SoftwareX 18 (2022) 101089

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

The Contiki-NG open source operating system for next generation IoT devices

George Oikonomou [a,*], Simon Duquennoy [b,c], Atis Elsts [d], Joakim Eriksson [b], Yasuyuki Tanaka [e], Nicolas Tsiftes [b]

[a] Electrical and Electronic Engineering, University of Bristol, Bristol, UK
[b] RISE Research Institutes of Sweden, Kista, Sweden
[c] Inria Lille - Nord Europe, France
[d] Institute of Electronics and Computer Science (EDI), Riga, Latvia
[e] Corporate Research and Development Center, Toshiba, Kawasaki, Japan

ARTICLE INFO

Article history:
Received 12 August 2021
Received in revised form 23 February 2022
Accepted 13 April 2022

Keywords:
Contiki-NG
Internet of Things
Resource-Constrained Devices

ABSTRACT

Contiki-NG (Next Generation) is an open source, cross-platform operating system for severely constrained wireless embedded devices. It focuses on dependable (reliable and secure) low-power communications and standardised protocols, such as 6LoWPAN, IPv6, 6TiSCH, RPL, and CoAP. Its primary aims are to (i) facilitate rapid prototyping and evaluation of Internet of Things research ideas, (ii) reduce time-to-market for Internet of Things applications, and (iii) provide an easy-to-use platform for teaching embedded systems-related courses in higher education. Contiki-NG started as a fork of the Contiki OS and retains many of its original features. In this paper, we discuss the motivation behind the creation of Contiki-NG, present the most recent version (v4.7), and highlight the impact of Contiki-NG through specific examples.

Code metadata

Current code version	v4.7
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00149
Code Ocean compute capsule	N/A
Legal Code License	3-Clause BSD
Code versioning system used	git
Software code languages, tools, and services used	C, Java (Cooja Simulator), Python (Multiple ecosystem tools)
Compilation requirements, operating environments & dependencies	Linux, macOS, Windows (partial support)
If available Link to developer documentation/manual	Guides and Tutorials: https://github.com/contiki-ng/contiki-ng/wiki API Documentation: https://contiki-ng.readthedocs.io
Support email for questions	Gitter: https://gitter.im/contiki-ng GitHub Discussions: https://github.com/contiki-ng/contiki-ng/discussions Stack overflow: https://stackoverflow.com/questions/tagged/contiki-ng

1. Motivation and significance

Research in the field of Wireless Sensor Networks started almost two decades ago [1]. During the initial years, research activities targeted extremely resource-constrained devices (kB or less of RAM, bit-level radio Application Programming Interface (API)), and utilising highly specialised application-specific software was the norm. As an example of this approach, the TinyOS operating system [2] used a purely event-based execution model and a custom programming language called nesC.

1.1. Historical background — The original Contiki OS

The original Contiki Operating System (OS) [3] was open sourced in 2006, but development had started as early as 2003. It was designed for resource-constrained wireless sensor devices with code memory in the order of 100 kB and less than 10 kB of volatile memory. Contiki was a major step in the evolution of modern IoT operating systems, with its main strengths being:

* Corresponding author.
E-mail address: g.oikonomou@bristol.ac.uk (George Oikonomou).

https://doi.org/10.1016/j.softx.2022.101089

- Use of the standard C programming language[1]; application developers no longer needed to learn a bespoke language such as nesC.
- Event-based kernel. In conjunction with tickless, platform-specific main loop code (implemented for some platforms) it allowed for fast reaction time to external events and energy-efficient execution.
- Cooperative multi-threading based process API [4]. It greatly simplified application programming, reducing the need for both explicit state machines and locks (compared to event-based API and preemptive thread API, respectively).
- Early, standards-compliant support for network protocols such as the Internet Protocol (IP) and IPv6 protocols through the uIP [5] stack. It also featured some of the very early open implementations of IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN) and of the Routing Protocol for Low-Power and Lossy Networks (RPL).

The official Contiki distribution also included Cooja, a simulator for IEEE 802.15.4 networks of devices running Contiki-based firmware. Alongside Cooja and other tools of its ecosystem, Contiki went on to become an extensively-used tool for academic research in the Wireless Sensor Network (WSN) field. As an indicator of Contiki's impact, the authors of this paper identified in excess of 2000 peer-reviewed publications by using the "Contiki AND OS" search string on the Scopus database. Among those papers, approximately 350 have been published in 2018 alone. Moreover, according to Scopus one of the first publications documenting Contiki [3] has been cited in excess of 1300 times.

1.2. From Contiki to Contiki-NG

As the Contiki OS gained traction and started being used extensively, some limitations began to emerge:

- Large legacy of old, extremely resource-constrained platforms. The 8-bit and 16-bit low-power microcontrollers which Contiki was originally designed for became obsolete over time; 32-bit ARM Cortex-M based devices with more sophisticated low-power modes are the new norm [6].
- Support for non-standard protocols. As the field of wireless sensor network research evolved to become one of the core enabling technologies of the IoT, interoperability and standards became increasingly important. Alongside standards-compliant protocol implementations, the Contiki code-base also featured older, experimental, non-standard protocols initially contributed as research artefacts. One such example is the Rime stack.

