A Novel IoT Protocol Architecture: Efficiency Through Data and Functionality Sharing Across Layers

Vinícius Galvão Guimarães, Adolfo Bauchspiess
Departamento de Engenharia Elétrica
Universidade de Brasília
Brasília, DF 70910-900, Brazil
vinicius_galvao@msn.com, adolfobs@unb.br

Renato Mariz de Moraes
Centro de Informática
Universidade Federal de Pernambuco
Recife, PE 50740-560, Brazil
renatomdm@cin.ufpe.br

Katia Obraczka
Department of Computer Engineering
University of California (UCSC)
Santa Cruz, CA 95064, USA
katia@soe.ucsc.edu

***Abstract*—Motivated by the need to accommodate IoT devices that have limited power, processing, storage, and communication capabilities, this work introduces the IoT Unified Services, or IoTUS, a novel network protocol architecture that targets energy efficiency and compact memory footprint. IoTUS uses an extensible *service layer* that facilitates cross-layer sharing. It promotes sharing of both network control information, (e.g., number of transmissions, receptions, collisions at the data-link layer) and functionality (e.g., neighbor discovery, aggregation) by different layers of the protocol stack. Additionally, IoTUS can be used by existing network stacks without having to modify the basic operation of their protocols. We implemented IoTUS on the Cooja-Contiki network simulator/emulator. Our experimental results show improved energy efficiency resulting in longer network lifetime, as well as more compact memory footprint resulting when compared to current IoT protocol architectures.**

***Index Terms*—Energy Efficiency, Internet of Things, Stack Framework, Wireless Networks.**

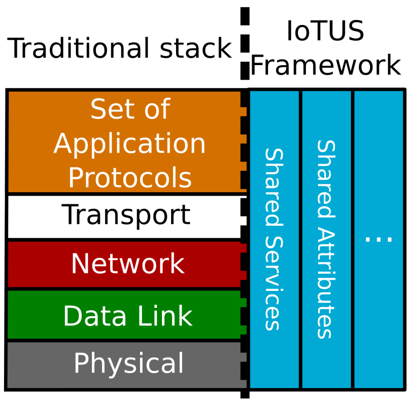

Fig. 1. IoTUS' extensible cross-layer sharing framework.

I. Introduction

Similarly to the ISO/OSI (Open System Interconnection) communication standard [1], the design of the Internet's TCP/IP protocol architecture also followed the principles of *layered system design*. As such, the functions performed by the TCP/IP protocol suite are implemented at different protocol layers, where each layer provides a specific set of services to the layer above through a well-defined interface. Using this interface, data being received or sent is passed up or down the stack on its way from source to destination. Accessing services provided by layer i through a well-defined interface shields layer $i+1$ from the implementation details of layer i, simplifying each layer's design and improving overall modularity, maintainability, and extensibility. However, layered design approaches can increase overhead, as each layer incurs additional communication- (e.g., additional header fields) and processing costs. Furthermore, limiting the flow between layers restricts sharing of functionality across layers and may lead to functions being duplicated at different layers.

Motivated by the emergence of wireless networks, the networking research community devoted considerable attention to cross-layer approaches as a way to circumvent the limitations imposed by the traditional TCP-IP layered protocol architecture. Accordingly, a wide range of techniques that use cross-layer information aiming at improving performance were proposed [2], [3].

As we enter the Internet of Things (IoT) era and the number of connected devices with limited power, processing, storage, and communication capabilities rapidly increases, as well as Internet applications become increasingly more complex and resource demanding, significant advancements in network protocol architecture efficiency are imperative to deliver the IoT vision of a connected world.

As described in more detail in Section II, a number of approaches have tried to bridge this gap. Notable examples include: using an adaptation layer that translates/optimizes header fields of standard protocols so they can run on capability-challenged devices [4], proposing monolithic protocol stacks [5], and developing more effective ways for cross-layer information sharing, e.g., CLAMP [6] and Rime [7]. However, most proposals to-date have either developed modules that require protocol redesign or used traditional cross-layer data exchange.

In this paper we introduce the IoT Unified Services framework, or IoTUS for short, which takes a step further in achieving efficiency through a novel approach to cross-layer sharing, while still preserving the benefits of layering, such as modularity and portability. As shown in Fig. 1, IoTUS proposes an extensible *service layer* that facilitates cross-layer sharing of not only control plane information, or *attributes* (e.g., number of transmissions, receptions, collisions at the data-link layer) but also *services* (e.g., neighbor discovery, data aggregation). IoTUS can be used by existing protocol stacks allowing information and functionality sharing among layers without requiring changes to existing protocols. Our evaluation results show that IoTUS is able to achieve significant energy savings as well as lower memory footprint when compared to existing IoT stacks, in particular Rime [7].

The rest of this paper is structured as follows. Section II presents an overview of related work while Section III describes the IoTUS framework in detail. Our evaluation methodology and results are presented in Sections IV and V, respectively. Section VI concludes the paper and discusses directions for future work.

II. Related work

In this section, we present an overview of the current state-of-the-art of protocol stacks and architectures that try to accommodate devices with constrained power, computation, storage, and communication capabilities.

Through experiments using the Cooja-Contiki simulator [8], the work reported in [8] shows that the control overhead in RPL [9] can represent about 25% of the overall traffic in a 20-node network and can go up to 75% with 100 nodes. It is worth noting that IPv6 and RPL use similar control packets for neighbor discovery [10], which could generate additional control overhead.

The Flexible Interconnection Protocol (FLIP) [11] uses flexible headers to interconnect heterogeneous devices, while accommodating their different capabilities. The overhead incurred by FLIP's header will depend on the capabilities of the device running FLIP and the needed functionality. For low-power devices like scalar sensors (e.g., temperature, humidity, etc.), which typically need to send out their readings periodically, FLIP provides considerable savings when compared to fixed-length protocols like IPv4 and IPv6.

