Comparison of Operating Systems TinyOS and Contiki

Tobias Reusing
Betreuer: Christoph Söllner
Seminar: Sensorknoten - Betrieb, Netze & Anwendungen SS2012
Lehrstuhl Netzarchitekturen und Netzdienste, Lehrstuhl Betriebssysteme und Systemarchitekturen
Fakultät für Informatik, Technische Universität München
Email: reusing@in.tum.de

ABSTRACT

Wireless sensor networks can be used in a lot of different application areas. Since such networks were first proposed, many different node platforms both in regard to hardware and software were introduced. In this paper we present operating systems for wireless sensor nodes in general and two of the best known operating systems, TinyOS and Contiki with their most notable differences and similarities. Both have strengths and weaknesses which are important for various requirements of wireless sensor networks.

Keywords

TinyOS, Contiki, Operating Systems, Sensor Nodes, Wireless Sensor Networks

1. INTRODUCTION

Wireless sensor networks consist of a huge number of single nodes. They can be used for a wide range of applications. For all different application areas different requirements have to be fulfilled. While in one application it may be most important that the nodes can operate unattended for very long periods of time, in another application they may have to be able to process huge amounts of data in short time frames. Therefore it is of high importance to choose the right hard and software components for the particular application. The operating system is one of the most important parts on the software side of this decision process. There are a lot of sensor node operating systems available and at first glance it is not always easy to determine which one is better suited for which applications. Therefore this paper presents two of the best known sensor node operating systems and compares them according to different requirements.

Section 2 of this paper presents operating systems for wireless sensor networks in general. Section 3 and section 4 give an overview of the features of TinyOS and Contiki respectively. In section 5 both operating systems are compared according to requirements for wireless sensor nodes and section 6 presents the conclusions.

2. OPERATING SYSTEMS FOR WIRELESS SENSOR NETWORKS

Operating systems that are designed for wireless sensor networks are very different from operating systems for desktop/laptop computers like Windows or Linux or operating systems for powerful embedded systems like smart phones. The biggest difference is the hardware on which the operating systems are running. The wireless sensor nodes (often called motes) usually have a microcontroller as a CPU that is not very powerful because the main focus of those motes lies in minimal power consumption since they are often designed to run on battery power for very long periods of time. And even though the microcontroller and all other components of motes are designed as low power devices, running them all at full power at all times would still consume way too much energy. So for that matter the main focus of those operating systems is energy conservation optimal usage of limited resources[1].

Operating systems for motes are very simple compared to other operating systems. But they still are often required to handle many different operations at the same time. A mote could for example be required to collect data from a sensor, process the data in some way and send the data to a gateway at the same time. Since the microcontrollers are only able to execute one program at the time, the operating systems have to have a scheduling system that shares the CPU resources between the different tasks, so that all of them can finish in the desired time frame.

Since the requirements for the operating system vary between applications it is in most cases not possible to exchange the client program on a mote without changing the operating system. In fact in most cases the operating system behaves more like a library: It gets integrated into the application and both the application and the operating system are compiled into one single binary that is then deployed onto the sensor node.

In summary the main requirements for an operating system for sensor networks are[1]:

- Limited resources: The hardware platforms offer very limited resources so the operating system should use them efficiently.
- Concurrency: The operating system should be able to handle different tasks at the same time.
- Flexibility: Since the requirements for different applications vary wildly, the operating system should be able to be flexible to handle those.
- Low Power: Energy conservation should be one of the main goals for the operating system.

3. TINYOS

TinyOS was developed at the University of California in Berkeley[2] and is now maintained as an open source project by a community of several thousand developers and users lead by the TinyOS Alliance[12]. The current Version of TinyOS from April 6, 2010 is 2.1.1.

TinyOS uses an event driven programming model and concurrency is achieved with non-preemptive tasks[1]. TinyOS programs are organized in components and are written in the NesC language[3], a dialect of C.

3.1 Programming Model

As already mentioned, TinyOS user applications and the operating system itself are composed of components. Components offer three types of elements: Commands, Events and Tasks [1]. All three are basically normal C functions but they differ significantly in terms of who can call them and when they get called. Commands often are requests to a component to do something, for example to query a sensor or to start a computation. Events are mostly used to signal the completion of such a request.

Tasks, on the other hand, are not executed immediately. When a task is posted, the currently running program will continue its execution and the posted task will later be executed by the scheduler (see section 3.2 for details).

