International Journal of Enhanced Research in Science Technology & Engineering, ISSN: 2319-7463
Vol. 3 Issue 10, October-2014, pp: (95-101), Impact Factor: 1.252, Available online at: www.erpublications.com

Low Power Memory Efficient Contiki Operating System for Wireless Sensor Networks Based Internet of Things

Harsh Gupta[1], Neenu Preetam. I[2]
[1,2]M. Tech. (Microelectronics), Department of ECE, SEEC, Manipal University Jaipur (MUJ), Rajasthan

Abstract: In this paper we introduce to the current State-of-the-art in Wireless Sensor Network (WSN) Operating Systems (OS). Contiki, an open source, highly portable for networking, memory-constrained multi-tasking operating system has a particular focus on low-power wireless Internet of Things (IoT). Contiki has a built in TCP/IP stack with a graphical user interface which needs only 10 kilobytes of RAM and 30 kilobytes of ROM memory footprint (such as 6LoWPAN) using IPv4 and IPv6 (for large address space) communication protocols. Contiki is based upon an event-driven kernel which provides support for both multi-threading and lightweight stackless protothreads. IoT is characterized by lightweight heterogeneous energy efficient sensors powered by MCUs and with minimal resources to reduce the power consumption for unattended operation by using solar means to recharge batteries. Contiki has an impact over IoT that can be used in street lighting systems, sound monitoring for smart cities, industrial process monitoring, and home automation. Our evaluation was performed in Low power hop based wireless Modular Sensor platform for running Contiki on devices, named as GreenNet. Such devices will have reliability, real time behavior, and an adaptive communication stack to integrate the Internet seamlessly.

Keywords - WSN, Contiki OS, IoT, 6LoWPAN, IPv4, IPv6, TCP/IP, MCUs

I. INTRODUCTION

Wireless sensor networks (WSNs) are unique networked systems, composed of wireless sensor nodes. WSNs distinguish themselves from other traditional wireless networks by relying on extremely constrained resources such as energy, bandwidth, and the capability to process and store data [1]. The number of applications that make use of WSNs is constantly increasing. The range of these applications goes from military and societal security to habitat and environment monitoring. For many applications it is essential to provide secure communications. In general, WSNs face the same security risks as conventional wired or wireless networks; eavesdropping, packet injection, replay and denial of service attacks are some of the common attacks in WSNs. Due to the inherent properties of sensor nodes, traditional security protocols are not suitable for WSNs. Figure 1 depicts the Sensor nodes scattered in a sensor field.

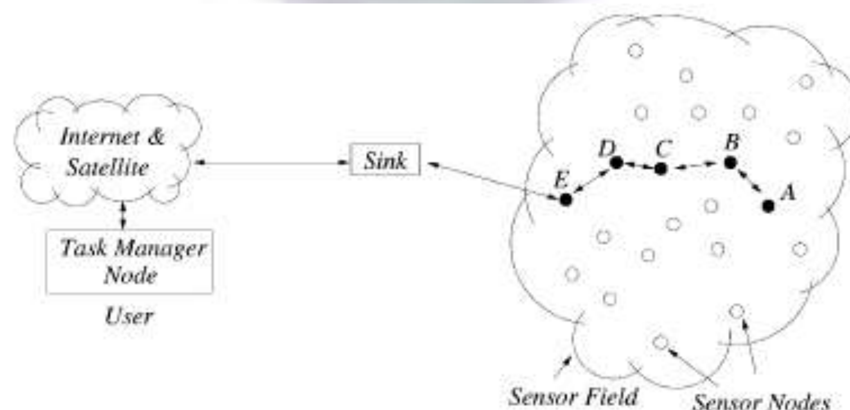

Figure 1: Sensor nodes scattered in a sensor field [1]

International Journal of Enhanced Research in Science Technology & Engineering, ISSN: 2319-7463
Vol. 3 Issue 10, October-2014, pp: (95-101), Impact Factor: 1.252, Available online at: www.erpublications.com

In 2004 Contiki was presented. Contiki is an open source, highly portable, multitasking operating system for memory-efficient networked embedded systems and wireless sensor networks [6]. Contiki has become quite popular and is attaining a good position in the WSNs community. Contiki does not offer any confidentiality or authenticity services for the communication between nodes. It was developed at the Swedish Institute of Computer Science by Dunkels et al. [6]. It's main features are dynamic loading and unloading of code at run time and the possibility of multi-threading atop of an event driven kernel, which are discussed in sec. 2.1 and sec. 2.2 respectively. Its current version is 2.5 released on September 12, 2011. The current version provides only cyclic redundancy checks (CRC-16) to achieve integrity [2]. Figure 2 shows the Wireless Sensor Data Acquisition and Distribution Network.