Other efforts towards efficient protocol architectures specifically targeting networks of low power, low capability devices (e.g., wireless sensor networks) have used a "monolithic" approach, i.e., combining the functionality of multiple layers into a single layer. A notable example is the Unified Cross-Layer module (XLM) [5] and Uniform Clustering with Low Energy Adaptive Hierarchy (UCLEAH) [12].

Adaptation-layer protocols like 6LoWPAN [4] have also been proposed as a way to translate protocol fields and optimize headers in order to be able to adapt standards such as IPv6 [13] and RPL [9] to run on top of IEEE 802.15.4 [14] in IoT devices. 6LoWPAN, IPv6, IEEE 802.15.4 and RPL can be considered the *de facto* protocols for the IoT stack, according to [15].

Other approaches try to improve IoT protocol stack efficiency through cross-layer information sharing. For example, Cross Layer Management Plane CLAMP [6] uses a publish/subscribe/update/query system that allows protocols at different layers to share information.

Another solution for cross-layer sharing is proposed by TinyXXL [16] which provides the TinyOS [17] embedded operating system with more efficient data storage, as well as a generic interface for data exchange. TinyXXL is part of TinyCubus [18], a new cross-layer based protocol stack for sensor networks.

Cross-layer information sharing in the Rime stack [7] which runs on ContikiOS [19] works as follows. Information produced by different protocol layers is stored as attribute-value pairs and is accessible by protocols at different layers. Shared information is used in building the protocol headers at the different layers. Rime uses a Radio-Duty Cycle (RDC) layer separate from the Medium Access Control (MAC) layer. This may cause portability issues as most implementations handle both layers (RDC and MAC) as a single layer [20].

It is interesting to point out that Riot-OS [21], a more recently developed operating system for IoT devices, can use different protocol stacks. The default stack is named GENERIC (GNRC) and does note make use of any cross-layer sharing.

IoTUS aims at facilitating sharing across protocol layers. In light of related work, IoTUS' main contributions include: (1) a systematic approach to cross-layer information sharing that yields both energy and storage efficiency as described in detail in Section III; and (2) a modular and extensible service layer that enables sharing common functionality by different protocol layers as illustrated in Fig. 2. Additionally, IoTUS can be used by existing network stacks without having to modify their protocols.

III. IoTUS

As previously discussed, IoTUS, promotes information and functionality sharing across protocol layers while maintaining the benefits of a layered design. Through efficient cross-layer sharing, IoTUS' main goal is to achieve energy efficiency as well as a more compact memory footprint, both of which are important to accommodate IoT devices with limited capabilities. Similar to other proposals ([6], [7], [16]), IoTUS provides modules to standardize the way information is stored and packets are built, allowing more effective information sharing among layers. As illustrated in Fig. 2, IoTUS is designed as a collection of service layers (or modules) which can be used by existing stacks without the need to modify their protocols' design.

IoTUs' main module is called *IoTUS-Core*. It is required both during compilation and runtime. For the compilation process, IoTUS-Core processes each protocol in the stack and includes IoTUS' modules as needed/requested.

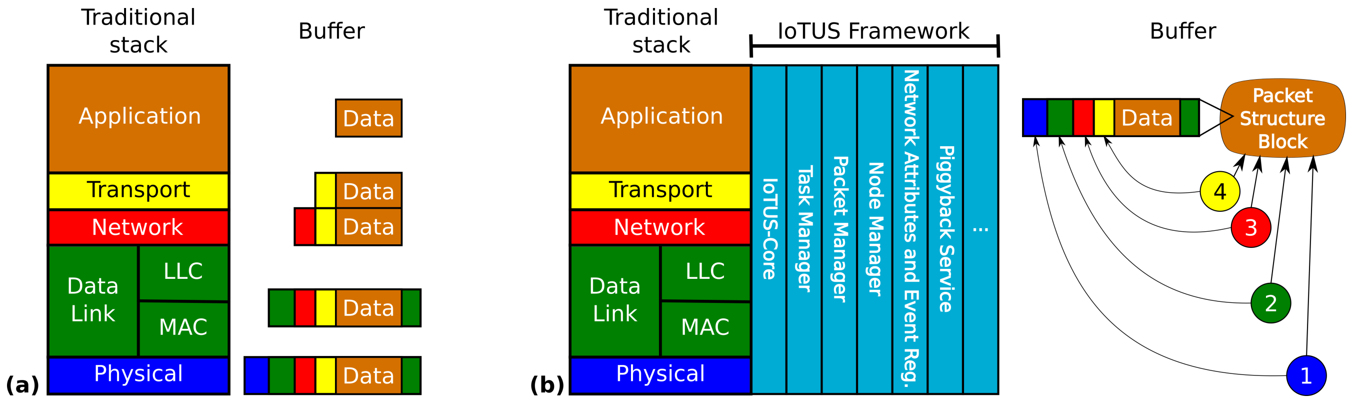

Fig. 2. (a) Traditional network stack; (b) IoTUS service modules as an extension to the stack.

IoTUS modules are similar to dependencies in Linux' Advanced Packaging Tool (APT) [22]. They can rely on other IoTUS services, i.e., services can be split into smaller services and protocols can select exactly which one(s) to use. This dependence management is also taken care by *IoTUS-Core*. At runtime, it starts every other module using the information gathered at the compilation stage.

Some modules are mandatory for the framework operation, like: *Node Manager*, *Task Manager*, and *Packet Manager*. *Node Manager* maintains information about the node's neighbors such as addresses, ranking in the routing tree, link layer sequence number, link quality (RSSI), etc. The *Packet Manager* module provides functions to build packets; these functions can be used by any layer of the protocol stack when they are adding, removing, or changing fields in their respective transmission units (e.g., packets at the network layer, frames at the data link layer, etc). The *Task Manager* assigns the control of a module to protocols, thus ensuring synchronization between procedures and protocols.