This combination of legacy platform code and support for non-standard networking protocols increased maintenance complexity and impeded code evolution.

Contiki-NG was first released in November 2017. Its aim was to eliminate some of Contiki's limitations in order to enable easier maintenance and quicker evolution. The main vision is to focus on:

- Standard protocols. Some of the standards supported by Contiki-NG are IEEE 802.15.4 TSCH, 6LoWPAN, 6TiSCH, RPL, CoAP, MQTT, and LWM2M.
- Support for modern hardware platforms.
- Dependability (reliability and security) through modern development practices, continuous integration using simulations and a physical testbed, and security testing techniques.

The first release (v4.0) was a fork of the Contiki OS. Besides the aforementioned changes in the high-level focus of the OS, Contiki-NG added a new configuration and logging system, a new lightweight and reliable RPL implementation (RPL-Lite), and a network administration shell. It also brought with it an extensive cleanup of the codebase, with legacy platforms, protocols and services removed, so as to eliminate constraints on future developments. All Contiki-NG releases and their changelogs are available at https://github.com/contiki-ng/contiki-ng/releases.

2. The Contiki-NG project

Contiki-NG introduces many new features, but it also re-uses – with or without modification – many of the features of the original Contiki OS, such as the scheduler, the event-based kernel, data structure manipulation libraries and storage. Contiki-NG also uses with minor modifications multiple networking-related software components, such as the original implementations of 6LoWPAN and RPL (henceforth called "RPL-Classic").

Contiki-NG primarily targets Arm Cortex-M platforms. The official repository includes support for hardware by Nordic Semiconductor, NXP, OpenMote, Texas Instruments, and Zolertia. All those platforms are powered by Cortex-M3 or -M4 chips. Outside the main repository exist numerous Contiki-NG forks that have added support for other hardware, for example platforms powered by ST Microelectronics chips such as those used at the FIT IoT-Lab testbed. To the best of our knowledge, there is no reason why Contiki-NG would not run on Cortex-M0. Lastly, the official repository also includes support for the 16-bit MSP430 architecture, which is mainly used inside the Cooja simulator (Section 1.2). In terms of adding support for more hardware, most of the effort revolves around the implementation of support for new Micro-controller (MCU) architectures and on-chip peripherals, including radio transceivers. Once this support has been added, porting Contiki-NG to a new board is much less onerous. Documentation of the steps required to port Contiki-NG to a new hardware platform can be found on the wiki.[2]

Contiki-NG positions itself within the same landscape as other operating systems for embedded devices (Table 1), such as RIOT [7], Zephyr,[3] Arm Mbed,[4] Apache Mynewt,[5] TinyOS and FreeRTOS. With its implementation of TSCH, RPL-Classic and RPL-Lite, the authors of this paper feel that Contiki-NG fills the niche of low-power, IEEE 801.15.4 wireless mesh networks. For comprehensive quantitative or qualitative comparisons of IoT operating systems, we refer the reader to already extensive literature, such as [8,9].

In the interest of brevity, the remainder of this section focuses on describing technical as well as non-technical aspects of the project that are either entirely new, or that have undergone significant changes since the launch of the Contiki-NG project. Describing the operating system in full detail is considered by the authors to be out of scope of this paper.

2.1. The Contiki-NG architecture and features

Broadly speaking, the Contiki-NG source base can be conceptually broken down into two parts: (i) hardware-independent and

[1] Developers should not use C switch statements inside protothreads. This was a limitation of the original Contiki OS and has been carried over to Contiki-NG. With the exception of this limitation, Contiki-NG is fully C standards-compliant.

[2] https://github.com/contiki-ng/contiki-ng/wiki/Porting-Contiki%E2%80%90NG-to-new-platforms

[3] https://www.zephyrproject.org/

[4] https://os.mbed.com/

[5] https://mynewt.apache.org/

Table 1
Feature overview of embedded OSs.

Project	Networking	Licence	Language	Threading
Contiki-NG	TSCH, 6LoWPAN, RPL	BSD	C, C++	Cooperative
Contiki	TSCH, 6LoWPAN, RPL	BSD	C	Cooperative
Apache Mynewt	BLE, LoRa, TCP/IP	Apache 2.0	C, C++	Preemptive
Arm Mbed	BLE, LoRa, lwIP	Apache 2.0	C++	Preemptive
FreeRTOS	TCP/IP	MIT	C, C++	Preemptive
RIOT	6LoWPAN, BLE	GNU LGPL	C, C++	Preemptive
TinyOS	6LoWPAN	BSD	nesC	Optional preemptive
Zephyr	BLE, Thread, 6LoWPAN	Apache 2.0	C, C++	Optional preemptive

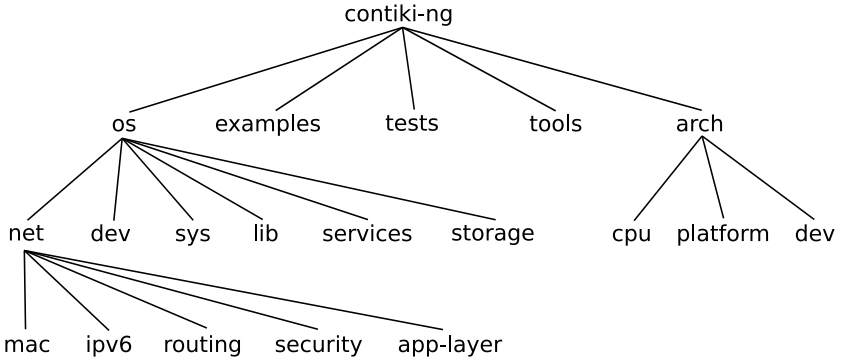

Fig. 1. Contiki-NG directory structure.