The rest of the paper is organized as follows. In the next section, we describe the Contiki's Architecture through a Programming and Execution model. In Section 3 we describe the Contiki OS for Internet of Things (IoT). Finally, the paper is concluded in Section 4.

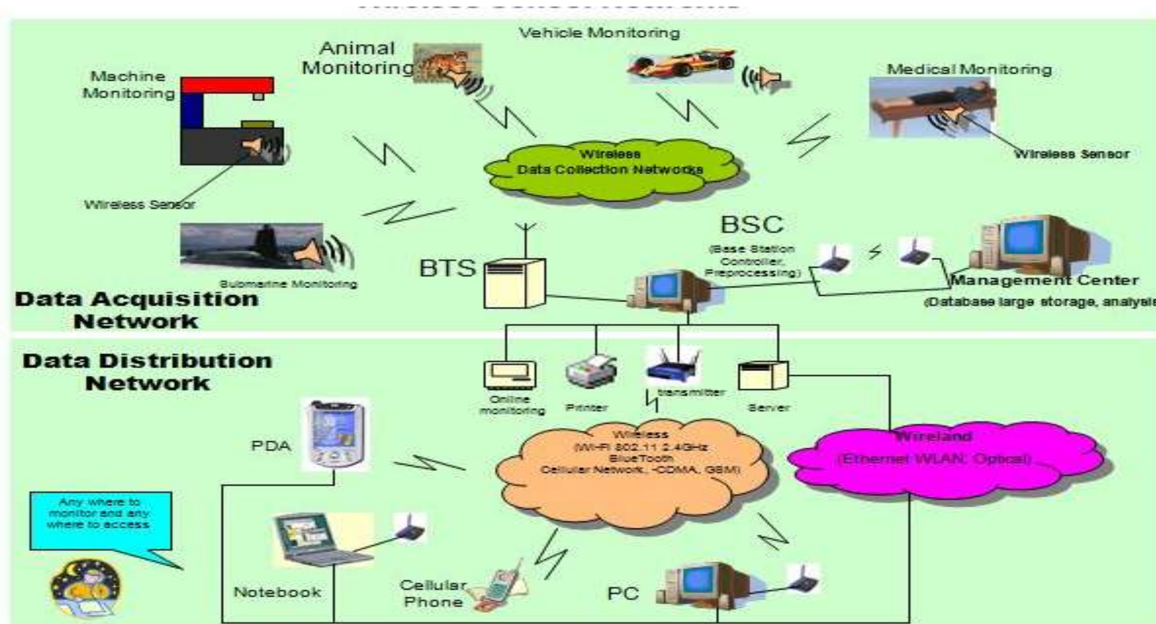

Figure 2: Wireless Sensor Networks [2]

II. ARCHITECTURE OF CONTIKI OS

A Contiki application consists of the Contiki kernel, libraries, the program loader and processes. Processes are either services or an application program. The difference between services and application programs is, that the functionality of services can be used by more than one other process, while application programs only use other processes and do not offer functionality for different other processes[6].

2.1 Programming Model

Each of the processes must implement an event handler function and can optionally implement a poll handler function. Processes can be executed only through these handlers. Every process has to keep its state information between calls of these functions, since the stack gets restored after the return from these functions[6].

One of the special features of Contiki is the ability to replace all programs dynamically at run-time. To accomplish that Contiki offers a run-time relocating function. This function can relocate a program with the help of relocation information that is present in the program's binary. After the program is loaded the loader executes its initialization function where one or more processes can be launched[6].

A running Contiki system consists of the operating system core and the loaded programs. The core typically consists of the kernel, different libraries and drivers and the program loader (See Figure 3). The core usually is deployed as one single binary while the loaded programs each can be distributed independently[6].