Other modules are optional and their utilization will depend on the protocol's functional needs. For example, the *Piggyback Service* module provides a data aggregation mechanism that can be used by protocols at different layers. Piggybacking helps achieve energy efficiency by ”packaging” as much information as possible into a packet. *Network Attributes and Event Register* module concentrates information about many general network values, e.g., number of transmissions, connection quality, package drop rate, and others.

To illustrate IoTUS' operation, let us consider an environmental monitoring application where nodes use a basic network protocol stack composed of a data link protocol, e.g., ContikiMAC [23], a static routing protocol, and the application layer protocol which sends periodic data from sensing nodes forming a tree rooted at the data sink. In addition to application-layer messages, keep-alive (KA) control messages are periodically generated by nodes to the data sink.

In most existing networks stack, packets are built based on an array of bytes, in which headers are added to the payload as packets are processed by each layer on their way down the stack. In IoTUS, A protocol uses information maintained by *Node Manager* to build a packet using the *Packet Manager*. During this step, additional information can also be added to the packet, such as timeout, priority, fragmentation/aggregation, etc. After the protocol finishes handling this packet, it sends a signal to the next layer carrying the packet's metadata. In this way, every field added to the packet has a globally known format, readable across different protocols. In the example shown in (Fig. 2), the application layer starts to build a packet and will signal the network layer when it is done. The network layer will evaluate the information already inserted in this packet block created by the application and inserts more processed data and header, sending a signal to the next layer as well.

Since the process of managing packets across layers is done using a centralized module, i.e., the *Packet Manager*, other modules such as *Piggyback Service* can use the outgoing packet to aggregate information from other layers. For example, in the case of an application message being built to be transmitted, a KA control packet can be piggybacked, which results in improve network efficiency.

As previously discussed, besides facilitating information sharing across layers, IoTUS also allows sharing of services, e.g., *Neighbor discovery*, *Tree Manager* services, which are responsible for discovering information about a node's neighbors and maintaining a routing tree, respectively.

The control of the sending packets is assigned by *Task Manager*, but other protocols aware of this module's operation can aggregate their requests, and therefore reduce overhead and/or improve connection speed.

IoTUS was developed to improve energy consumption by allowing protocols to synchronize and/or aggregate their procedures. Hence, with more protocols and more complex tasks, it is expected better memory usage, code reduction and network lifetime in general. However, IoTUS comes at the cost of processing time and an initial additional code. This processing cost caused by IoTUS framework can increase CPU consumption, but the energy saved in radio operations is expected to cause more impact on the overall network consumption.

The remainder of this section describes in detail all of

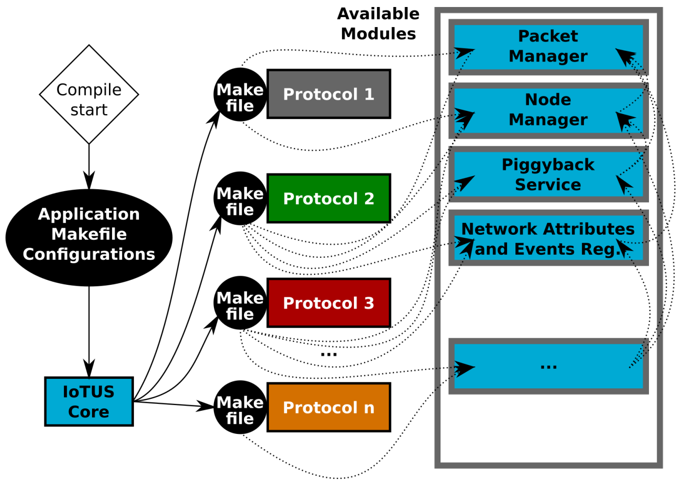

Fig. 3. Example of IoTUS-core installing requested modules from different protocols.

IoTUS' modules that have been currently implemented.

A. *IoTUS Core*

Its main function is given by the software compilation stage, where its compilation directives (here referenced as *makefiles*) manage which modules will be installed.

As shown in Fig. 3, at the beginning of a device operation, the framework core (*IoTUS-core*) interfaces different protocols. The application *makefile* should have some expected parameters to start and set how IoTUS operates. Then, this module's *makefile* continues the compilation, reading and installing the other necessary requested modules from each protocol *makefile*. This is a compilation step, which selects only the necessary codes to be installed, reducing the overall implementation size.

Each module also has its own *makefile*, which can request another module to be installed, creating the dependency system. As the protocols know exactly which module they will be using, the interface functions and API (Application Programming Interface) are known, making every information in that module to be understood by the operating protocol.

IoTUS Core is also responsible for starting all installed modules at runtime. It also provides functions to simplify procedures between mandatory modules, e.g., getting a neighbor's reference (by address) within *Node Manager* and using it to create a packets with *Packet Manager*.

B. *Node Manager*

Many information extracted by different layers can be attached to a given neighbor node; however, protocols usually would not share this data. This module centralizes the data gathered about neighbors. It creates a standardized way to share a determined set of information; thus, allowing shared neighbor data to be stored in structures of blocks, retrieving its block by reference pointers or search by address number.

For example, the link quality extracted by the physical layer can be attached to the node structure block, along with the address given by the second layer and the rank (how distant, in number of hops, the node is from a sink node) given by the third layer. Such approach reduces memory usage, redundancy and improves cross-layer decisions.

C. *Packet Manager*

Similar to the *Node Manager*, *Packet Manager* is responsible to concentrate most of packet information into one structure, as represented by the small circles being added to the big packet block in Fig. 2(b). That means having shared data in one place, where every layer can understand what is being added to the header. Moreover, *Packet Manager* works on top of the *Node Manager* and saves memory space by pointing to its structure block instead of copying all of it.