Components expose which commands they can send and which events they can handle through interfaces. An interface consists of a number of commands and events that are specified by their function signatures. A component can either specify to use an interface or to provide it. Components that provide an interface have to implement all of the specified commands and can signal the specified events. Components that, on the other hand, use an interface have to implement all of the specified events and can use all of the commands [4, p. 26] (cf. Figure 1).

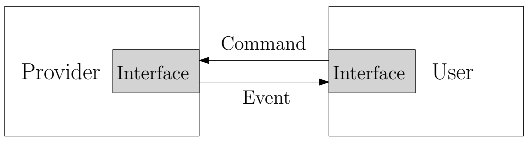

Figure 1: An interface specifies which commands the user of the interface can call and which events the provider can signal

The connections between components are specified in so-called configurations. A configuration defines for each interface of a component which other component uses the provided interfaces and which component provides the used interfaces. These connections are called wirings[4, p. 31]. Configurations are itself components, which means that they can also provide and use interfaces. This makes it possible to build TinyOS applications in a hierarchical manner where components on a higher level are made up of several components of a lower level (cf. Figure 2).

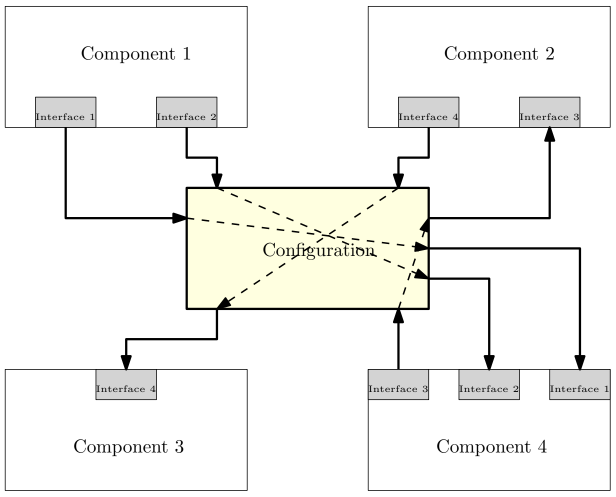

Figure 2: Configurations map the exposed interfaces of components onto each other

TinyOS programs and the operating system itself are written in NesC[3]. NesC is a dialect of C. It incorporates all of the concepts of TinyOS, like components, interfaces, commands, events, tasks, configurations etc. into a C like language. The biggest difference between C and NesC is how the function to be executed is selected. In C the function to be executed at a function call is selected by its name. In NesC when a command or event should be executed the programmer explicitly selects with the wiring in configurations which component's implementation of the command or event should be used[4, p. 13]. But "normal" C functions can still be used in NesC and to differentiate between them and commands, events and tasks special keywords are used for invoking each of the non-C function types. C libraries and C preprocessor directives can be used[4, p. 46-47]. NesC is also designed to allow whole-program analysis which allows the detection of data-race conditions which can improve reliability and it can help with inlining across component border which can reduce resource consumption[3].

3.2 Execution Model

TinyOS uses a split-phase execution model[1]. This means that the request to do an operation and the response after completion are decoupled. This approach resembles how interaction with hardware in a sensor node often works: The micro controller sends a request to a hardware component, which then works on it independently from the controller and later signals the completion of the task through an interrupt. In TinyOS both hardware and software modules follow this split-phase execution model, which is represented in the programming model: Both are components that can handle commands and at a later time signal events after the completion of the operation.

Concurrency in TinyOS is achieved with tasks. Tasks are basically functions that can be posted by other tasks or interrupt handlers[1]. They don't get executed immediately but instead will later be executed by the scheduler. The TinyOS scheduler executes one task after another and so tasks can never preempt each other. Each task runs until completion and the next task is started after that. This

```
// BlinkC.nc - Blink module
module BlinkC {
  uses interface Boot;
  uses interface Timer;
  uses interface Leds;
}
implementation {
  event void Boot.booted() {
    call Timer.startPeriodic(1000);
  }

  event void Timer.fired() {
    call Leds.led0Toggle();
  }
}
```

```
// BlinkAppC.nc
configuration BlinkAppC { }
implementation {
  components MainC, LedsC, TimerC, BlinkC;

  BlinkC.Boot -> MainC.Boot;
  BlinkC.Leds -> LedsC.Leds;
  BlinkC.Timer -> TimerC.Timer;
}
```

Listing 1: Minimal example of a TinyOS application that turns a LED on and off every second. With modifications from [4]

makes it possible that all tasks can use the same stack, which saves memory because not every task needs it's own designated stack like in a multi-threaded approach (see section 3.3 for details). Normally tasks are executed in a first-in-first-out (FIFO) order, so tasks run in the same order they're posted[4], but it is possible to implement other (more complex) scheduling strategies[2]. If there is no more task in the queue after completion of the previous task, TinyOS sets the mote into a low-power sleep state until an interrupt wakes the microcontroller[4].

