Dynamic Reconfiguration of Network Protocols for Constrained Internet-of-Things Devices

Peter Ruckebusch(✉), Jo Van Damme, Eli De Poorter, and Ingrid Moerman

Ghent University, iGent Tower, Technologiepark 15 (Zwijnaarde),
9052 Ghent, Belgium
peter.ruckebusch@intec.ugent.be

Abstract. The Internet-of-Things paradigm shifts the focus of sensor networks from simple monitoring to more dynamic networking scenarios where the nodes need to adapt to changing requirements and conditions. For this purpose many configuration options are added to the network protocols. Today, however, they can only be modified at compile-time, which seriously limits the ability to adapt the behaviour of the network.

To overcome this, a solution is proposed that allows reconfiguring the entire network stack remotely using CoAP. The Contiki implementation shows that for a small memory overhead (1.2 kB) up to 57 configuration parameters can be reconfigured dynamically. The average latency for reconfiguring one parameter in a twenty node network is only three seconds. A simple case-study illustrates how the energy consumption of an application can be reduced with (50 %) by dynamically fine-tuning the MAC duty-cycle.

Keywords: Internet-of-things · CoAP · Contiki · Dynamic reconfiguration · Wireless sensor networks · Network management

1 Introduction

The Internet of Things (IoT) philosophy [4] announces the third wave of digitalisation. After the rise of the PC (a computer in every home) and the smartphone (a mobile computer for every person) also appliances (or things) will be equipped with a mini-computer and communication interface in the near future. When connected to the Internet, they become IoT devices and enable to further digitise certain aspects of modern day society. In first instance, these devices will automatise or optimise certain processes but there are endless new application possibilities in areas such as healthcare, surveillance, agriculture, personal fitness, home automation and many more.

The vast majority of IoT devices are constrained end-devices (a.k.a. sensors and actuators) with limited computing power and memory. For these type of devices, specialized software (e.g. operating systems (OS), network protocols, etc.) is required. Given the wide range of possible applications and the specific limitations, much effort was spent by a broad research community to develop this software. Custom OSs such as Contiki, RIOT and TinyOS were proposed

B. Mandler et al. (Eds.): IoT 360° 2015, Part II, LNICST 170, pp. 269–281, 2016.
DOI: 10.1007/978-3-319-47075-7_31

that specifically target such devices. Also many standards emerged, providing answers to the specific challenges posed by these applications and devices.

Despite the societal and business potential, the uptake by industry of innovative large-scale sensor/actuator applications is slow. One of the major hurdles that retains innovation is the lack of built-in support for maintenance of such devices. Perhaps the most important aspect in maintainability is the possibility to dynamically reconfigure the network stack. Although the standards provide many options to configure network protocols, it is almost impossible to change the configuration settings at run-time. This implies that the entire network needs to reconfigured off-line if changes are required. In research this is not a major problem since experiments can be repeated using different settings. In real-life, however, application requirements and conditions change continuously. There is hence a clear need for enabling dynamic reconfiguration of the network protocols to adapt to changing situations.

In this work a light-weight approach is presented that enables to change configuration settings in the entire network stack using only a minimal amount of resources. Moreover, the presented solution is able to expose the configuration settings both locally, for a local controller, and remotely, for a network-wide controller. The core of the solution is completely agnostic to the access method (e.g. local or remote). The communication protocol for enabling remote access is also transparent. The proof-of-concept was implemented in Contiki and uses the REST-based ERBIUM Constrained Application Protocol (CoAP) supported by Contiki.

2 Background

This section gives an overview of the relevant protocol standards and operating systems targeted by the proposed control extensions. For each standard also the possible configuration settings, as described in the standard, are explored. Note that currently they can only be modified at compile time in the targeted OSs.

2.1 Operating Systems

TinyOS, RIOT and Contiki are operating systems specifically designed for constrained sensor devices in the IoT. They share three common features [3]: (1) platform and hardware abstraction for portability; (2) multi-tasking or multi-threading support; and (3) lightweight IPv6 compliant network stack.

Although these OSs also provide alternative network stacks, the IPv6 stack is chosen as default. This is because there is a continuous push towards standardization within the IoT ecosystem. Therefore, the protocols included in the default IPv6 stack are selected as primary candidates for the dynamic reconfiguration extensions.