Many packet parameters like source and destination addresses are available in a standardized manner. Also, this information is available across the layers when the packet is being built. However, differently from the static way Rime/ContikiOS allocates the whole collection of possible data, IoTUS framework does it dynamically, allocating only the necessary memory for the information attached through linked lists.

Along with the *Node Manager*, the packet references their source and destination nodes using this new system; thus, creating an integration that facilitates other services to operate, like aggregation of packets.

Fig. 2 shows a side-by-side comparison between a traditional layered stack and the same stack extended with the IoTUS framework by showing how an application message is processed before being transmitted. In the traditional way Fig. 2(a), messages are sent down the stack using abstract functions and encapsulating headers in the buffer. In Fig. 2(b), the same stack with IoTUS framework builds the message using dynamic packet structure blocks. The *Packet Manager* creates a buffer that can hold headers and small information block attached to it. Each small block contains well defined and known information, the packet fields.

Hence, after the application layer signals the messaging procedure in the shared layer, the packet structure block (buffered) is reserved and a signal is sent to the lower layer. Layers below will set small structure blocks and attach to the packet structure block along with their headers. This process is represented by the small circles numbered with the layer rank (4 to 1). At the physical layer, all information attached is readable and the header is ready.

The creation of a packet container in the IoTUS system can be seen with more details in Fig. 4, where step 1 represents a calling from a protocol to the message creating function, defining some basic information (payload, destination node, parameters, timeout and others). Continuing, lower layers will get information (step 2) to process this packet. In this way, packets are always stored in a list of dynamic containers that all layers have access. IoTUS' services generally use pointer references to other services block structures, which allows to keep information up to date.

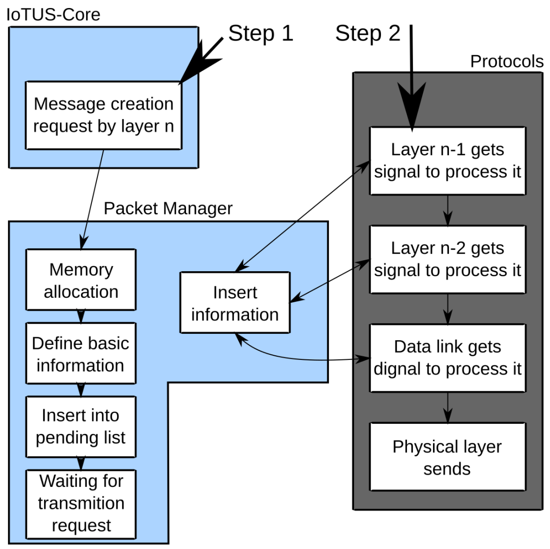
IoTUS-Core
Step 1
Step 2
Protocols
Message creation request by layer n
Layer n-1 gets signal to process it
Packet Manager
Layer n-2 gets signal to process it
Memory allocation
Insert information
Define basic information
Data link gets dignal to process it
Insert into pending list
Physical layer sends
Waiting for transmition request

Fig. 4. Message construction with IoTUS.

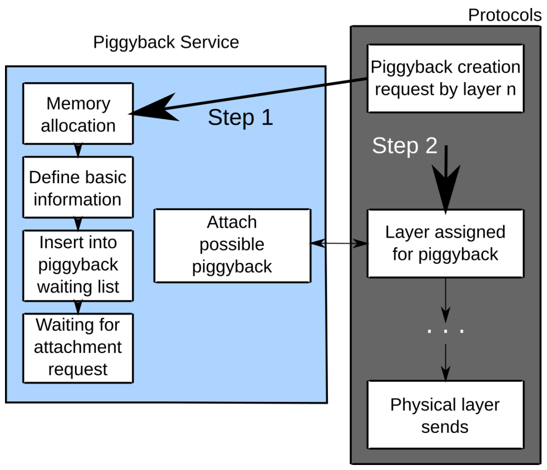
Protocols
Piggyback Service
Piggyback creation request by layer n
Memory allocation
Step 1
Step 2
Define basic information
Attach possible piggyback
Insert into piggyback waiting list
Layer assigned for piggyback
Waiting for attachment request
. . .
Physical layer sends

Fig. 5. Using IoTUS Piggyback Service.

D. Task Manager

This manager module assigns tasks to each running protocol. Since IoTUS framework proposes shared services and functionality, it would be possible that two or more protocols using the same service would then request it to start a procedure more than once. Hence, synchronization is necessary. However, it does not block any protocol to do a redundant operation by itself, instead it just informs all layers which service will be done by each protocol, keeping the redundancy, if necessary.

The process of assigning for a task is done at the start-up of the device by each protocol using this framework. *Task Manager* module has a subscription system that allows each protocol to check which one is finally responsible for each task. Inside this module, priorities are usually given to the protocols located in lower layers, i.e., protocols in physical and data link layers have higher priority than network and application layers. This also solves some issues with addressing, packet aggregation and other tasks.

In the case of addressing, most of the data link protocols have their own methods to look for neighbor's address. Thus, they already have to use the header to insert these addresses. A network layer that checks that the data link layer is already addressing can use this feature to synthesize their addressing system accordingly, which is done in a similar way by the 6LoWPan protocol [4].

Another case is given by the aggregation service. In many cases it is done by the network layer, but if the data link layer has some procedures that would benefit from aggregating their control packets, then it could request to operate the shared service called aggregation.

E. Piggyback Service

IoTUS' provides data aggregation through its *Piggyback Service* module, which is critical to optimize energy consumption by reducing network overhead. Protocols can create piggyback blocks (pieces) and set defined parameters like timeout, destination address and others. Furthermore, if timeout expires, *Piggyback Service* will signal the callback function of the block owner, so that the protocol sends that data as soon as possible. However, if conditions are matched, the piggyback container will be aggregated (with small compact headers) into the outgoing packet.

