all publications may be impossible. Because of this, late subscribers may not be able to reach the data without a bootstrap mechanism, described in next section.

2) Sync phase: The sync phase is a means by which a subscriber retrieves the current sequence number of the publisher. This is done by sending an interest with a specific suffix "sync". However caching should be avoided so that the interest can always reach the publisher. Two solutions are possible. In the first one, the consumer sends an interest `/<topic>/sync` with the flag MustBeFresh, to which the producer answers with the current sequence number with a very low FreshnessPeriod. In the second one, the consumer appends a random suffix to the name of the first solution, with the effect that this name is unique (almost surely), hence not cached. Since the RIOT version of NDN did not fully support the freshness timeout, the random suffix solution was implemented.

3) Managing multiple producers: To extend this protocol to multiple publishers for a topic, one idea would be to append a unique suffix to the topic name, e.g. a publisher identifier. Then a prior discovery protocol (described later) would be used to collect all published identifiers, before being able to receive ROS2 topic data.

E. Real Time perspective

1) Requirements: Since latency can be a problem in robotics, we also want to tackle this problem with our implementation of publish/subscribe (again, see the demo [17], which needed real-time control over 802.15.4). Indeed, some data streams in robotics have a refresh frequency so high that latency can outmatch the period between production of two successive data, which may lead to loose synchronization between publisher and subscriber. To enable this, pipelining interesting is possible, as in NDN-RTC [31]: instead of just one interest, a window of several pending interests (with successive sequence numbers) is sent, one by one. Upon receiving data, the window advances, e.g. a new interest is sent: the protocol has the good property of self-clocking.

F. Additional Communication Primitives

This section documents additional primitives that have not yet been implemented for microcontrollers, but have been implemented on our version of the system running on Linux (not described here).

1) Discovery: ROS2 provides primitives (through its API) to discover what are the ROS2 nodes connected to the network, the topics that they publish or subscribe to, and the services that they provide. This could be handled as an instance of distributed dataset synchronization such as VectorSync [32] or iRoundSync [33], etc., where each node publishes it sets of local ROS2 publishers, ROS2 subscribers, and ROS2 nodes.

We use the solution similar to Notification Interests found for instance in VectorSync [32]. Such interests are solely used as advertisement messages, and do not retrieve any Data. Thus the solution chosen to do so is to broadcast "heartbeat" interest messages that provide all the useful information periodically. For instance, a node named `/robot1` connected to the network would send periodically notification interests named `/ros2/discovery/node/robot1`. A publisher on topic `/robot/speed` with the id `123` would send `/ros2/discovery/publisher/robot/speed/123`. Receivers are then aware of this node and this publisher.

2) Service (Request/Response): Adaptation to NDN is necessary to carry "parameters" of the requests: the two possible solutions are either to serialize the parameters and append them to the name of the interest message; or using the interest-on-interest solution where the service would retrieve the parameters of the request with a Data message.

V. RIOT-ROS2 Measurements

A. Setup

A simple setup is used to for the measurements: we create a ROS2 application with one publisher node, and one subscriber node. The hardware used is composed of two SAMR21 boards communicating over 802.15.4 (with CSMA). We perform a micro-benchmarks. Parameters: one of the boards contains a Publisher sending 20 bytes of data every 100 milliseconds while the other contains the subscriber.

B. Memory

The first essential question, is: how does RIOT-ROS2 fits on targeted microcontrollers's memory? Table I gives a preliminary evaluation of the memory requirements for the complete RIOT-ROS2 stack. Notice, at link time, parts of the code of some modules are not linked in the final binary, making the total Flash less than the sum of the individual modules. We observe that only 30% of the memory available on the SAMR21 boards is required: ≈10kB RAM and ≈78kB Flash are required. However, some modules are using dynamic allocation of RAM memory (all ROS2 modules and NDN), which make hard to evaluate the final size used for RAM, even if taken in account in Table I. We also tested on other Cortex-M4 boards, the latter absolute numbers are very similar (the main difference is that RIOT requires slightly less Flash memory). Bottomline: 70% (or more) of the RAM and of the Flash is available to host more functionalities (e.g. a minimal autopilot) and for dynamic RAM usage – which is excellent.