The default IPv6-compliant network stack is illustrated on the left side (a) in Fig. 1. The following protocols and standards are included: an IEEE-802.15.4-2006 compliant PHY and MAC, 6LowPan header compression, IPv6 (addressing,

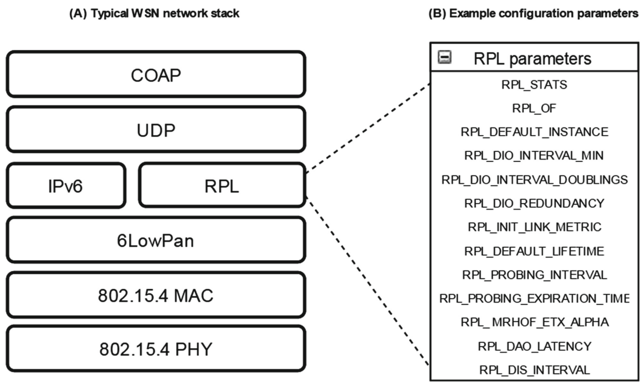

Fig. 1. (A) IPv6 compliant network stack available in TinyOS, Contiki and RIOT. (B) possible configuration parameters to fine-tune the RPL routing protocol.

headers and ICMP), Routing Protocol for Low-Power and Lossy Networks (RPL) routing, TCP/UDP transport and CoAP.

2.2 Network Protocols and Standards

This section briefly summarizes the main configuration settings in the standard IPv6 stack [13]. In total there are 57 available parameters. The possible configuration settings for the RPL routing protocol are illustrated, as example, on the right side (b) of Fig. 1.

The PHY and MAC protocols are based on the IEEE-802.15.4 standard [6]. PHY settings include channel, tx power and CCA threshold. Many custom implementations of MAC protocols exist. In Contiki, the default radio duty cycling MAC protocol is ContikiMAC [2]. It uses periodical wake-ups to listen for packet transmissions from neighbours. The wake-up interval can be modified as well as the number of listens (i.e. CCA checks) during each periodical wake-up or before packet transmission. On top of ContikiMAC, a CSMA based protocol controls the medium contention and packet retransmissions which can also be configured. Protocols with similar behaviour are also available in TinyOS and RIOT.

The network layer includes RPL [14], a proactive, distance-vector routing protocol specifically designed for Wireless Sensor Networks (WSN)s. RPL uses control packets (DIO, DAO and DIS) for building a tree like topology, called a Destination-Oriented Directed Acyclic Graph (DODAG). Many settings allow to fine-tune the various intervals that are used for maintaining the DODAG. Also the link estimation algorithms can be changed and configured. Next to RPL, various parameters controlling the IPv6 neighbour discovery [11] process can be configured. Also TCP/UDP implementations allow to configure the number of retransmissions and various time-out settings.

The application layer protocols tailored for WSNs focus mainly on integrating the sensing and actuating applications in the IoT. One of the most prominent examples is CoAP [12], a REST based protocol that runs over UDP and allows to define resources (e.g. sensors and/or actuators) which can be retrieved or changed using GET/POST/PUT methods using a response-request approach. CoAP can be easily integrated in web-based applications and has a limited overhead. The number of retransmissions and various time-outs and intervals used by the CoAP engine can be configured. Alternatives [10] for CoAP are MQTT and AMQP, both run over TCP and use a publish-subscriber approach managed by a message broker that allows nodes to publish and/or subscribe to topics. Compared to COAP they have a higher overhead and are less supported by operating systems for WSNs.

3 Design

This section discusses the design of the extensions required to support dynamic reconfiguration. First the requirements will be summarized, then the high-level architecture will be described and, subsequently, the communication flow. Finally, the most appropriate application layer protocol will be chosen.

3.1 Requirements

From a functional viewpoint the main requirement is to enable *updating configuration parameters of network protocols* after deploying the network. This functionality must be use-able by both a node-local and network-wide control engine.

In order to support remote control, the system must allow *automatic parameter discovery and bootstrapping* in the entire network. Moreover, *batch configuration* is required for enabling to change multiple parameters at once. From a user perspective, it must be possible to reconfigure the network via *web-based applications.*

From a non-functional viewpoint the main requirements is *resource efficiency.* The memory, CPU and network overhead must be as small as possible and scale with the number of parameters that can be changed at run-time.

