The Design of a Lightweight Portable Operating System for Tiny Networked Sensor Devices

Adam Dunkels, Björn Grönvall, Thiemo Voigt, Juan Alonso
Swedish Institute of Computer Science
{adam,bg,thiemo,alonso}@sics.se

March 10, 2004

SICS Technical Report T2004:05
ISSN 1100-3154
ISRN:SICS-T–2004/05-SE
Keywords: operating systems, sensor networks, embedded systems

Abstract
Wireless sensor networks are composed of large numbers of tiny networked devices that communicate untethered. In this paper we present the design of Contiki, a lightweight and portable operating system for such tiny devices. In this work, we try to find the right operating system abstractions that enable dynamic and efficient operation of a system with severe limitations. Contiki is built around a lightweight event scheduler and provides suitable abstractions for dynamic loading of programs, device drivers, and run-time linking of libraries. The system is highly portable and the kernel can be ported without changing a single line of code (except device drivers). We show how higher level abstractions such as multi-threading can be implemented as libraries on top of the lightweight event kernel.

1 Introduction

Wireless sensor networks are composed of large numbers of tiny sensor devices with wireless communication capabilities. The sensor devices autonomously form networks through which sensor data is transported. The sensor devices are often severely resource constrained. An on-board battery or solar panel can only supply limited amounts of power. The small physical size and low per-device cost limit the complexity of the system. Typical sensor devices [1, 2, 5] are equipped with 8-bit microcontrollers, code memory on the order of 100 kilobytes, and less than 20 kilobytes of RAM. Moore’s law predicts that these devices can be made significantly smaller and less expensive in the future. While this means that sensor networks can be deployed to greater extents, it does not necessarily imply that the resources will be less constrained.

To the designer of an operating system for sensor nodes, the challenge lies in finding the right lightweight mechanisms and abstractions to provide a rich enough execution environment while staying within the limitations of the constrained devices. We have developed Contiki, an operating system based on a very lightweight event-driven kernel. On top of the kernel higher level mechanisms such as preemptive multithreading can be built. Contiki draws from previous work on operating systems for
tiny sensor nodes [14]. Contiki adds a number of features currently not found in other systems: support for run-time loading and linking of libraries, programs, and device drivers as well as support for preemptive multi-threading.

1.1 Downloading code at run-time

The motivating factors behind Contiki are the practical problems one faces when deploying wireless sensor networks. First, having the ability to dynamically load and unload programs from a large number of sensor devices greatly reduces development time. Second, being able to add or remove functionality in an already deployed network is of importance. Third, software bugs in a running system may need to be patched [9]. In general, it is not realistic to physically collect and reprogram these devices, but in-situ procedures are required.

1.2 Portability

As the number of different sensor device platforms increases (e.g. [1, 2, 5]), highly portable systems are desirable. The currently available sensor platforms carry completely different sets of sensors and communication devices. Due to the application specific nature of sensor networks, we do not expect that this will change in the future. The single unifying characteristic of today’s platforms is the CPU architecture which uses a memory model without segmentation or memory protection mechanisms. Program code is stored in reprogrammable ROM and data in RAM. We have designed Contiki so that the only abstraction it provides is event based CPU multiplexing and the necessary memory management to support loadable programs. As a consequence of the application specific nature of sensor networks, we believe that other abstractions are better implemented as libraries or application programs.

1.3 Partially event driven system

While event driven system designs have been found to work well for many kinds of sensor network applications [17] they are not without problems. The state driven programming model can be hard to manage for programmers [16]. Also, not all programs are easily expressed as state machines. One example is the lengthy computation required for cryptographic operations. Such operations typically take several seconds to complete on CPU constrained platforms [20]. Purely event based systems would not be able to respond to external events during lengthy computations. Basing the operating system on preemptive multi-threading would help remedy these problems. However, it would add additional memory overhead for per-thread stacks and would also require programs to be reentrant and to use locking mechanisms. Instead, Contiki uses a compromise: the system is based on an event-driven kernel. Preemptive multi-threading is implemented as a library that is optionally linked with programs that explicitly require it.

Because of the non-preemptive nature of the kernel, Contiki cannot provide real-time guarantees. If such guarantees are needed, they can be implemented using interrupts provided by an underlying real-time executive or hardware timers. By design, Contiki never disables interrupts even for short periods of time.

The main contribution of this paper is the design of our portable lightweight event driven kernel. We also show how it can be used to efficiently implement mechanisms
for dynamic loading of programs and libraries. Furthermore, we show that this architecture can be extended with support for higher level primitives such as preemptive multi-threading.