(ii) hardware-specific. The former hosts portable, cross-platform implementations of all hardware-agnostic components of the OS, including the kernel, software timers, data structure libraries, and networking protocols.

The latter provides the code required to make the OS work on specific devices. It consists of drivers for hardware components including timers, radio interfaces and other on-chip and off-chip peripherals, such as Universal Asynchronous Receiver/Transmitter (UART), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), LEDs, user buttons and sensing elements. In order to make the addition of support for more devices as effortless as possible and in order to increase code portability, Contiki-NG defines Hardware Abstraction Layers (HALs) for common interfaces, including General Purpose Input/Output (GPIO) and SPI. A Hardware Abstraction Layer (HAL) for I2C is part of the short-term roadmap. Those HALs include declarations of hardware-specific functions that need to be implemented by the developer of a new hardware port. This allows new platform developers to focus their effort on implementing the hardware-specific parts of well-defined APIs without having to devise new programming interfaces. Once chip-specific implementations have been developed, adding support for different platforms based on the same chip merely requires trivial configuration. Where possible, within the HALs Contiki-NG provides platform-independent functions that can be used to access hardware elements. Those functions will work on all supported platforms without any further effort. For example, a developer can use `spi_transfer()` to send a sequence of bytes to an SPI peripheral; this function has well-defined behaviour on all hardware platforms that implement the hardware-specific parts of the SPI HAL.

2.1.1. Repository structure

Fig. 1 illustrates the directory structure of the Contiki-NG codebase. All platform-independent code can be found under '`os/`' and all hardware-specific drivers can be found under '`arch/`'. The '`tests/`' directory hosts the automated Continuous Integration testing suite (see Section 2.5), whereas '`tools/`' contains helper utilities, such as scripts used to upload firmware to supported devices, and to generate documentation.

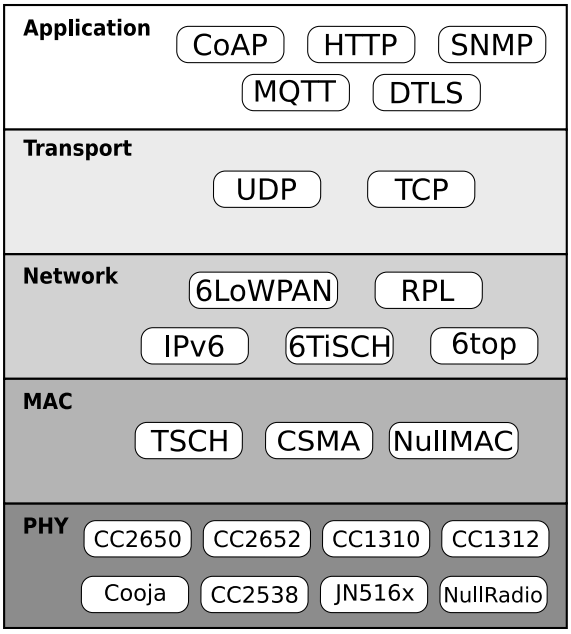

Fig. 2. Contiki-NG network stack.

2.2. Networking support

As discussed in Section 1.2, among Contiki-NG's main goals is to provide standards-compliant, dependable low-power networking for severely resource-constrained wireless embedded devices. Fig. 2 illustrates an overview of the Contiki-NG network stack. In the remainder of this subsection we provide an overview of the Contiki-NG implementation of selected supported protocols and specifications.

2.2.1. TSCH and 6TiSCH

Time Slottted Channel Hopping (TSCH) is a Medium Access Control (MAC) layer defined in the IEEE 802.15.4-2015 standard

[10]. It is aimed towards Industrial Internet of Things and other use cases that require high reliability, low latency, and low energy consumption. 6TiSCH [11] is a set of existing and upcoming IETF standards that aim to describe a complete TSCH-enabled IPv6 network stack for these use cases. Contiki-NG supports TSCH [10], the IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) minimal configuration [12], the 6top Protocol (6P) [13], RPL [14], and other protocols for 6TiSCH. The TSCH and 6TiSCH implementations [15] of Contiki-NG have been validated in a number of IETF interoperability events and were shown to be interoperable with other implementations, including OpenWSN.

Two 6TiSCH scheduling functions are available on Contiki-NG: Orchestra [16] and the Minimal Scheduling Function (MSF) [17]. Orchestra is a fully autonomous scheduling function which does not need any signalling traffic to configure TSCH links. On the other hand, MSF uses 6P for TSCH link allocations, and adapts its schedule to traffic changes.