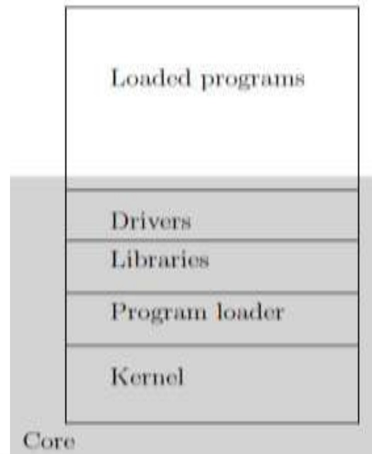

Figure 3: The partitioning of the Contiki core and the loaded programs [3]

The Contiki kernel offers no hardware abstraction. If hardware abstraction is desired libraries and/or drivers have to implement it themselves. All components of a Contiki application have direct access to the underlying hardware [3].

2.2 Execution Model

The Contiki kernel is event-driven. Processes can only be executed by the scheduler when it either dispatches an event to the event handler of the process or by calling the polling handler. While events always have to be signaled by a process, the scheduler can be configured to call the polling handlers of all processes that implement one in periodic intervals between the dispatching of events. Both the event handlers and the polling handlers are not preempted by the scheduler and therefore always run to completion. A Contiki Protothread example:

```
// led_blink . c
PROCESS( blink_process , ”blink_example ” ) ;
AUTOSTART_PROCESSES(&blink_process ) ;
PROCESS_THREAD( blink_process , ev , data )
  {
   PROCESS_BEGIN( ) ;
   leds_off (LEDS_ALL) ;
   static struct etimer et ;
   while (1) {
     etimer_set (&et , CLOCK_SECOND) ;
     PROCESS_WAIT_EVENT( ) ;
     leds_toggle (LEDS_GREEN) ;
   }
  PROCESS_END( ) ;
}
```

There are two kinds of events in Contiki: Asynchronous and synchronous events. When an asynchronous event is signaled the scheduler enqueues the event and will call the corresponding event handler after all currently enqueued events were processed. Synchronous events on the other hand are more like inter-process procedure calls: The scheduler immediately calls the corresponding event handler and returns the control to the calling process after the event handler has finished running [4]. Figure 4 demonstrates the Python Testbed for Server and Client interaction on two different machines. First, the server opens its mport 5000 for listening to the client. The server runs the client side script on the client-PC using ssh login. The client side script listens to port 5000 of the server. Eventually the server writes the commands on port 5000. The client picks up the command by opening a serial port and sends the command to the sensor node (mote). In acknowledgement, the client listens to the node response and sends it to the server on port 5000.

Figure 4: Python Testbed for Server-Client Interaction

While event handler cannot preempt each other, interrupts can of course preempt the current running process. To prevent race-conditions from happening, events cannot be posted from within interrupt handlers. Instead they can request the kernel to start polling at the next possible point in time.

On top of this basic event-driven kernel other execution models can be used. Instead of the simple event handlers processes can use Protothreads[7]. Protothreads are simple forms of normal threads in a multi-threaded environment. Protothreads are stackless so they have to save their state information in the private memory of the process. Like the event handler Protothreads cannot be preempted and run until the Protothread puts itself into a waiting state until it is scheduled again[7].

Contiki also comes with a library that offers preemptive multi-threading on top of the event-driven kernel. The library is only linked with the program if an application explicitly uses it. In contrast to Protothreads this multi-threading approach requires every thread to have its own designated stack[6]. Figure 5 shows the interaction of sensor nodes through Cooja - the Network Simulator for Contiki OS [9].

Both multi-threading approaches were introduced because modeling application in the event-driven approach can be very complex depending on the specific requirements of the program. The event-driven execution model requires in most cases the implementation of a state machine to achieve the desired behavior, even if the programmer may not be aware of that. Dunkels et al. suggest in [7] that the code size of most programs can be reduced by one third and most of the state machines could entirely removed by using Protothreads. While the overhead in execution time is minimal there is a moderate increase in the program memory required by this approach.