Therefore, *Piggyback Service* uses *Node Manager* and *Packet Manager*. The main conditions to insert a piggyback block into a packet are:

- There is a outgoing packet;
- The packet has to be flagged as allowing aggregation by its creator protocol;
- The packet is addressed to the same next router(s). (Eventually, to the same final destination node).

This service creates control frames overhead that avoid separate transmissions to shared destinations in the network. Thus, many layers can rely on it to have their control packet optimized, as is the case of DAO packet in RPL.

The process of creating a piggyback block is given in Fig. 5. Protocols would start by a similar process as the *packet Manager*, allocating resources with specific functions of the module (Step 1 of Fig. 5). Later, if any protocol was previously assigned by the *Task Manager*, it will try to attach possible piggyback blocks into a packet being built, aggregating its information (Step 2 of Fig. 5).

From another point of view, Fig. 6 shows the piggyback piece (P.B.p) being created by "Protocol y", which represents the Step 1 in Fig. 5. When "Protocol x" creates a packet, headers (Hdr) are attached. At some assigned layer, *Piggyback Service* is called and it attaches the pieces that matches this condition, representing the Step 2 in Fig. 5. Bellow the dashed line, this process is represented by a node using this procedure and delivering his packet to its destination, in which the piggyback piece is detached again and delivered to the target node's protocol.

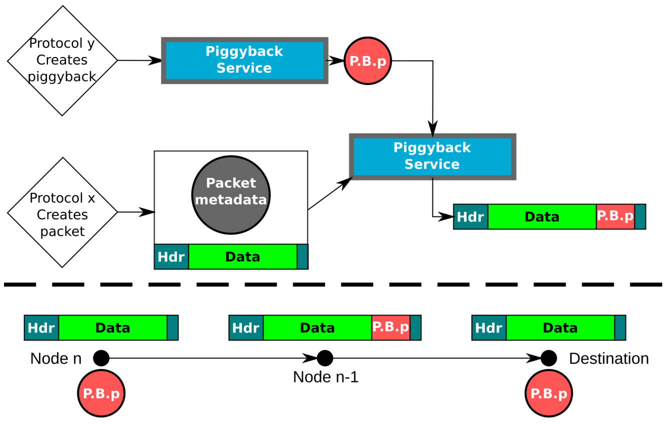

Fig. 6. Piggyback aggregation and transmission.

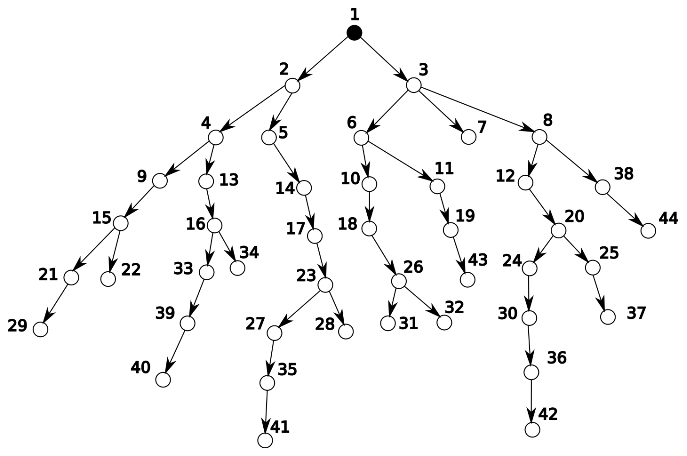

Fig. 7. Tree topology considered for an environmental monitoring.

IV. Experimental Methodology

We evaluated IoTUS using the Cooja-ContikiOS [8], [19] simulation/emulation platform. We used Cooja for the following reasons: first, it provides an experimental platform specifically designed for networks of capability-constrained wireless devices (e.g., wireless sensor networks, IoT) where we can run controlled and reproducible experiments; as such, it includes an implementation of the Rime protocol stack [7], which we used as basis of comparison against IoTUS; additionally, the same code developed to run on Cooja-ContikiOS can be directly ported to real devices running ContikiOS, as well as testbeds which are accessible to the research community. In particular, we developed IoTUS under ContikiOS [19] for the TMote Sky [24] device emulated within Cooja

To drive our experiments, we consider a general environmental monitoring application in which sensing nodes are deployed to periodically sense the environment (e.g., temperature, humidity, etc.) and transmit their readings through the network to a data sink. Fig. 7 shows a 44-node tree topology that we used in our experiments. The tree is rooted at the data sink (node 1) and intermediate and leaf nodes are sensing nodes. Note that intermediate tree nodes act both as traffic generators as well as forwarders.

Table I shows the simulation parameters and their corresponding values used in our experiments. The values used for *sensing rate* and *application payload size* have been previously used in the literature (e.g., [25]). For the keep alive messages, which, as described in Section III, are control messages periodically generated by the network layer that are transmitted through the tree towards the root, we set their size and frequency based on ContikiOS' implementation of RPL.

TMote Sky's power consumption specification is summarized in Table II. Note that Cooja's emulation of Tmote Sky provides four different power modes for the radio, namely *Reception*, *Transmission*, *Idle*, and *Sleep*, and two for the micro-controller, *Active*, and *Sleep*. Energy consumption measurements were provided by two different Cooja-ContikiOS tools, namely: *PowerTrace* and *PowerTracker*. *PowerTrace* is a tool available inside the ContikiOS implementation, which periodically reports energy consumption information through its serial port. *PowerTrace* reports time spent in each state (transmitting, receiving, idle, or sleep for the radio, and active or idle for CPU). *PowerTracker* is available in the Cooja simulator and also provides power consumption measurements for the radio. However, due to its microsecond simulation granularity, it provides more accurate power consumption measurements than *PowerTrace*. As such, we use *PowerTracker* to measure energy consumed by the radio while CPU consumption is still extracted from the *PowerTrace* tool.