Code in TinyOS can only be executed in the context of either a task or an interrupt. Interrupts can preempt tasks and other interrupt handlers of lower priority. Code that is reachable from an interrupt handler is called asynchronous code and special measures have to be taken to handle concurrency issues that can occur because of that (see Figure 3 or [4, p. 192 ff] for further details).

Since tasks in TinyOS can not be preempted, long running tasks, for example tasks that handle complex computations like cryptographic functions, will block the whole application and tasks that are time critical, like tasks handling sending and receiving of data over a communication medium, may be waiting for too long before the scheduler executes them. In those cases the long running computation should be split up in shorter running tasks that post themselves after completing a part of the computation, which enables other tasks to run in between them. Since posting and executing a task generates a overhead of about 80 clock cycles on a current mote platform, a tradeoff between lots of short tasks and long running tasks that execute the same computation has to be found. Since data can not be kept on the stack between the execution of tasks, all state information has to be kept in the private memory of the tasks component[4, p. 74].

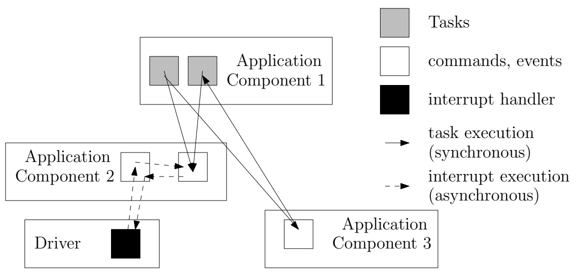

Figure 3: The TinyOS execution model. Component boundaries are crossed between all of the components[4]

3.3 Resource Use

TinyOS was built with the goal of minimal resource consumption since wireless sensor nodes are generally very constrained in regards to processing speed, program memory, RAM and power consumption.

3.3.1 Processing Power

To preserve processing power in TinyOS boundary crossings between different components are optimized. Since all function locations are known at compile time and there is no address-space crossing basic boundary crossing has at most the overhead of a single procedure call. With whole-program analysis many boundary crossings can be entirely removed. In some cases the compiler can even inline a whole component into it's caller[3].

To keep the overhead of task-switching minimal the scheduler in TinyOS is very simple. For example tasks have no return value and take no parameters so the scheduler does not need to take care of them[4, p. 72].

3.3.2 Program Memory

Since it is known at compile time, which components and which parts of those components of the application and the operating system are used, the compiled image of the application includes only the actually used procedures and as good as no dead code. Also the operating system itself has a very low memory footprint: The core TinyOS operating system uses less than 400 bytes of program memory[3].

3.3.3 RAM

Keeping the RAM usage of wireless sensor nodes low is very important, since the used microcontrollers are very restricted in RAM size. For example the Atmel Atmega128L micocontrollers used in the MICA2 sensor node[15] only offers 4 Kbytes of RAM[16] which have to be shared between the operating system and all running user programs. There are no memory management units (MMU) or other memory protection measures available on these microcontrollers so the risk of stack overflows (when the stack exceeds it's maximum size) should be avoided. The execution model with run to completion tasks is very efficient in terms of RAM usage, since all tasks use the same stack as opposed to a classic multi-threading approached where each thread needs a designated stack for itself. To further decrease stack size deep call hierarchies inside tasks should be avoided. TinyOS programmers are especially discouraged of using recursion and other similar techniques[4, p. 36-38].

3.3.4 Power Consumption

Since many wireless sensor nodes run on battery power, energy consumption should be kept as low as possible. To achieve a low power consumption the microcontroller should be kept in a low power sleep state for as long as possible. For example the Atmel Atmega128L needs a supply current in the magnitude of a few milliamperes when active. In the lowest sleep state a current of only a few microamperes[16] is sufficient, which means the difference between power consumption in active and sleep states can be a factor of 1000 or more. TinyOS copes with this by using the split-phase and event-driven execution model. As long as there are no tasks in the task queue the scheduler puts the CPU in sleep mode. So in combination with the split-phase operation the CPU does not waste energy while waiting for other hardware components[1].