2.2.2. RPL-Classic and RPL-lite

RPL is a protocol defined by the IETF RFC6550 [14] for routing over low-power and lossy networks. In a nutshell, nodes build a multi-hop Directed Acyclic Graph (DAG) topology, enabling routing towards a root (along a gradient) or towards any other node (following either routing tables or via source routing). The Contiki OS provided one of the earliest open RPL implementations already back in 2010 (“RPL-Classic”), which was evidenced as being interoperable with the version of RPL distributed with TinyOS [18]. Contiki-NG adopted this implementation and contributed a new version, called “RPL-Lite”.

RPL-Lite achieves two things: (i) It retains only a selected subset of relevant modes of operation from the very flexible standard, based on years of experience from RPL-Classic, and (ii) it offers a complete re-factoring of a code-base that had accumulated a substantial amount of technical debt over the years. As such, RPL-Lite only supports one single DAG at a time, one single “RPL instance”, and only the non-storing mode of operation. These choices minimise the amount of state maintained at each constrained node in the network, allowing for more robust operation. RPL-Lite was implemented in parallel with research on ultra-reliable RPL [19], and benefits from many of the reliability mechanisms devised as part of this research.

2.2.3. Multicast support

The Contiki-NG kernel supports IPv6 multicast forwarding in 6LoWPANs through an API that allows the easy addition of new multicast forwarding engines. Contiki-NG contributes a standards-compliant implementation of the Multicast Protocol for Low-Power and Lossy Networks (MPL): A multicast forwarding protocol proposed by the Internet Engineering Task Force (IETF) [20]. MPL support accompanies the two multicast engines adopted from the original Contiki OS: (i) Stateless Multicast RPL Forwarding (SMRF) [21,22] and (ii) Enhanced SMRF (ESMRF) [23].

2.2.4. CoAP and LWM2M

The Constrained Application Protocol (CoAP) [24] is an application layer protocol similar to HTTP, but it has been designed to be more suitable for constrained environments. Contiki-NG’s CoAP implementation supports many key CoAP features including: (i) Block-wise transfers [25] for transporting large blocks of data that cannot fit in a single packet, and (ii) CoAP observations [26]. The implementation and respective API make it easy to add new CoAP resources registered on a specific path, as well as more complex resource handlers that can be invoked on all CoAP requests. The CoAP implementation is interoperable with the libcoap.[6] command line tools, as well as with node-coap[7] Interoperability with the former is tested automatically as part of our GitHub actions CI workflow (Section 2.5) to prevent regressions. The implementation is also interoperable with the “Copper (Cu)”[8] and “Copper for Chrome (Cu4Cr)”[9] browser addons.

Contiki-NG also supports version 1.0 of the Open Mobile Alliance (OMA) Lightweight M2M (LWM2M) specification with plain text, JavaScript Object Notation (JSON) and Type-Length-Value (TLV) data formats. The implementation supports the LWM2M security mode with pre-shared keys and the server and security objects for configuring security. The LWM2M engine implementation is accompanied by an implementation of the LWM2M “IP for Smart Objects” (IPSO) objects, which be used as an example starting point by developers wishing to adopt LWM2M for their work. The implementation is interoperable with Eclipse Leshan[10] as well as with Eclipse Wakama[11]. Interoperability with Leshan is also tested as part of our CI workflow (Section 2.5).

The original Contiki OS supports CoAP as well as LWM2M, but both implementations have been reworked and evolved extensively by the Contiki-NG project.

2.2.5. MQTT

Contiki-NG features a lightweight client implementation of Message Queuing Telemetry Transport (MQTT), an open publish/-subscribe protocol. The implementation supports MQTT versions 3.1 (adopted from Contiki OS) and v3.1.1 (contributed by Contiki-NG). Support for MQTT version 5 has already been merged into the development branch and will be included in the next release. Contiki-NG also provides a platform-independent MQTT client example that is interoperable with the Eclipse Mosquitto MQTT broker, as well as with the IBM Watson IoT Platform.[12] Interoperability with Mosquitto is tested as part of our CI workflow (Section 2.5) to prevent regressions.

2.2.6. Limitations

Currently, Contiki-NG does not officially support: (i) The Coordinated Sampled Listening (CSL) mode of IEEE 802.15.4; (ii) 6LoWPAN Neighbor Discovery; (iii) 6LoWPAN “Mesh-Under” operation (only “Route-Over” is supported); and (iv) secure messages in RPL. Open source implementations for Contiki-NG do exist in mirror repositories and adding official support is among the project’s long-term ambitions.

2.3. Responsible disclosure and security advisories

Contiki-NG places increased focus on software security through fuzz testing and other methods. The project has a dedicated email address that can be used for responsible disclosure of security vulnerabilities. As a result of internal testing processes and community reports, the project recently released its first security advisories.

2.4. Documentation

Contiki-NG is documented in a GitHub-hosted wiki[13] that contains an extensive list of guides and tutorials for beginners as well as for more advanced users.