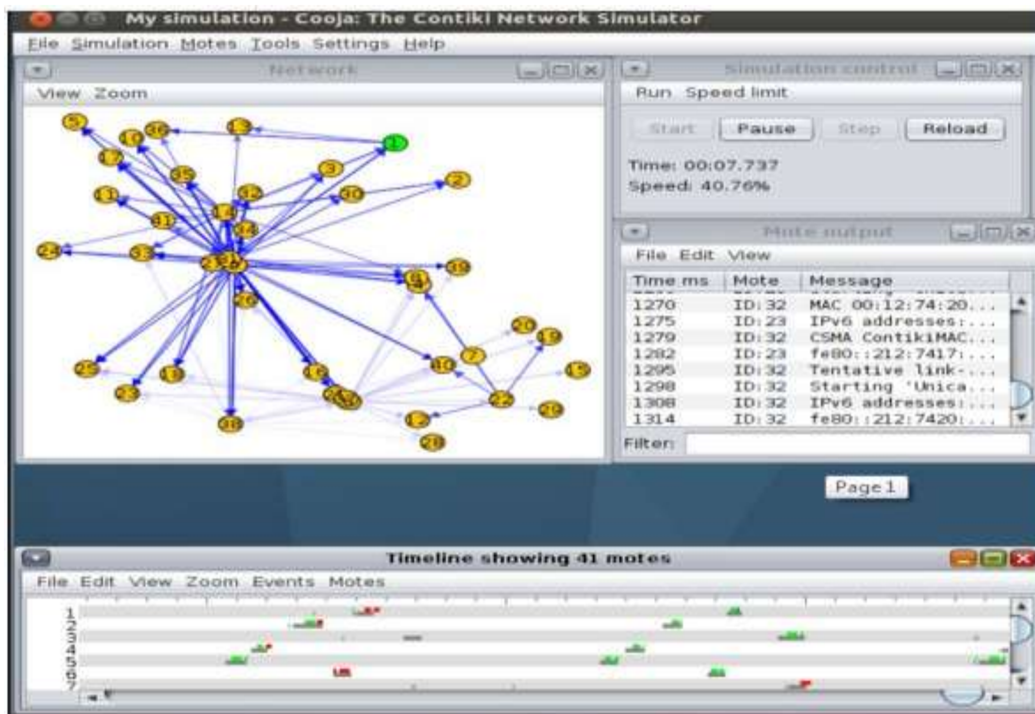

Figure 5: Cooja - The Contiki Network Simulator [4]

2.3 Resource Use

Since the scheduler and the kernel in general are more complex in Contiki and the possibility of dynamically load processes, which doesn't allow cross boundary optimization, the required program memory and execution time for Contiki programs is higher. When using the preemptive multi-threading library the RAM usage will be higher than using only the event-driven kernel for scheduling [6].

2.4 Energy Consumption

The Contiki operating system offers no explicit power saving functions. Instead things like putting the microcontroller or peripheral hardware in sleep modes should be handled by the application. For that matter the scheduler exposes the size of the event queue so that a power saving process could put the CPU in sleep mode when the event queue is empty.

2.5 Hardware Platforms

Contiki has been ported to a number of mote platforms on basis of different microcontrollers. Supported microcontroller include the Atmel AVR, the Texas Instruments MSP430 and the Zilog Z80 microcontrollers [6]. Porting Contiki requires to write the boot up code, device drivers and parts of the program loader. If the multithreading library is used, its stack switching code has to be adapted. The kernel and the service layer are platform independent.

2.6 Toolchain

Contiki is written in plain C so a native C compiler for the target platform can be used [5].
Figure 6 demonstrates the MultiNode Setup (MNS) with 1 Edge Router (ER) and 2 Leaf Nodes (LF). Here, the server and client works independently on a single machine. Firstly, the server sends the commands to the client side over port 5000 (through socket communication). The client sends the commands to first leaf node (LF-1) in a duty cycled fashion. It looks for the Radio blinking (RBB) and connection setup (CON). The client sends the RBB and CON commands to the second leaf node (LF-2) also in a similar fashion. In response, LF-1 and LF-2 sends the UDP packet to Gateway (GW) using port number 3000. Since ER is the parent of all connected Leaf Node(s), the traffic passes through Edge Router. ER sends the packets through serial USB and mport routes them into the IPv6 tunnel. A Python script running on the Host PC continuously listens to the mport log. Whenever a packet is received, the python script opens the port 5000 of the server side and writes the data.