But the CPU is not the only power critical component in a wireless sensor mode, periphery hardware can use a lot of power, too. For example the radio is in most cases the most expensive part of the node in respect to energy consumption[4, p. 7]. To save energy in those components TinyOS has a programming convention that allows subsystems to be put in a low power idle state. Components have an interface, which exposes commands to tell the component to try to minimize it's power consumption, for example by powering down hardware, and to wake up again from those power saving states[1].

3.4 Hardware Platforms

The current release of TinyOS supports quite a few sensor node hardware platforms. This includes different motes by Crossbow Technologies like the MICA family of motes or the iMote2 developed at Intel Research and many more[13]. These hardware platforms run on a range of different microcontroller including the ATMEL AVR family of 8-bit microcontrollers, the Texas Instruments MSP430 family of 16-bit microcontrollers, different generations of ARM cores, for example the Intel XScale PXA family, and more[13].

It is of course possible to use TinyOS for a custom or not yet supported hardware platform. The platforms in TinyOS are designed to be very modular. For each of the supported microcontrollers and other hardware, for example radio chips, components exist in the TinyOS installation. To port TinyOS to a platform it basically suffices to specify which components to use, how they are connected to each other (for example which pins of the microcontroller are connected to the radio chip) and to configure each of the components correctly[14].

If a hardware device of a platform is not supported by TinyOS a driver has to be programmed or a existing driver has to be ported to TinyOS. This is not trivial and can get very complex depending on the hardware[14].

To make software for TinyOS as platform independent as possible but at the same time offer the possibility to push the hardware to its limits with platform specific code, the hardware abstraction architecture (HAA) was introduced. The HAA offers three levels of hardware abstraction for drivers[4, p. 206-207]:

- The hardware interface layer (HIL): This is the most platform independent level of device drivers. It offers only the functionality that is common to all devices that use the common interfaces.
- The hardware adaption layer (HAL): The HAL is a tradeoff between platform independence and the use of platform specific code. It should offer platform independent interfaces when possible and platform specific interfaces in all other cases
- The hardware presentation layer (HPL): The platform specific level of the HAA. It sits directly on top of the hardware and offers all of it's functionality in a NesC friendly fashion.

3.5 Toolchain

The core of the TinyOS toolchain is the NesC compiler. Current implementations of the NesC compiler take all NesC files, including the TinyOS operating system, that belong to a program and generate a single C file. This C file can then be compiled by the native C compiler of choice for the target platform. The resulting binary can then be deployed on the motes in a appropriate way. Many optimizations are already done by the NesC compiler, for example the exclusion of dead code. Furthermore the output C file of the NesC compiler is constructed in a way, that makes it easy for the C compiler to further optimize it. Since it is just one single file the C compiler can freely optimize across call boundaries.

Apart from the actual build toolchain there is also a TinyOS simulator called TOSSIM[5]. It can simulate whole TinyOS programs and the underlying motes without the need of actually deploying it on hardware. With TOSSIM it is possible to simulate sensor node networks of thousands of nodes.

4. CONTIKI

Contiki is a open source operating systems for sensor 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. 4.1 and sec. 4.2 respectively. It's current version is 2.5 released on September 12, 2011.

4.1 Programming Model

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].

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. 4). The core usually is deployed as one single binary while the loaded programs each can be distributed independently[6].

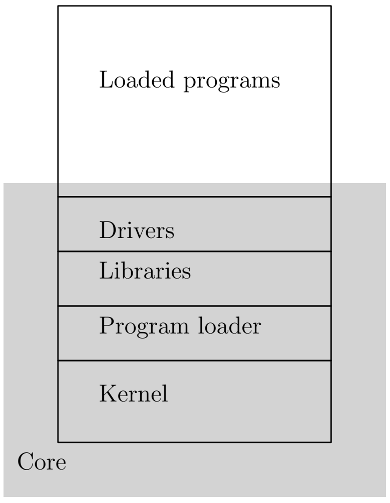

Figure 4: The partitioning of the Contiki core and the loaded programs

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.