Other important concerns are *modifiability, portability and compatibility.* To address them, the control engine must be (a) independent of the protocol used for providing remote access; (b) easily ported to existing operating systems; and (c) compatible with web-based applications.

From the aforementioned requirements, it can be deducted that a generic method for accessing configuration settings is required both on the node-local and network-wide level. For this purpose parameters are maintained in a repository and can be reconfigured using a generic interface implemented by a control engine. Different application layer protocols can be used for enabling remote access.

3.2 Architecture

Figure 2 illustrates the high-level design of the architecture. It includes the following entities: (a) *a sensor configuration server*, which offers a control API or UI for external users (human or software); (b) *a sensor gateway server* that maintains a network-wide view on the current configuration settings and acts as a border router for the WSN; and (c) *the actual sensor devices* that can be reconfigured.

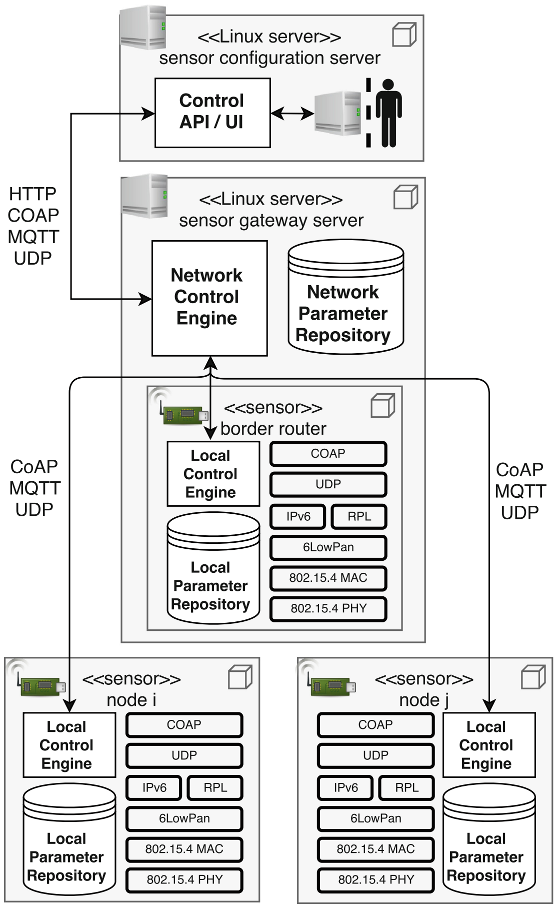

Fig. 2. The architecture of the reconfiguration system.

The sensor devices implement a *local parameter repository*, which maintains a local view on the current node configuration settings, and a *local control engine* implementing a remote configuration interface

The local control engine is responsible for (a) making parameters discoverable and participating in the bootstrapping phase; (b) implementing a generic interface that allows to get/set a specific parameter or a group of parameters; and (c) parsing remote configuration messages and performing the necessary interface calls for the get or set operation. To enable all these interactions, the local parameter repository stores references to each parameter. This reference contains function pointers to the getters and setters provided by the different protocols.

The sensor gateway server includes a sensor border router that acts as a gateway for the sensor network. The Linux host further implements a *network parameter repository*, which maintains a network-wide view on the current configuration of each node, and a *network control engine* implementing a remote configuration interface.

The network-wide control engine is responsible for (a) device discovery and bootstrapping; (b) enabling remote access to the configuration parameters; (c) performing batch configurations in a transactional manner; (d) translating messages between the local sensor network and the remote control API; (e) input validation when changing settings; and (f) authenticating remote access. The network parameter repository serves as a cache during get operations and facilitates roll-backs during set operations.

The sensor configuration server serves as a single entry point for reconfiguring the network both for humans (UI) and software processes (API). It consists of a single component, *the control API/UI* implementing an easy to use API and UI. This component is responsible for translating the API/UI calls into configuration messages and parsing the result.

3.3 Communication Flow

Figure 3 illustrates an example communication flow between the different entities in the architecture, the active components are depicted using white boxes and bold text. In this example HTTP is used by the Control API to configure sensor j. For this purposes, the network control engine translates the HTTP requests/responses into CoAP requests/responses and vice-versa. Another possibility is to directly use CoAP in the Control API. The network control engine will then serve as a proxy for delegating COAP requests/responses. The intermediate sensor nodes (e.g. border router and node i) do not process the CoAP message but forward it to the destination using RPL. Packets coming from the sensor network are injected directly in the Linux IPv6 stack by the border router.