C. Latency

One other main performance figure of RIOT-ROS2 is the delay between the publication by the publisher and the reception by the subscriber (the latency). To evaluate it, we followed the flow of data finely, from the production by one ROS2 node on one board to its reception on the ROS2 consumer node, by recording the precise time of the data in all software components (ROS2 app., ROS2 RMW, NDN, RIOT net., radio driver). Note that in our scheme, consumer Interests are sent in advance, and do not increase latency.

Module	Flash	RAM
ROS2 - RCLC	4kB	<1kB
ROS2 - RCL	60kB	1kB
ROS2 - RMW NDN	7kB	<1kB
NDN	13kB	2kB
RIOT	38kB	5kB
Total linked in binary	78kB	10kB

TABLE I: RIOT-ROS2 static memory requirements.

Layer	Publisher	Subscriber	Total
RCL + RCLC	179 μs	125 μs	304 μs
RMW NDN	1293 μs	911 μs	2204 μs
(incl. signature)	1192 μs		
NDN + GNRC	457 μs	495 μs	953 μs
Radio			4675 μs
Total			8135 μs

TABLE II: RIOT-ROS2 average latency measurements.

The results obtained from repeating 5 experiments with 50 messages sent (totalling 250 measurements), are represented in Fig. 2 and Table II. The "xtimer" clock of RIOT (currently based on an internal microcontroller oscillator for SAMR21) is used with a global error[1] that is estimated to be within 1.3%. The results are with CSMA (incurring an avg. additional delay of 1.5 ms).

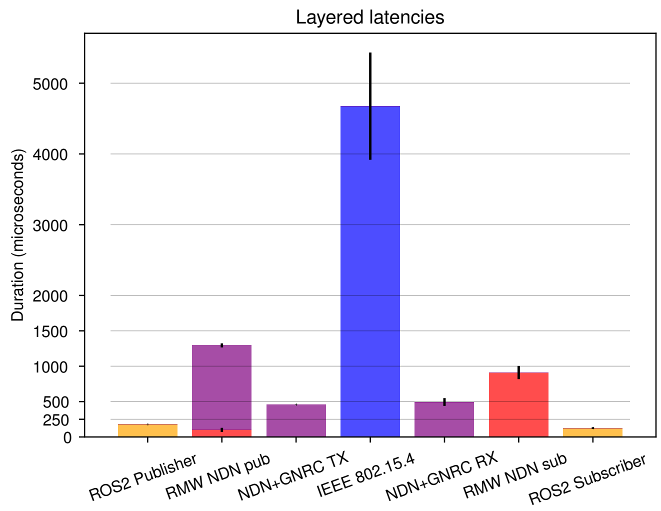

Fig. 2: Layer-wise latency

For the Fig. 2, the time spent in each communication layer was measured (std. dev. in error bars). The yellow areas are the time spent in the ROS2 Client Library (RCLC, RCL), for which the median is 179 and 125 microseconds (3.7% in total). The red area is the time spent in the RMW NDN layer,

[1]In order to synchronize the two boards, we wire them together (one board generates a 10 Hz signal received by the other via GPIOs): after linear drift correction, on all performed experiments, we observe the clocking error is within +/- 650 ppm (0.65%). Independent experimentation shows similar clocking error between precise GPS time and one board. Hence the estimate.

for which the median is 1293 microseconds (16%) for the publisher and 911 μs (11%) for the subscriber. Note that the time of the creation and signature of the NDN Data message is included in the RMW NDN layer, and takes 1192 μs, which corresponds to almost all the time of the publisher side. The purple area is the time spent in the RIOT NDN and GNRC layers, basically the cost of the RIOT communication stack when configured for NDN. The median of this layer is 495 μs (5.9%) for receiving and 457 μs (5.6%) for sending. The blue area is the time spent in the two (sender, and receiver) 802.15.4 radio chips (AT86RF2XX). We can see that the total avg 8.1 ms latency is well-below requirements, and although experiments are with 10 msg/second, measured values yield theoretic limits one order of magnitude above (ideally - but there would be other problems like wireless collisions). We can see also that most of the time is spent in the radio transmission and then NDN crypto; an average of only 2.27 ms is spent in the non-crypto purely software parts of RIOT-ROS2.