Moreover, the Contiki-NG source codebase is annotated with Doxygen[14] comments that are used to generate an HTML-based

6 https://libcoap.net/

7 https://www.npmjs.com/package/coap
8 https://github.com/mkovatsc/Copper
9 https://github.com/mkovatsc/Copper4Cr
10 https://www.eclipse.org/leshan/
11 https://github.com/eclipse/wakaama
12 https://quickstart.internetofthings.ibmcloud.com
13 https://github.com/contiki-ng/contiki-ng/wiki
14 http://www.doxygen.nl/

API documentation. The API documents can be built and viewed by users locally on their computers, but are also hosted online on "Read the Docs"[15]. This automatically-updated online space hosts multiple versions of the API documents: One per Contiki-NG release (since v4.2 when the feature was first introduced), as well as one for the latest version of the `develop` branch.

2.5. Continuous integration testing

Contiki-NG is automatically tested using a Continuous Integration (CI) workflow on GitHub Actions. This workflow replaced the older test suite on the Travis-CI[16] platform with the release of version v4.6. The test suite is triggered automatically each time a code change gets merged into one of Contiki-NG's main git branches, as well as each time a pull request is opened or updated. All pull requests must pass all CI tests before they can be considered for inclusion in the official source codebase. All source code changes must take place through a pull request, including code changes proposed by members of the Contiki-NG maintainers team. This strategy enforces scrutiny by peers, therefore increasing overall code quality and reducing the likelihood of introducing errors.

The Contiki-NG test suite comprises seventeen jobs, with each job executing multiple CI tests. The test suite covers the following code elements:

- Successful compilation of code examples for multiple hardware platforms under multiple different configurations.
- Validation of the correct operation of various elements of the networking subsystem through multiple Cooja simulations and native code execution scenarios.
- Successful compilation of the doxygen API documentation (Section 2.4).

Users who wish to exercise their code with Contiki-NG's CI workflow can do so trivially by enabling GitHub Actions on their fork repository. Alternatively, the Contiki-NG docker image provides all tools required to execute the test suite locally on their computers.

2.6. Nightly builds

In addition to the per-contribution CI described above, we perform automated nightly testbed runs on real hardware: a bespoke, 25-node testbed installed at RISE SICS. Each node consists of both a control device (Raspberry PI) and a set of sensors (at the time of writing, a Zolertia Firefly and a JN516x dongle). Every night, a cron job schedules four 2-hour experiments, each with a different configuration of the network stack. Control nodes deploy to all 25 Firefly devices a firmware that performs response–request communication between the root node and every other node in the network. Each experiment involves over 10k round-trip network packet exchanges over multiple hops on power-saving devices. A log file post-processing script extracts key metrics from each run: end-to-end round-trip delivery ratio, latency, radio duty cycle, hop counts, and frequency of network topology changes. Raw log files and processed statistics are pushed automatically to a public website for visualization and monitoring.[17]

The benefits of nightly builds are twofold. Firstly, they provide automated testing on real hardware, as opposed to simulation/emulation used in CI. Secondly, they give developers feedback, and the ability to compare the performance of various network stack configurations and to spot performance degradation.

[15] https://contiki-ng.readthedocs.io

[16] https://travis-ci.org/contiki-ng/contiki-ng

[17] https://contiki-ng.github.io/testbed/

```
#include "contiki.h"
#include "net/ipv6/simple-udp.h"
#include "net/mac/tsch/tsch.h"
#include "lib/random.h"
#include "sys/node-id.h"

#define UDP_PORT               8765
#define SEND_INTERVAL         (60 * CLOCK_SECOND)

PROCESS(node_process, "RPL Node");
AUTOSTART_PROCESSES(&node_process);
simple_udp_callback rx_callback; /* Defined in the file
    rx_callback.c */

PROCESS_THREAD(node_process, ev, data)
{
  static struct simple_udp_connection udp_conn;
  static struct etimer periodic_timer;
  uip_ipaddr_t dst;

  PROCESS_BEGIN();

  /* Initialization; `rx_callback` is the function for
     packet reception */
  simple_udp_register(&udp_conn, UDP_PORT, NULL, UDP_PORT
     , rx_callback);
  etimer_set(&periodic_timer, random_rand()   if(node_id
     == 1) {  /* Running on the root? */
    NETSTACK_ROUTING.root_start();
  }

  /* Main loop */
  while(1) {
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&
     periodic_timer));
    if(NETSTACK_ROUTING.node_is_reachable()
       && NETSTACK_ROUTING.get_root_ipaddr(&dst)) {
      /* Send network uptime timestamp to the network
     root node */
      uint64_t network_uptime =
     tsch_get_network_uptime_ticks();
      simple_udp_sendto(&udp_conn, &network_uptime,
     sizeof(uint64_t), &dst);
    }
    etimer_set(&periodic_timer, SEND_INTERVAL);
  }

  PROCESS_END();
}
```

Listing 1: Network-wide time synchronisation example

3. Illustrative example

As an indicative Contiki-NG user application example, we select the network-wide time synchronisation demo project.[18] The purpose of the example is to demonstrate how an application developer can use the time synchronisation provided by TSCH in order to determine the delay between the time of packet origination by a source network node and the reception time on the destination node, which also acts as the network time source. This example was selected because the application code (Listing 1) remains simple while demonstrating multiple key interfaces: Contiki-NG process initialisation (lines 23–27); waiting for timer events (line 31); sending network packets (line 36); packet reception (present in the full source code, with the respective function prototype shown in line 12 of the listing); and how to interact with routing and TSCH protocols from the application layer.