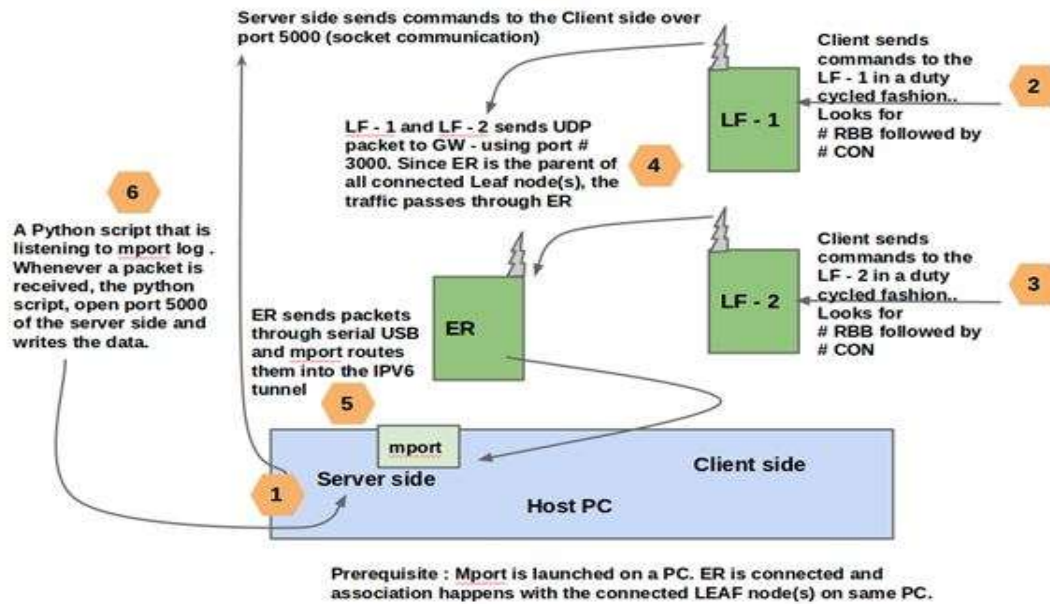

Figure 6: MultiNode Setup (MNS) with 1 Edge Router (ER) and 2 Leaf Nodes (LF)

III. CONTIKI OS FOR INTERNET OF THINGS

The Internet of Things (IoT), a technology that enables the automated exchange of key information between machines, and then ultimately to humans, promises a future where a whopping 50 billion devices will have the ability to "talk to each other". The 'things' that are getting connected vary significantly in size and shape that includes tiny wireless sensors (smart dust) to electrical appliances at home to electronics used in space experiments. Things are characterized as low-memory and low-power devices, where memory is measured in bits or bytes (not megabytes or gigabytes) and power consumption is so low that devices can run for years with a pair of AAsize battery! [7]. See Figure 7. Things are also referred to as smart objects which collect valuable data from the surroundings and communicate to the networked system. Thereby smart objects become a bridge between the real and virtual worlds. They contain sensors, microprocessor, communication and power source. The operating system (OS) in these objects controls all of its operations [10].

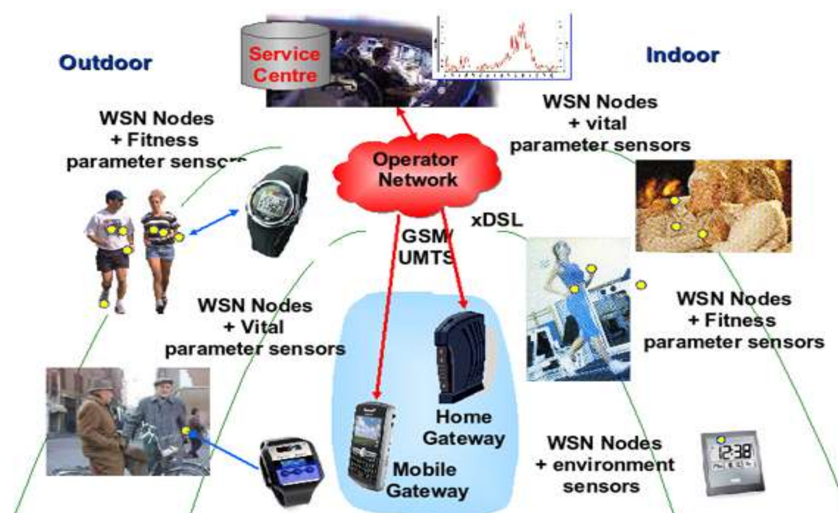

Figure 7: Internet of Things (IoT) [9]