TABLE I
Simulation Parameters.

Parameter	Value
Sensing rate (application packets)	30 seconds
Application payload size	20 bytes
Keep alive control data size	12 bytes
Keep alive transmission rate	30 seconds

V. Results

We evaluated IoTUS in terms of its energy efficiency and memory footprint. As baseline, we use ContikiOS' Rime stack [7] implementation. Results reported here were obtained by averaging over 10 runs using random seeds with a 95% confidence interval.

While latency is an important performance metric for delay-sensitive applications, it is not considered to be critical for environmental monitoring. Though we do not report latency re-

TABLE II
Energy consumption by states of Tmote Sky [24].

State	Current
micro-controller Active (No Radio)	1.8 mA @1MHz,3V
micro-controller sleep (No Radio)	5.4 µA
Reception	18.8 mA
Transmission	17.4 mA (0 dBm)
Idle	18.8 mA
Sleep	0.426 µA

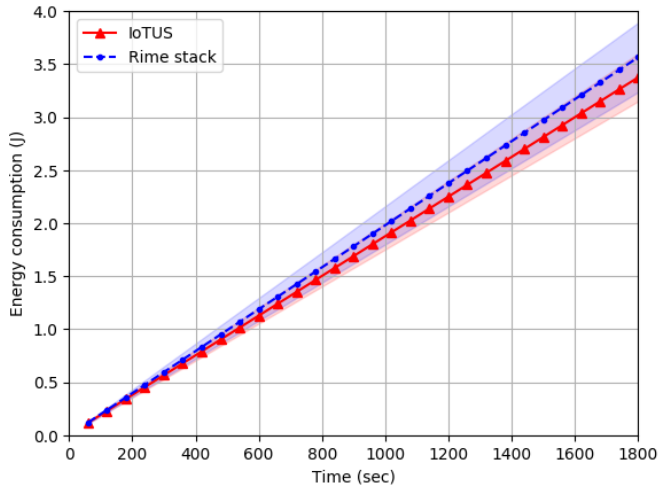

Fig. 8. Average energy consumption per node in a 44-node tree network.

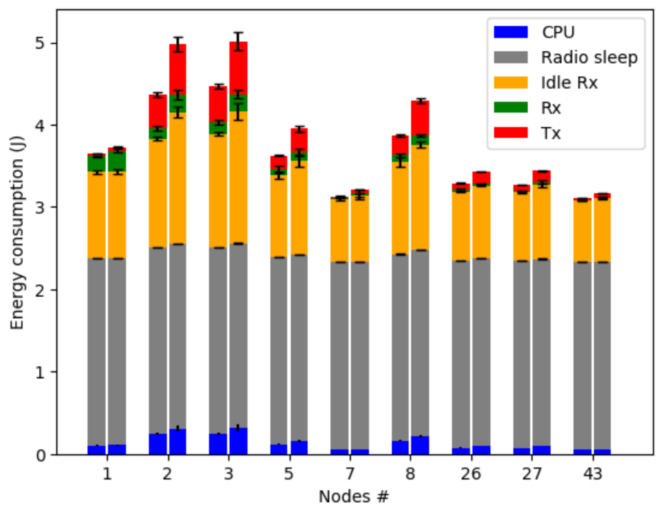

Fig. 9. Total consumption by states of selected nodes in the 44 nodes network. The left side bar of each node describes the IoTUS framework consumption. The right side bar indicates the consumption of the node for the Rime stack.

sults in this paper, we note that additional processing incurred by IoTUS (e.g., packet construction, data aggregation) may increase latency. For example, the duration between building a message at application layer and finally transmitting it was on average 4.1ms with IoTUS, while the standard Rime stack processed the same request in 1.3ms. Depending on the end-to-end propagation delay, this difference may be negligible. As part of our ongoing work, we have been evaluating IoTUS impact on latency.

Fig. 8 plots energy consumption averaged over all 44 network nodes using the topology illustrated in Fig. 7 for both IoTUS and Rime over time. The shaded areas represent the confidence interval of each line according to their colors. We ran the experiments for 30 minutes to allow enough time for the system to reach steady state. Although IoTUS' average energy consumption gain is about 5.33%, as shown in Fig. 9, nodes 2 and 3 experience energy consumption gains of 13% each. Note that these are the closest nodes to the root of the tree and, thus, are the ones that need to forward the highest number of packets on their way to the sink. Indeed, Fig. 9, which plots consumption by power state for selected nodes in the network, shows that most of nodes' 2 and 3 energy consumption gains by IoTUS come from the radio, more specifically by spending less time in transmission mode when compared to the Rime stack. This is mainly due to IoTUS' aggregation feature provided by the *Packet Manager* and *Piggyback Service*. For comparison purposes, node 43, which is a leaf node had the overall minimum consumption. Fig. 9 also shows energy consumption for other nodes which are a mix of intermediate and leaf nodes.

As expected, for this experiment, radio functions are by far the most energy consuming. Even though active states can consume thousands more than sleep state and up to 10 times more than CPU, Fig. 9 reveals that nodes' radios spent most of the time in either idle or sleep, and these two states contributed the most to the overall energy consumed by nodes.