The example hides a lot of the advanced Contiki-NG functionality under the hood; some details of how the Contiki-NG network stack handles it are shown in Fig. 3. From the perspective of transmitting and receiving traffic, the application only has a direct interface with the transport layer (the UDP implementation in this particular example) through the `simple_udp_` API. However, the application also has an indirect interface with

[18] Found under `examples/6tisch/timesync-demo` of the source tree

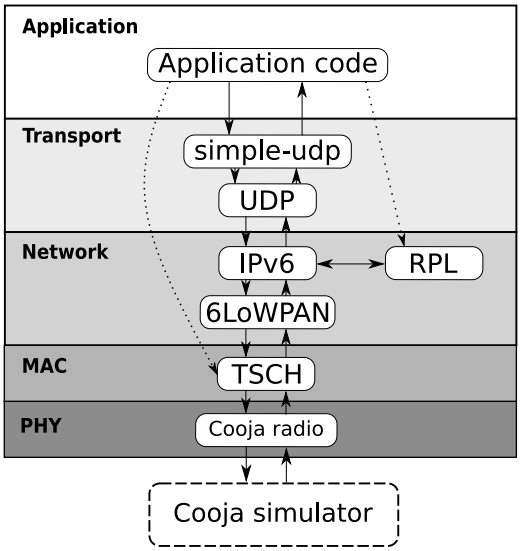

Fig. 3. Contiki-NG networking stack as used by the network-wide time synchronisation example when executed in the Cooja simulator.

RPL and TSCH (dotted lines in Fig. 3). In the former case, the `NETSTACK_ROUTING` API is used in order to determine whether the device has successfully joined the network before it tries to transmit. In the latter case, `tsch_get_network_uptime_ticks()` is used in order to retrieve accurate time information.

On the receive path and depending on network stack configuration, frame reception is either handled through interrupts, or by polling the radio driver inside TSCH RX timeslots. In the former case of interrupt-based operation, interrupts are processed in architecture-specific handler functions. For radios that form part of a System-on-Chip, a dedicated radio interrupt is asserted when a frame has been received in full without errors. For SPI radios, frame reception triggers a GPIO interrupt. The corresponding interrupt handler typically polls the radio driver's main process and returns immediately, thus limiting the time spent executing code instructions inside an interrupt context. The polled radio driver process is subsequently called outside the interrupt context, reads the frame from the radio hardware's buffer and places it in main RAM, alongside relevant frame reception attributes such as the Received Signal Strength Indicator (RSSI). In the case of TSCH, radio interrupts are disabled. Using the `NETSTACK_RADIO` API, the platform-independent TSCH implementation polls the radio driver for received frames at the correct time during timeslot operation, thus ensuring the timing accuracy required by TSCH. Frames get processed by the implementation of the respective protocol as they are handed upwards the network stack, and the application code gets notified by way of a function callback.

Under the `/examples` directory, Contiki-NG provides numerous example projects that can be used as first steps with Contiki-NG, or as a starting point by users who wish to develop their own application. These examples cover all elements of the network stack described in this paper, as well as all hardware abstractions. To try out the example in this section, or any of the other existing examples on the repository, we recommend using the Contiki-NG Docker container image[19] and accompanying wiki documentation. This image provides all necessary compilers and tools, including the Cooja simulator that allows experimentation without access to Contiki-NG supported hardware.

[19] https://hub.docker.com/r/contiker/contiki-ng

4. Impact

Since its open source release in 2006, the original Contiki OS has been used by numerous research projects funded by a host of organisations, for example: (i) The European Commission (EC) under Horizon Europe, Horizon 2020 (H2020), as well as by previous framework programmes, (ii) Various national research funding bodies, such as the UK's Engineering and Physical Sciences Research Council (EPSRC) or the Swedish Knowledge Foundation.

The added value and benefits of Contiki-NG can be summarised as:

- Simplified porting to new hardware platforms due to removal of legacy code, and due to new, platform-independent system initialisation, main loop code and HALs.
- Simplified development of new features due to improved documentation and cleaner examples.
- Increased code quality due to modern development practices including the *git-flow* workflow, continuous integration, nightly testbed runs, and a script to test in excess of 1200 project builds for all platforms.

Due to these features, Contiki-NG is both an off-the-shelf tool for building multihop, low-power, constrained wireless networks with five-nines reliability [19], as well as a research platform for innovation in low-power wireless embedded systems at all levels of the networking stack, including for example 6TiSCH scheduling, routing, security, and power-saving MAC layers.

Despite its relatively short history, Contiki-NG has already facilitated the research published in a number of peer-reviewed papers, including ones authored by teams not affiliated with the project, for example [6,27–31].

Contiki-NG has been particularly successful in enabling novel research in IEEE 802.15.4 TSCH and IETF 6TiSCH networks [30–32]. Contiki / Contiki-NG offers one of only two open source IEEE 802.15.4 TSCH implementations for real hardware, the other one offered by OpenWSN [33]. The authors of this paper do not consider OpenWSN as a direct competitor, as it is a network stack, not a complete operating system.