3.4 Application Layer Protocol

Selecting the appropriate application layer protocol for exchanging configuration messages across the network is very important because this will have a high impact on the resource efficiency of the overall solution. Several candidates were

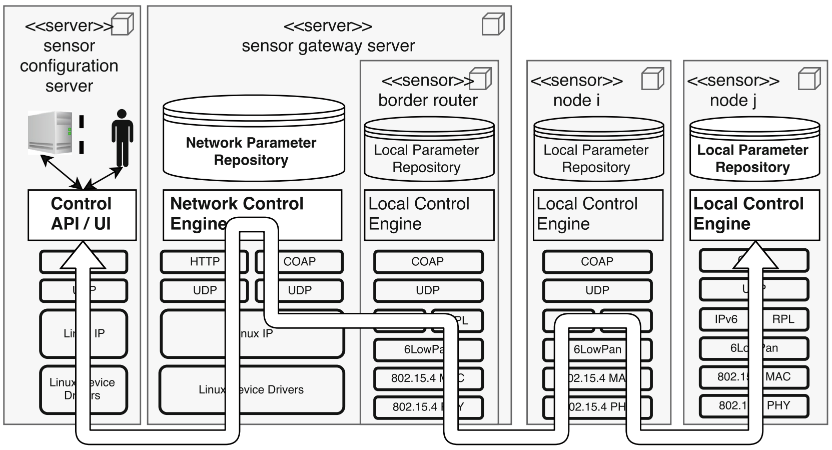

Fig. 3. The communication flow between the different entities in the architecture. The white boxes with bold text depict the active components in this example.

compared in [7] and evaluated based on the device memory requirements and message size overhead.

The most dominant application layer protocol for constrained IoT devices today is CoAP. It has built-in support for resource (e.g. parameter) discovery and block wise (e.g. batch configuration) transfers. From a functional viewpoint, all required features are present. Since CoAP is tailored for constrained devices, the memory and CPU requirements are limited. Moreover, the message overhead is also minimal because the CoAP header is very small and UDP is used as transport protocol. With portability and compatibility in mind, CoAP is also a logical choice because it is well supported by nearly all OSs and easily integrateable in web-based systems since it is REST based.

An alternative for CoAP is MQTT [5], a publish-subscriber system with a central MQTT broker that runs over TCP. MQTT clients can publish or subscribe to topics (e.g. parameters). For each parameter, two topics are required: (1) one published by the sensor node for supporting the get operation; and (2) one published by the configuration server for supporting the set operation. Because of this, MQTT will have a much higher device memory overhead. Also the message overhead will be bigger since it runs over TCP. Moreover, it is less supported, only a Contiki implementation is available. Other alternatives are AMQP and XMPP. Both also use TCP as transport protocol and have much higher device memory requirement and message overhead since they are not tailored for constrained devices.

To conclude, *CoAP is the most appropriate application layer protocol* to support dynamic reconfiguration, as also indicated in [13].

4 Evaluation

The evaluation consists of an analytical part, that investigates how CoAP can be most efficiently used for enabling remote access to configuration parameters, and an experimental part, in which the memory overhead and latency for changing parameters are determined. Also a proof-of-concept case study is presented emphasizing the practical use of the dynamic reconfiguration solution.

4.1 Analysing CoAP Memory Overhead

The CoAP memory requirements constitute of the fixed overhead for the CoAP engine (8.5 kB ROM/1.5 kB RAM [8]) and the variable amount of ROM occupied by the CoAP resources. The additional memory overhead for exposing configuration settings hence depends on the number of CoAP resources required to expose the parameters. Three granularity levels are considered: (1) a CoAP resource per configuration parameter; (2) a CoAP resource per protocol; and (3) one CoAP resource for the entire network stack.

In principal, CoAP is text based and resources are identified using unique string names encoded in the resource URI. Both need to be stored in the ROM memory of each sensor device. Depending on the granularity level, the string name of each parameter (1), protocol (2) or stack (3) is stored in memory causing extra ROM overhead. Moreover, when using granularity level (2) or (3), parameters still need to be identified. This can be done using either unique names, encoded in the URI query variable, or unique IDs, encoded in the payload.