The fundamental piece for that is the OS such as Contiki. Just like most consumer electronics devices come with chips embedded in them, for Internet of Things it is essential that there is some basic software such as Contiki running in them. IoT refers to linking the Internet to physical objects: Equipping objects, such as household appliances, with networking capabilities, allowing mutual interaction between them and greater monitoring and control over them by end users. IoT applications have traditionally required deep expertise in electronics design, embedded systems software, low-power wireless communication, networking protocols, Internet server software, and smartphone app development [8]. Contiki is found in "typical types" of IoT systems, including smart grids, smart buildings and homes, and smart streetlights. It is also used as a wireless radioactive radiation detector by D-Tect systems that uses Contiki to allow the detectors to form self-healing wireless networks to transport radiation readings.

This allows programs to use blocking waits, without having to manually be turned into state machines. The trade-off is that programs must block only at the top level, but this is enough for the kind of programs we find in IoT devices that typically do only a set of relatively simple tasks [9]. One of the priorities is to create better APIs to make development easier and simplify the process of porting the OS to new platforms. The Internet of Things will ultimately have a profound impact on people's lives. Tomorrow, the Internet of Things will mean that we will have instant access to not just information, but literally everything around us, just as we have access to information on the Internet today [10].

IV. CONCLUSION

The Internet of Things (IoT) allows us to build apps that make the real world a little more like the Internet. The Internet of Things builds on a range of technologies: Low-power radios, routing protocols, sleek software. This allows items, environments, places, and devices to be directly connected to the Internet and exchange information. Both obtaining information from the Internet, for example to better control heating and lighting, and pushing information to the Internet; for example providing information of where people are moving around in a city. The Contiki operating system has been built to let battery-operated, low-power systems connect to the Internet. Contiki added an operating system layer on top of uIP and added IPv6 support, which made it possible to connect multiple applications to the Internet simultaneously. All protothreads run on the same stack, while retaining linear code flow.

V. REFERENCES

[1]. P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer TinyOS: An operating system for sensor networks In Ambient intelligence, p. 115-148, Springer, Berlin, 2005.

[2]. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, K. S. J. Pister System architecture directions for networked sensors In SIGPLAN Not. 35 (11), p. 93–104, ACM, 2000.

[3]. D. Gay, P. Levis, R. von Behren, M.Welsh, E. Brewer, and D. Culler The nesC language: A holistic approach to networked embedded systems In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI), ACM, 2003.

[4]. P. Levis, N. Lee, M. Welsh, D. Culler TOSSIM: Accurate and scalable simulation of entire TinyOS applications In Proceedings of the 1st international conference on Embedded networked sensor systems, p. 126-137, ACM, 2003.

[5]. Dunkels A., Österlind F., Tsiftes N., He Z.: Software-based Sensor Node Energy Estimation. In: Proceedings of the 5th international conference on Embedded networked sensor systems (SenSys 2007), pp.409--410. ACM, New York (2007).

[6]. [6] A. Dunkels, B. Gr¨onvall, T. Voigt Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors In Proceedings of the First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, USA, 2004.

[7]. A. Dunkels, O. Schmidt, T. Voigt, M. Ali Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems In Proceedings of the Forth International Conference on Embedded Networked Sensor Systems, p. 29-42, ACM, 2006.

[8]. A. Dunkels, N. Finne, J. Eriksson, T. Voigt Run-time dynamic linking for reprogramming wireless sensor networks In Proceedings of the 4th international conference on Embedded networked sensor systems, p. 15-28, ACM, 2006.

[9]. http://internetofthings.electronicsforu.com/2013/04/contiki-the-operating-system-for-internet-of-things/.

[10]. http://www.computerworld.com.au/article/438327/contiki_an_operating_system_internet_things_/.