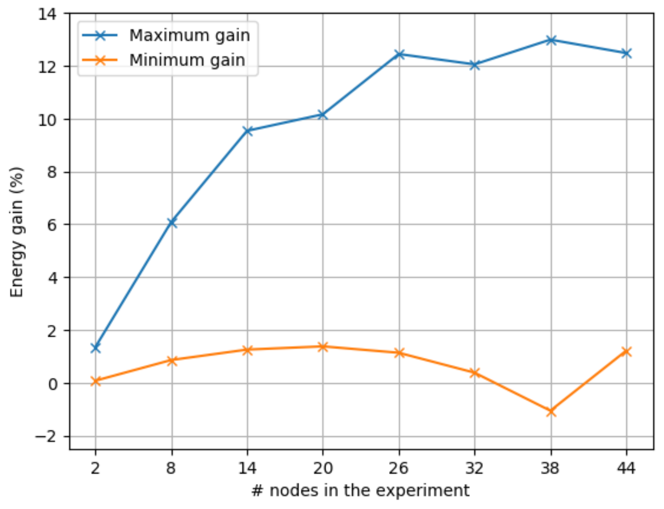

Fig. 10. Maximum and minimum radio energy gain per nodes in the network.

Fig. 10 shows the maximum and minimum energy consumption gains attained for networks of varying size. These results were obtained by using trees of sizes 2, 8, 14, 20 nodes, etc. where each tree corresponds to the tree shown in Fig. 7 but only including nodes up to node id 2, 8, 14, 20, etc., respectively. We observe that, as the network increases in size, so does the energy consumption gains obtained using IoTUS. As previously discussed, the highest gains were observed by nodes 2 and 3, which also had the highest energy consumption according to Fig. 9. Since these nodes are the most likely to be the first to have their battery depleted, guaranteeing them the highest energy savings results in extending the network's overall lifetime.

Fig. 11 presents results for expected node lifetime for tree topologies of varying sizes. Node lifetime is defined as the

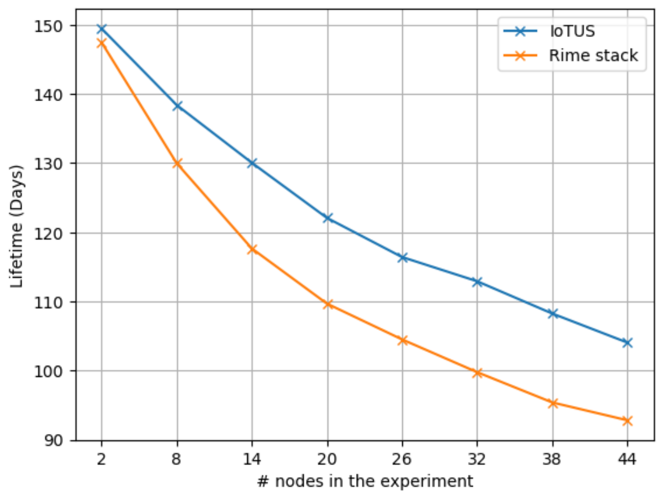

Fig. 11. Maximum lifetime per number of nodes in the network.

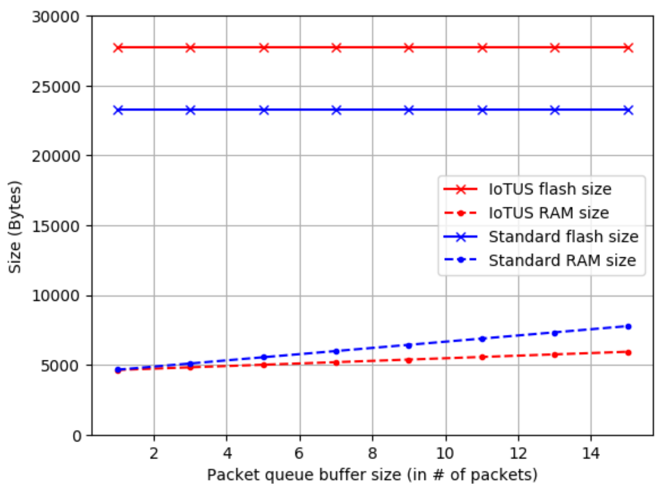

Fig. 12. Memory usage when increasing packet buffer capacity.

time between the start of the experiment until the node's battery is depleted. To calculate lifetime, we consider that each device is supplied with two AA cells (2000mAh of NiMH type) that would provide up to 22.32 kJ at 3.1 Volts according to [26]. A node's lifetime is then obtained by dividing the node's battery capacity by the node's average power consumption. Considering nodes 2 and 3 which exhibit the highest energy consumption, IoTUS achieves a lifetime gain of up to 11 days representing an improvement to 12.11% over Rime. Furthermore, for this particular topology, once nodes 2 and 3 deplete their batteries, the rest of the network gets disconnected from the sink and therefore is no longer able to perform its monitoring task.

We should point out that, with the current setup, we were not able to run experiments using trees with more than 44 nodes. The problem is due to RAM size needed by the Rime stack which exceeded the 10KByte TMote Sky RAM capacity. However, using IoTUS, larger tree topologies could still be emulated which attests to IoTUS' efficient memory footprint.

Fig. 12 shows a comparative memory footprint characterization between IoTUS and Rime as packet buffer size increases. While IoTUS requires additional flash space (less than 5KBytes, or 18% more than Rime), it saves RAM storage through its ability to share information across layers and avoid information duplication. As shown in Fig. 12, IoTUS' memory footprint savings increases with the size of the packet buffer. In this experiment, memory saving reaches 23.63% for a packet buffer size of 15 packets.

VI. Conclusion

In this paper, a new framework called Internet of Things Unified Services, or IoTUS, was introduced. Its main goal is to facilitate sharing across protocol layers while preserving the benefits of layered protocol architectures, in particular modularity and portability. To this end, IoTUS proposes an extensible *service layer*, that allows sharing of control plane information (e.g., collisions at the data-link layer, number of transmissions/receptions, radio packet size, ID address size) as well as sharing of services (e.g., neighbor discovery, network events log, data aggregation). Additionally, IoTUS can be used by existing network stacks without having to modify the basic operation of their protocols.