In order to make well-founded decisions, the impact on the ROM memory usage for different granularities was analysed using stub resources in Contiki for CoAP. This allows to devise a mathematical model that can be applied on a real example network stack to estimate the overhead in each options. The total ROM overhead of an option is denoted by s_{rom}^{total} and comprises of S_{rom}^{res}, or the ROM required for the resource definition and the GET/POST/PUT handlers, and the *string_length* of the resource name. Note that S_{rom}^{res} will be different for each granularity because the GET/POST/PUT handlers are implemented differently

Also the parameter identification method in level (2) and (3) were investigated. For this purpose the auxiliary function $s_{rom}^{id}(param_i)$ is defined (Eq. 1) that returns the *string_len* when using unique names or $sizeof(int)$ when using unique IDs.

$$s_{rom}^{id}(param_i) = \begin{cases} string\_len(param_i) & \text{if } id \text{ is string} \\ sizeof(integer) & \text{if } id \text{ is integer} \end{cases} \quad (1)$$

A resource per parameter enables direct addressing of parameters without requiring any transformation. It is the most straightforward for integration in browsers using add-ons such as Copper [9]. The ROM overhead s_{rom}^{total}, on the other hand, will be high because for each parameter the string name must be stored and a resource must be defined (S_{rom}^{res}) as denoted in Eq. 2.

$$s_{rom}^{total} = \sum^{param_i} (string\_len(param_i) + S_{rom}^{res}) \tag{2}$$

where $S_{rom}^{res} = 157$

A resource per protocol groups parameters on a protocol level. They are addresses indirectly via the protocol resource implying that an if-else structure is required in the GET/POST/PUT handlers for identifying the correct parameter. Equation 3 defines the total ROM overhead s_{rom}^{total} as the sum over all $proto_i$ of the ROM memory required for storing the protocol name, the fixed CoAP resource overhead (S_{rom}^{res}) and, per parameter, the identification ($s_{rom}^{id}(param_j)$) and if-else (S_{rom}^{ifelse}) overhead.

$$s_{rom}^{total} = \sum^{proto_i} (S_{rom}^{res} + string\_length(proto_i) + s_{rom}^{param}(param_j \in proto_i)) \tag{3a}$$

$$s_{rom}^{param}(param_j \in proto_i) = \sum^{param_j} \left(s_{rom}^{id}(param_j) + S_{rom}^{ifelse}\right) \tag{3b}$$

where $S_{rom}^{res} = 280$ and $S_{rom}^{ifelse} = 40$

A resource for the entire stack has the advantage that there is looser coupling with the protocols, compared to the previous options. A tight coupling implies that a protocol update also require updating the CoAP resources(s). The third approach, however, requires an explicit implementation of a parameter repository that can be used by the generic resource to manipulate configuration settings and by the protocols to (de-)register parameters. Equation (4) expresses the ROM overhead s_{rom}^{total} when using a single resource for the entire stack. Now the fixed CoAP resource overhead (S_{rom}^{res}) also includes the resource name and the parameter repository implementation. For each parameter, a fixed amount of ROM S_{rom}^{param} is required for the parameter structure. The identification overhead $s_{rom}^{id}(param_i)$ depends on the chosen method.

$$s_{rom}^{total} = S_{rom}^{res} + \sum^{param_i} \left(s_{rom}^{id}(param_i) + S_{rom}^{param})\right) \tag{4a}$$

$$s_{rom}^{id}(param_i) = \begin{cases} name\_length(param_i) & \text{if } id \text{ is string} \\ 2 & \text{if } id \text{ is numeric} \end{cases} \tag{4b}$$

where $S_{rom}^{res} = 392$ and $S_{rom}^{param} = 12$

Conclusion: Figure 4 gives an overview of the ROM overhead estimated for the different resource granularities and identification methods. The results clearly show that a single generic resource requires 80 % less memory (1.2 kB) compared to a resource per parameter (11.6 kB) and 60 % less compared to a resource per protocol (4.5 kB). Using unique IDs instead of names also has a major impact. Given the size in ROM of the default Contiki IPv6 stack (+ − 30 kB), using *a single generic CoAP resource and unique IDs is the preferred choice.*