VI. Conclusion

In this article, we have detailed the design, implementation and evaluation, of a generic system for minibots (ROS2 on top of RIOT, using NDN-based communication). We have detailed how the most important communications primitives of ROS2 are mapped on NDN. Then we have actually implemented the Publish/Subscribe primitive, and tested it on microcontrollers through micro-benchmarking. Results show that RIOT-ROS2 Publish/Subscribe can fit on microcontrollers as low as 80kB Flash and 10Kb RAM - and have been actually tested on 256kB Flash and 32kB RAM Board.

Network performance results of RIOT-ROS2 Publish/Subscribe evidence an average end-to-end latency of 8.2 ms over 802.15.4 for 20 bytes messages, with most of it, 4.7 ms (58%), being taken by the radio. Furthermore, the data signature, compulsory in NDN is taking 90% of the RMW layer, and could be improved by using specific crypto hardware, or skipping this stage for microcontrollers. The latency of our system is already excellent (e.g. would be extrapolated to a theoretical throughput of $>$ 100 messages per second under ideal conditions), and fully fits the requirements of minibots. Moreover, it could further be improved by an order of magnitude by proper crypto and wireless layers, enabling even more complex control scenarios (lower-level remote control and/or cloud-based).

References

[1] C. Bormann, M. Ersue, and A. Keranen, "Terminology for Constrained-Node Networks," IETF, RFC 7228, May 2014.

[2] F. Kaup, P. Gottschling, and D. Hausheer, "Powerpi: Measuring and modeling the power consumption of the raspberry pi," in *LCN2014*.

[3] C. Anderson, "DIY Drones," 2018, available at https://diydrones.com.

[4] M. Le Goc, L. H. Kim, A. Parsaei, J.-D. Fekete, P. Dragicevic, and S. Follmer, "Zooids: Building blocks for swarm user interfaces," in *User Interface Software and Technology Symposium*. ACM, 2016.

[5] A. Udanis, "Cheerson cx-10 micro drone teardown," available at https://www.allaboutcircuits.com/news/teardown-tuesday-micro-drone/, 2018.

[6] G. Passault, Q. Rouxel, F. Petit, and O. Ly, "Metabot: a low-cost legged robotics platform for education," in *ICARSC 2016*. IEEE, 2016.

[7] M. Q. et al., "Ros: an open-source robot operating system," *Proceedings of ICRA Workshop on Open Source Software*, 2009.

[8] "The ArduPilot Unmanned Vehicle Autopilot Software Suite," available at http://ardupilot.org, 2018.

[9] V. Mayoral *et al.*, "Towards an Open Source Linux Autopilot for Drones," in *LibreCon, 2014*, 2014.

[10] T. Zhao and H. Jiang, "Landing system for AR. Drone 2.0 using onboard camera and ROS," in *Guidance, Navigation and Control Conference (CGNCC), 2016 IEEE Chinese*. IEEE, 2016, pp. 1098–1102.

[11] O. Hahm *et al.*, "Operating Systems for Low-End Devices in the Internet of Things: a Survey," *IEEE Internet of Things Journal*, 2016.

[12] E. Baccelli *et al.*, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE IoT Journal*, 2018 (to appear).

[13] R. Barry. FreeRTOS. Available at http://www.freertos.org, 2018.

[14] "NuttX Real-Time Operating System," available at http://nuttx.org, 2018.

[15] OMG, "The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol, Version 2.2," 2014.

[16] "FreeRTPS," available at https://github.com/ros2/freertps, 2015.