We implemented IoTUS on ContikiOS and evaluated its performance using ContikiOS' Cooja network simulator/emulator. Our results demonstrate that IoTUS is able to achieve energy efficiency as well as more compact memory footprint when compared to Rime, a layered stack used by ContikiOS.

Directions for future work include implementing protocol standards like RPL using IoTUS, deploying and evaluating IoTUS in real-world testbeds, as well as evaluating IoTUS latency performance.

Acknowledgment

This work was supported in part by CAPES Brazil, by CNPq Brazil, and by the US National Science Foundation under grant CNS 1321151.

References

[1] Webopedia, "The 7 layers of the OSI Model." [Online]. Available: https://www.webopedia.com/quick_ref/OSI_Layers.asp

[2] B. Fu, Y. Xiao, H. J. Deng, and H. Zeng, "A survey of cross-layer designs in wireless networks," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1, pp. 110–126, 2014.

[3] L. D. Mendes and J. J. Rodrigues, "A survey on cross-layer solutions for wireless sensor networks," *Journal of Network and Computer Applications*, vol. 34, no. 2, pp. 523–534, 2011.

[4] S. Chakrabarti, G. Montenegro, R. Droms, and J. Woodyatt, "Ipv6 over low-power wireless personal area network (6lowpan) esc dispatch code points and guidelines," February 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8066

[5] I. Akyildiz, M. Vuran, and O. Akan, "A Cross-Layer Protocol for Wireless Sensor Networks," in *Proceedings of the 2006 40th Annual Conference on Information Sciences and Systems*. Princeton, NJ, USA: IEEE, mar 2006, pp. 1102–1107.

[6] S. A. Madani, S. Mahlknecht, and J. Glaser, “A Step towards Standardization of Wireless Sensor Networks: A Layered Protocol Architecture Perspective,” in *Proceedings of the 2007 International Conference on Sensor Technologies and Applications (SENSORCOMM 2007)*. Valencia, Spain: IEEE, oct 2007, pp. 82–87.

[7] A. Dunkels, F. Österlind, and Z. He, “An adaptive communication architecture for wireless sensor networks,” in *Proceedings of the 5th international conference on Embedded networked sensor systems - SenSys ’07*. Sydney, Australia: ACM Press, 2007, pp. 335–349.

[8] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, “Cross-level sensor network simulation with COOJA,” in *Proceedings of the Conference on Local Computer Networks, LCN*. IEEE, 2006, pp. 641–648.

[9] T. Winter, Ed., P. Thubert, Ed., A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, “Rpl: Ipv6 routing protocol for low-power and lossy networks,” March 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6550

[10] A. Parasuram, D. Culler, and R. Katz, “An Analysis of the RPL Routing Standard for Low Power and Lossy Networks,” *Technical Report No. UCB/EECS-2016-106*, p. 98, 2016.

[11] I. Solis and K. Obraczka, “FLIP: A Flexible Interconnection Protocol for heterogeneous internetworking,” *Mobile Networks and Applications*, vol. 9, no. 4, pp. 347–361, 2004.

[12] K. Babber and R. Randhawa, “A Cross-Layer Optimization Framework for Energy Efficiency in Wireless Sensor Networks,” *Wireless Sensor Network*, vol. 09, no. 06, pp. 189–203, 2017.

[13] S. Deering and R. Hinden, “Internet protocol, version 6 (ipv6) specification,” December 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2460

[14] IEEE, “IEEE Standard for Low-Rate Wireless Networks,” *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, apr 2016.

[15] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, “Standardized protocol stack for the internet of (important) things,” *IEEE Communications Surveys and Tutorials*, vol. 15, no. 3, pp. 1389–1406, 2013.

[16] A. Lachenmann, P. J. Marrón, D. Minder, M. Gauger, O. Saukh, and K. Rothermel, “TinyXXL: Language and runtime support for cross-layer interactions,” in *Proceedings of the 2006 3rd Annual IEEE Communications Society on Sensor and Adhoc Communications and Networks, Secon 2006*, vol. 1, Reston, VA, USA, 2007, pp. 178–187.

[17] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and Others, “TinyOS: An operating system for sensor networks,” in *Weber W., Rabaey J.M., Aarts E. (eds) Ambient intelligence*. Berlin, Heidelberg: Springer, 2005, pp. 115–148.

[18] P. J. Marrón, D. Minder, A. Lachenmann, and K. Rothermel, “TinyCubus: An Adaptive Cross-Layer Framework for Sensor Networks (TinyCubus: Ein Adaptives Cross-Layer Framework für Sensornetze),” *it - Information Technology*, vol. 47, no. 2, jan 2005.

[19] Contiki, “The Open Source OS for the Internet of Things.” [Online]. Available: http://www.contiki-os.org/

[20] K. Roussel and Y.-q. Song, “A critical analysis of Contiki’s network stack for integrating new MAC protocols,” Ph.D. dissertation, INRIA Nancy, 2015.

[21] Riot, “The friendly Operating System for the Internet of Things.” [Online]. Available: https://riot-os.org/

[22] Wikipedia, “Advanced Packaging Tool.” [Online]. Available: https://pt.wikipedia.org/wiki/Advanced_Packaging_Tool

[23] A. Dunkels, “The contikimac radio duty cycling protocol,” *SICS Technical Report T2011:13*, 2011.

[24] Moteiv, “Tmote sky - Low Power Wireless Sensor Module.” [Online]. Available: http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf

[25] F. Osterlind, E. Pramsten, D. Roberthson, J. Eriksson, N. Finne, and T. Voigt, “Integrating building automation systems and wireless sensor networks,” in *Proceedings of the 2007 IEEE Conference on Emerging Technologies & Factory Automation (EFTA 2007)*, Patras, Greece, 2007, pp. 1376–1379.

[26] Wikipedia, “AA battery.” [Online]. Available: https://en.wikipedia.org/wiki/AA_battery