Contiki-NG enables future research in many directions, including but not limited to IoT security; energy harvesting; network mobility; multi-protocol/multi-radio/multi-frequency band IoT communications.

Beyond scientific publications, Contiki-NG is starting to make an impact through getting used: (i) to support the work undertaken as part of funded research projects in multiple disciplines (e.g. Industrial IoT, digital health, smart cities), (ii) for teaching in higher education.

Table 2 lists a small number of funded R&D projects that have made extensive use of Contiki-NG. Presenting a comprehensive list of such projects would be prohibitively long and is considered out of scope of this paper.

Contiki-NG is used as the basis of multiple commercial products. This list includes consumer/home applications, for example smart home heating systems and smart light bulbs. The list also includes industrial applications, such as asset monitoring/tracking, and smart agriculture.

5. Conclusions

This paper has discussed the Contiki-NG operating system for severely-constrained wireless embedded devices. It has provided the reader with an overview of the project's architecture and some of its key non-technical features. It has highlighted the value that Contiki-NG adds to the research community compared to its predecessor. Lastly, it has provided some indicators about its impact on research and higher education.

Table 2
Funded research projects using Contiki-NG.

Project title	Funder	Ref.
F-Interop	H2020	
5G-CORAL	H2020	[34]
SPHERE	UK EPSRC	
EurValve	H2020	[32]
Vessedia	H2020	[35]
E-care@home	Swedish Knowledge Foundation	[36]
aSSIsT	Swedish Foundation for Strategic Research	
SYNERGIA	Innovate UK	

Contiki-NG comes with a roadmap that can be accessed directly on GitHub. Updates that are currently in the pipeline are labelled with "`roadmap`", while features in the project's longer-term wish list are labelled with "`roadmap/long-term`"[20]

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors are the individuals among the current group of Contiki-NG maintainers who were available to contribute to the preparation of this manuscript. While the authors have written a very significant proportion of the source code, we do not claim ownership of the entire codebase. Over the years, source code has been developed and contributed by in excess of 250 individuals, including hobbyists, students, researchers and industry professionals. The GitHub "contributors" page[21] provides a list of contributors within the last 10 years, but the full code history dates back to 2006.

With that in mind, first and foremost we gratefully acknowledge individuals who have contributed their work to Contiki-NG and the original Contiki OS since its first open source release in 2006. We acknowledge Adam Dunkels – the inventor of the original Contiki OS – and all individuals who have acted as project maintainers over the years.

This work has been partially supported by VINNOVA and the Swedish Foundation for Strategic Research through the aSSIsT project.