[17] L. Dauphin, E. Baccelli, C. Adjih, and H. Petersen, "NDN-based IoT robotics," in *ACM/ICN conference*. ACM, 2017, pp. 212–213.

[18] Atmel, "Atmel SAM R21 Xplained Pro - User Guide."

[19] B. Gerkey, "Why ROS 2.0?" available at http://design.ros2.org/articles/why_ros2.html, 2015.

[20] H. Project, "micro-ROS: Platform for seamless integration of resource constrained devices in the robot ecosystem," available at https://github.com/microROS/micro-ROS, 2018.

[21] L. Y. Sørensen, L. T. Jacobsen, and J. P. Hansen, "Low cost and flexible uav deployment of sensors," *Sensors*, vol. 17, no. 1, p. 154, 2017.

[22] V. Mayoral, "Towards ROS-native Drones," available at https://medium.com/@vmayoral/towards-ros-native-drones-2-edcfc37cc18f, 2016.

[23] H. Petersen, C. Adjih, O. Hahm, and E. Baccelli, "Iot meets robotics-first steps, riot car, and perspectives," in *ACM International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2016.

[24] L. Dauphin, H. Petersen, C. Adjih, and E. Baccelli, "Low-cost robots in the internet of things: Hardware, software & communication aspects," in *NextMote Workshop-? Next Generation Platforms for the Cyber-Physical Internet*, 2017.

[25] C. B. Schindler, T. Watteyne, X. Vilajosana, and K. S. Pister, "Implementation and characterization of a multi-hop 6tisch network for experimental feedback control of an inverted pendulum," in *WiOpt*. IEEE, 2017, pp. 1–8.

[26] Z. Sheng, S. Yang, Y. Yu, A. V. Vasilakos, J. A. McCann, and K. K. Leung, "A survey on the ietf protocol suite for the internet of things: Standards, challenges, and opportunities," *Wireless Communications, IEEE*, vol. 20, no. 6, pp. 91–98, 2013.

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

[28] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named Data Networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

[29] W. S. et al., "The design and implementation of the ndn protocol stack for riot-os," *IEEE Globecom Workshops*, 2016.

[30] J. Chen, M. Arumaithurai, L. Jiao, X. Fu, and K. Ramakrishnan, "COPSS: An efficient content oriented publish/subscribe system," in *ACM/IEEE ANCS*. IEEE, 2011, pp. 99–110.

[31] P. Gusev and J. Burke, "NDN-RTC: Real-time videoconferencing over named data networking," in *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. ACM, 2015, pp. 117–126.

[32] W. Shang, A. Afanasyev, and L. Zhang, "Vectorsync: distributed dataset synchronization over named data networking," in *Proceedings of the 4th ACM ICN conference*. ACM, 2017, pp. 192–193.

[33] A. Z. Hindi, M. Kieffer, C. Adjih, and C. Weidmann, "Ndn synchronization: iroundsync, an improved roundsync," in *Proceedings of the 4th ACM ICN Conference*. ACM, 2017, pp. 194–195.

[34] W. Shang, Y. Yu, L. Wang, A. Afanasyev, and L. Zhang, "A survey of distributed dataset synchronization in named data networking," Technical Report NDN-0053, NDN, Tech. Rep., 2017.

[35] E. B. et al., "Information centric networking in the iot: Experiments with ndn in the wild," *ACM ICN*, 2014.

[36] C. Bormann and P. Hoffman, "Rfc 7049, concise binary object representation (cbor)," 2013.

[37] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*. IEEE, 2003, pp. 200–206.

[38] "Data distribution service (dds) specification, version 1.4," available at https://www.omg.org/spec/DDS/, 2015.

[39] V. Mayoral, "ros2 embedded nuttx," available at https://github.com/ros2/ros2_embedded_nuttx, 2014.

[40] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, L. Yanbiao, S. Mastorakis, Y. Huang *et al.*, "Ndn technical report: Nfd developer's guide - report ndn-0021, revision 8," 2018.