4.2 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. Like the tasks in

```
// 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();
}
```

Listing 2: The example from Listing 1 implemented as a protothread for Contiki. With modifications from [17]

TinyOS these handlers all can operate on the same stack and do not need a private stack of their own[6].

There are two kinds of events in Contiki: Asynchronous and synchronous events. Asynchronous events work similar to the posting of tasks in TinyOS. 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.

While event handler can not preempt each other, interrupts can of course preempt the current running process. To prevent race-conditions from happening, events can not 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 can not 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 it's own designated stack[6].

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.

4.3 Resource Use

Since the scheduler and the kernel in general are more complex in Contiki than in TinyOS 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 than that of TinyOS programs. When using the preemptive multi-threading library the RAM usage will be higher than using only the event-driven kernel for scheduling[6].

4.4 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.

4.5 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. According to Dunkels et al. in [6] the port for the Atmel AVR was done by them in a view hours and the Zilog Z80 port was made by a third party in one single day.

4.6 Toolchain

Contiki is written in plain C so a native C compiler for the target platform can be used.

5. DISCUSSION

As presented in sec. 2 operating systems for wireless sensor nodes have to fulfill a few requirements. After the look at both TinyOS and Contiki we now compare both operating systems by means of these requirements:

- Limited resources: Both operating systems can be run on microcontrollers with very limited resources. But due to the higher complexity of the Contiki kernel TinyOS can generally get by with lower resource requirements.
- Concurrency: TinyOS offers only the event-driven kernel as a way of fulfilling the concurrency requirements. While Contiki also uses an event-driven kernel it also has different libraries that offer different levels of multithreading on top of that. But there are efforts to offer libraries similar to those of Contiki, for example by Klues et al. [8] or MacCartney et al. [9].
- Flexibility: Both operating systems are flexible to handle different types of applications. When it comes to updating an application that is already deployed Contiki can dynamically replace only the changed programs of the application, while an application using TinyOS has to be replaced completely, including the operating system. But there are solutions for making dynamic loading op application code possible for TinyOS, for example by introducing a virtual machine[10, 11].
- Low Power: TinyOS has out-of-the-box better energy conservation mechanisms but for Contiki similar power saving mechanisms can be implemented.

6. CONCLUSION

Both operating systems can generally fulfill all of the discussed requirements. In details there are differences, so while TinyOS is better suited when resources are really scarce and every little bit of saved memory or computing power can help, Contiki might be the better choice when flexibility is most important, for example when the node software has to be updated often for a large amount of nodes.

This paper is not an in depth discussion of both operating systems. When choosing the operating system for a specific application many things have to be considered and not all could be presented here. These two operating systems are also not the only operating systems for wireless sensor nodes so others may be considered.

7. 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, D. Gay *TinyOS Programming* Camebridge University Press, Camebridge, 2009

[5] 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

[6] A. Dunkels, B. Grönvall, 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] K. Klues, C.J.M. Liang, J. Paek, R. Musaloiu-E, P. Levis, A. Terzis, R. Govindan *TOSThreads: thread-safe and non-invasive preemption in TinyOS* In Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, p. 127-140, ACM, 2009

[9] W. P. McCartney, N. Sridhar *Stackless Preemptive Multi-Threading for TinyOS* In Proceedings of the 2011 International Conference on Distributed Computing in Sensor Systems (DCOSS), IEEE, 2011

[10] P. Levis, D. Culler *Maté: a tiny virtual machine for sensor networks* In Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, p. 85-95, ACM, 2002

[11] 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

[12] *TinyOS Open Technology Alliance*, `http://www.cs.berkeley.edu/~culler/tinyos/alliance/overview.pdf`

[13] *TinyOS Wiki - Platform Hardware*, `http://docs.tinyos.net/tinywiki/index.php?title=Platform_Hardware&oldid=5648`

[14] *TinyOS Wiki - Platforms*, `http://docs.tinyos.net/tinywiki/index.php?title=Platforms&oldid=4712`

[15] Crossbow Technology *MICA2 Datasheet* `http://bullseye.xbow.com:81/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf`

[16] Atmel Corporation *ATmega128/L Datasheet* `http://www.atmel.com/Images/doc2467.pdf`

[17] *Zoleria Dokumentation Wiki*, `http://zolertia.sourceforge.net/wiki/index.php?title=Mainpage:Contiki_Lesson_1&oldid=1138`