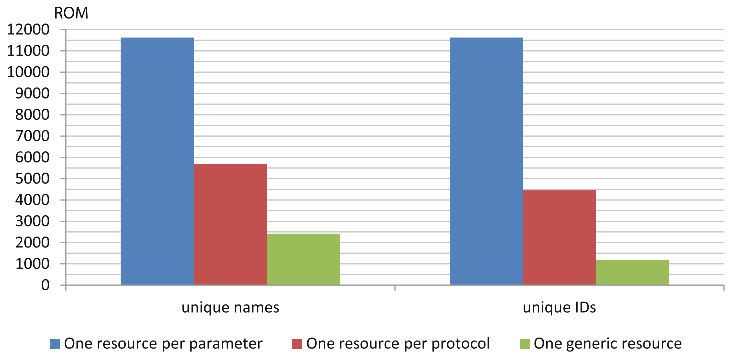

Fig. 4. Estimated ROM overhead for different resource granularities and identification methods.

4.2 Experimental Evaluation

Evaluation Set-Up: The sensor configuration and gateway server were implemented on a general purpose embedded PC running Linux. The sensor code was developed in Contiki 3.0 and executed on a Zolertia Z1 (16 MHz CPU, 92 KB ROM, 10 KB RAM and an IEEE-802.15.4 compliant transceiver). A single resource combined with a parameter repository is used for configuring the network stack. All communication between the different entities is CoAP based. On Linux libCoAP [1] is used while in Contiki the ERBIUM CoAP [8] engine is utilized.

Latency: The average latency for changing parameters depends on the number of PUT/POST requests needed to perform a batch configuration on all nodes. It is measured on the sensor gateway by calculating the delay between the first request and last response. The average latency is an important performance indicator because it defines the duration in which the network is in an inconsistent state. Figure 5 illustrates the average latency in seconds for one to twenty POST/PUT requests (e.g. number of nodes) in steps of four. Also the standard deviation over all experiments is indicated. The results clearly show that the average latency scales with the number of POST/PUT requests.

Case Study: Dynamically Reconfiguring the ContikiMAC Duty-Cycle: To illustrate the usefulness of dynamic reconfiguration, a simple case study is presented in which the duty-cycle of ContikiMAC is dynamically adapted based on the application load. It is applicable on use-cases such as a HVAC monitoring and control system which requires more traffic during the office hours. The duty-cycle of ContikiMAC [2], normally statically defined at compile time, is now dynamically configured using the $ChannelCheckRate$ (CCR) parameter (e.g. the rate for checking RX activity). Figure 6 shows the energy consumed daily by the radio (RX/TX) and CPU (active/LPM) for three different CCR settings.

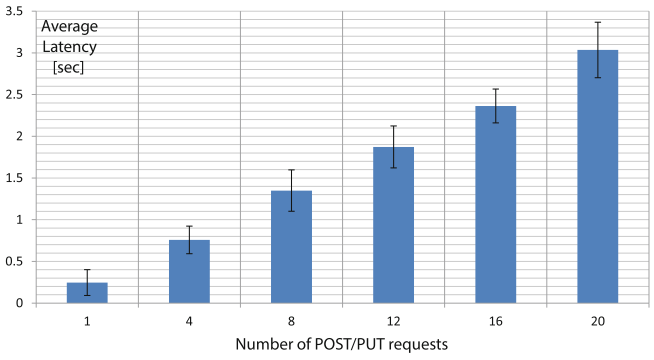

Fig. 5. Average latency for increasing number of POST/PUT requests (e.g. nodes). Also the standard deviation is denoted on the chart.

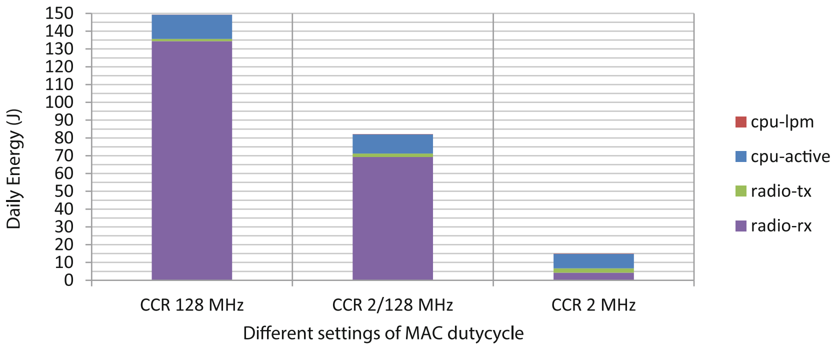

Fig. 6. The daily energy consumption for different settings of the channel check rate (CCR) in ContikiMAC.