References

[1] Akyildiz IF, Su W, Sankarasubramaniam Y, Cayirci E. Wireless sensor networks: a survey. Comput Netw 2002;38(4):393–422.
[2] Levis P, Madden S, Polastre J, Szewczyk R, Whitehouse K, Woo A, et al. TinyOS: An operating system for sensor networks. In: Ambient intelligence. Springer; 2005, p. 115–48.
[3] Dunkels A, Gronvall B, Voigt T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In: 29th Annual IEEE international conference on local computer networks. 2004, p. 455–62.
[4] Dunkels A, Schmidt O, Voigt T, Ali M. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In: Proceedings of the 4th international conference on embedded networked sensor systems. New York, NY, USA: ACM; 2006, p. 29–42.
[5] Dunkels A. Full TCP/IP for 8-bit architectures. In: Proceedings of the 1st international conference on mobile systems, applications and services. MobiSys '03, New York, NY, USA: ACM; 2003, p. 85–98.
[6] Kim H-S, Andersen MP, Chen K, Kumar S, Zhao WJ, Ma K, et al. System architecture directions for post-soc/32-bit networked sensors. In: Proceedings of the 16th ACM conference on embedded networked sensor systems. ACM; 2018, p. 264–77.
[7] Baccelli E, Hahm O, Günes M, Wählisch M, Schmidt TC. RIOT OS: TOwards an OS for the internet of things. In: 2013 IEEE conference on computer communications workshops. IEEE; 2013, p. 79–80.
[8] Javed F, Afzal MK, Sharif M, Kim B-S. Internet of things (IoT) operating systems support, networking technologies, applications, and challenges: A comparative review. IEEE Commun Surv Tutor 2018;20(3):2062–100.
[9] Silva M, Cerdeira D, Pinto S, Gomes T. Operating systems for internet of things low-end devices: Analysis and benchmarking. IEEE Internet Things J 2019;6(6):10375–83.
[10] IEEE. 802.15.4-2015 - IEEE Standard for low-rate wireless networks. 2016, IEEE Std 802.15.4-2015.
[11] Thubert P. An architecture for IPv6 over the time-slotted channel hopping mode of IEEE 802.15.4 (6tisch). 2021, RFC 9030.
[12] Vilajosana X, Pister K, Watteyne T. Minimal IPv6 over the TSCH mode of IEEE 802.15.4e (6tisch) configuration. 2017, RFC 8180.
[13] Wang Q, Vilajosana X, Watteyne T. 6TiSCH operation sublayer (6top) protocol (6P). 2018, RFC 8480.
[14] Alexander R, Brandt A, Vasseur J, Hui J, Pister K, Thubert P, et al. RPL: IPV6 routing protocol for low-power and lossy networks. 2012, RFC 6550.
[15] Duquennoy S, Elsts A, Al Nahas B, Oikonomou G. TSCH And 6tisch for contiki: challenges, design and evaluation. In: Proc. IEEE DCOSS. IEEE; 2017, p. 11–3.
[16] Duquennoy S, Nahas BA, Landsiedel O, Watteyne T. Orchestra: Robust mesh networks through autonomously scheduled TSCH. In: Proceedings of the international conference on embedded networked sensor systems. Seoul, South Korea; 2015.
[17] Chang T, Vučinić M, Vilajosana X, Duquennoy S, Dujovne D. 6TiSCH minimal scheduling function (MSF). 2021, RFC 9033.
[18] Ko J, Eriksson J, Tsiftes N, Dawson-Haggerty S, Terzis A, Dunkels A, et al. ContikiRPL and TinyRPL: Happy together. In: Workshop on extending the internet to low power and lossy networks (IP+ SN). Vol. 570. Citeseer; 2011.
[19] Duquennoy S, Eriksson J, Voigt T. Five-nines reliable downward routing in RPL. 2017, arXiv.
[20] Hui JW, Kelsey R. Multicast protocol for low-power and lossy networks (MPL). 2016, RFC 7731.
[21] Oikonomou G, Phillips I. Stateless multicast forwarding with RPL in 6lowpan sensor networks. In: Proc. IEEE international conference on pervasive computing and communications workshops. Lugano, Switzerland: IEEE; 2012, p. 272–7.
[22] Oikonomou G, Phillips I, Tryfonas T. IPV6 multicast forwarding in RPL-based wireless sensor networks. Wirel Pers Commun 2013;73(3):1089–116.
[23] Abdel Fadeel KQ, El Sayed K. ESMRF: Enhanced stateless multicast RPL forwarding for IPv6-based low-power and lossy networks. In: Proc. 2015 workshop on IoT challenges in mobile and industrial systems. IoT-Sys '15, New York, NY, USA: ACM; 2015, p. 19–24.
[24] Shelby Z, Hartke K, Bormann C. The constrained application protocol (CoAP). 2014, RFC 7252.
[25] Bormann C, Shelby Z. Block-wise transfers in the constrained application protocol (CoAP). 2016, RFC 7959.
[26] Hartke K. Observing resources in the constrained application protocol (CoAP). 2015, RFC 7641.
[27] Tomasic I, Khosraviani K, Rosengren P, Jörntén-Karlsson M, Lindén M. Enabling IoT based monitoring of patients' environmental parameters: Experiences from using OpenMote with openwsn and contiki-NG. In: 2018 41st International convention on information and communication technology, electronics and microelectronics. IEEE; 2018, p. 0330–4.
[28] Algora CMG, Reguera VA, Fernández EMG, Steenhaut K. Parallel rendezvous-based association for IEEE 802.15.4 tsch networks. IEEE Sens J 2018;18(21):9005–20.
[29] Yang G, Urke AR, Øvsthus K. Mobility support of IoT solution in home care wireless sensor network. In: 2018 Ubiquitous positioning, indoor navigation and location-based services. IEEE; 2018, p. 475–80.
[30] Cheng X, Sha M. Cracking the TSCH channel hopping in IEEE 802.15.4e. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. ACM; 2018, p. 2210–2.

20 https://github.com/contiki-ng/contiki-ng/issues?q=is%3Aopen+is%3Aissue+label%3Aroadmap
21 https://github.com/contiki-ng/contiki-ng/graphs/contributors

[31] Lee S-B, Kim E-J, Lim Y. Contiki-NG-based IEEE 802.15.4 TSCH throughput evaluation. In: Proceedings of the Korean institute of information and commucation sciences conference. The Korea Institute of Information and Commucation Engineering; 2018, p. 577–8.
[32] Elsts A, Pope J, Fafoutis X, Piechocki R, Oikonomou G. Instant: A TSCH schedule for data collection from mobile nodes. In: Proc. 2019 international conference on embedded wireless systems and networks. EWSN, 2019.
[33] Watteyne T, Vilajosana X, Kerkez B, Chraim F, Weekly K, Wang Q, et al. OpenWSN: a standards-based low-power wireless development environment. Trans Emerg Telecommun Technol 2012;23(5):480–93.
[34] Li C-Y, Chien H-T, editors. Communication, dissemination, standardisation and exploitation achievements of Y1 and plans for Y2. 2018, 5G-CORAL Deliverable D5.1.
[35] Peyrard A, Kosmatov N, Duquennoy S, Raza S. Towards formal verification of contiki: Analysis of the AES–ccm* modules with frama-c. In: Proc. workshop on recent advances in secure management of data and resources in the IoT. RED-IOT, Madrid, Spain; 2018.
[36] Alirezaie M, Renoux J, Köckemann U, Kristoffersson A, Karlsson L, Blomqvist E, et al. An ontology-based context-aware system for smart homes: E-care@home. Sensors 2017;17(7):1586.