A high duty-cyle (left) results in a high reliability at the cost of much energy spent in RX mode. On the other hand a low duty cycle (right) requires ten times less energy at the expense of reliability. When using a high-duty cycle during office hours and a low duty-cycle otherwise (middle), a high reliability can be achieved for half the amount of energy. The energy was mea

5 Conclusions

This paper presents a flexible approach for enabling dynamic reconfiguration of protocol settings in the entire network stack, either local or remote. The high level architecture is applicable on multiple OSs and compatible with different application layer protocols for providing remote access to the devices. The Contiki implementation uses CoAP for this purpose. A single CoAP resource is defined for the entire network stack combined with a parameter repository that allows protocols to register parameters. A configuration server can reconfigure the entire network using CoAP via the gateway server that acts as a CoAP proxy.

By carefully considering how CoAP is used, the overall memory overhead could be reduced from 11.6 kB to 1.2 kB. The proof-of-concept results also show that the latency for reconfiguring parameters scales with the number of POST/PUT requests (e.g. the number of nodes). To reconfigure a parameter in a network of twenty nodes, on average three seconds are required. The case-study that dynamically reconfigures the duty-cycle of ContikiMAC based on the traffic load, shows that 50 % of energy can be saved without sacrificing other performance indicators such as reliability and throughput.

Future work could built-up from this solution and develop more advanced case-studies where the effect of changing multiple parameters on the network performance can be investigated. To allow this, only limited modifications are required in the protocols to expose the parameters. The developed solution can hence be used by many experimenters for optimizing the network performance via parameter reconfiguration.

Acknowledgments. This work was partially supported by the Fund for Scientific Research-Flanders (FWO-V), project “ASN: wireless Acoustic Sensor Networks”, grant #G.0763.12, and the agency for Innovation by Science and Technology Flanders (IWT-V), project “SAMURAI: Software Architecture and Modules for Unified RAdIo control”, and the iMinds IoT Strategic Research program, and European Commission Horizon 2020 Programme under grant agreement n645274 (WiSHFUL).

References

1. Bergmann, O.: libcoap: C-implementation of CoAP (2015)
2. Dunkels, A.: The contikimac radio duty cycling protocol. Technical report, SICS (2011)
3. Farooq, M.O., Kunz, T.: Operating systems for wireless sensor networks: a survey. Sensors **11**(6), 5900–5930 (2011)
4. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of things: a vision, architectural elements, and future directions. Future Gener. Comput. Syst. **29**(7), 1645–1660 (2013)
5. Hunkeler, U., Truong, H.L., Stanford-Clark, A.: MQTT-S: a publish/subscribe protocol for wireless sensor networks. In: Communication Systems Software and Middleware and Workshops, COMSWARE (2008)
6. IEEE: IEEE standard for local and metropolitan area networks-part 15.4: low-rate wireless personal area networks (LR-WPANS). IEEE Std 802.15.4-2011 (2011)
7. Karagiannis, V., Chatzimisios, P., Vazquez-Gallego, F., Alonso-Zarate, J.: A survey on application layer protocols for the internet of things. Trans. IoT Cloud Comput. **3**(1), 11–17 (2015)
8. Kovatsch, M., Duquennoy, S., Dunkels, A.: A low-power CoAP for Contiki. In: Mobile Adhoc and Sensor Systems, MASS (2011)
9. Kovatsch, M.: Demo abstract: humanCoAP interaction with copper. In: Distributed Computing in Sensor Systems, DCOSS (2011)

10. Luzuriaga, J., Perez, M., Boronat, P., Cano, J., Calafate, C., Manzoni, P.: A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. In: Consumer Communications and Networking Conference, CCNC (2015)
11. Narten, T., Nordmark, E., Simpson, W., Soliman, H.: Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (2007)
12. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (2014)
13. Sheng, Z., Yang, S., Yu, Y., Vasilakos, A.V., McCann, J.A., Leung, K.K.: A survey on the ietf protocol suite for the internet of things: standards, challenges, and opportunities. IEEE Wirel. Commun. **20**(6), 91–98 (2013). doi:10.1109/MWC.2013.6704479
14. Winter, T., Thubert, P., Brandt, A., Hui, J., Kelsey, R., Levis, P., Pister, K., Struik, R., Vasseur, J., Alexander, R.: RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550 (2012)