The rest of this paper is structured as follows. Section 2 reviews related work and Section 3 presents an overview of the Contiki system. The design of the Contiki kernel is described in Section 4 and the system libraries in Section 5. Finally, the paper is summarized in Section 6.

2 Related work

TinyOS [14] is probably the earliest operating system that directly targets the specific applications and limitations of sensor devices. TinyOS is also built around a lightweight event scheduler where all program execution is performed in tasks that run to completion. TinyOS uses a special description language for composing a system of smaller components [12] which are statically linked with the kernel to a complete image of the system. After linking, modifying the system is not possible [16]. In contrast, Contiki provides a dynamic structure which allows programs and drivers to be replaced during run-time and without relinking.

In order to provide run-time reprogramming for TinyOS, Levis and Culler have developed Maté [16], a virtual machine for TinyOS devices. Code for the virtual machine can be downloaded into the system at run-time. The virtual machine is specifically designed for the needs of typical sensor network applications. Similarly, the MagnetOS [7] system uses a virtual Java machine to distribute applications across the sensor network. The advantages of using a virtual machine instead of native machine code is that the virtual machine code can be made smaller, thus reducing the energy consumption of transporting the code over the network. One of the drawbacks is the increased energy spent in interpreting the code—for long running programs the energy saved during the transport of the binary code is instead spent in the overhead of executing the code. Contiki programs use native code and can therefore be used for all types of programs, including low level device drivers without loss of execution efficiency.

SensorWare [8] provides an abstract scripting language for programming sensors, but their target platforms are not as resource constrained as ours. Reijers and Langendoen [19] use a patch language to modify parts of the binary image of a running system. This works well for networks where all nodes run the exact same binary code but soon gets complicated if sensors run slightly different programs or different versions of the same software.

The Mantis system [3] uses a traditional preemptive multi-threaded model of operation. Mantis enables reprogramming of both the entire operating system and parts of the program memory by downloading a program image onto EEPROM, from where it can be burned into flash ROM. Due to the multi-threaded semantics, every Mantis program must have stack space allocated from the system heap, and locking mechanisms must be used to achieve mutual exclusion of shared variables. In contrast, Contiki uses an event based scheduler without preemption, thus avoiding allocation of multiple stacks and locking mechanisms. Preemptive multi-threading is provided by a library that can be linked with programs that explicitly require it.

The preemptive multi-threading in Contiki is similar to fibers [4], and the lightweight fibers approach by Welsh and Mainland [21]. Unlike the lightweight fibers, Contiki does not limit the number of concurrent threads to two. Furthermore, unlike fibers, Contiki threads optionally support preemption.
As Exokernel [11] and Nemesis [15] Contiki tries to reduce the number of abstractions that the kernel provides to a minimum [10]. Abstractions are instead provided by libraries that have nearly full access to the underlying hardware. While Exokernel strived for performance and Nemesis aimed at quality of service, the purpose of the Contiki design is to reduce size, complexity, and to preserve flexibility.

3 System overview

Figure 1: Interactions between kernel, applications, libraries and hardware.

Contiki consists of an event driven kernel, a program loader, and a set of libraries. The kernel provides the basic CPU multiplexing and the libraries implement higher level functionality such as preemptive multi-threading and communication. The basic idea is to keep the kernel completely platform independent in order to make it portable. The kernel contains no platform specific code, such code is contained in libraries. The platform specific libraries contain device drivers specifically implemented for this platform. There are also platform independent libraries for communication and routing.

All events are handled in the context of a process. A process consists of a block of code, a part of the available RAM, and an event handler function. The process state is held in the process’ own memory. On our target platform [5], the process state consists of 23 bytes. All processes share the same address space and do not run in different protection domains. Interprocess communication is performed using events.

A Contiki system is partitioned into two parts: the core and the loadable programs as shown in Figure 2. The partitioning is made at compile time and is specific to the deployment in which Contiki is used. Typically, the core consists of the Contiki kernel, the program loader, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image that is stored in the devices prior to deployment. It is not possible to modify the core after deployment.

Programs are loaded into the system by the program loader. The program loader may obtain the program binaries either by using the communication stack or by using directly attached storage such as EEPROM.

1 It should be noted that it still is possible to use a special boot loader with the abilities to overwrite or patch the core.
Figure 2: Partitioning into core and loadable programs.

4 Kernel architecture

The Contiki kernel is based on a lightweight event scheduler. All program execution is triggered by events that are dispatched by the kernel. The kernel does not preempt an event handler once it has been scheduled, event handlers must therefore run to completion. However, event handlers may use external mechanisms to achieve preemption such as the preemptive multi-threading library described in Section 5.2.

There are three kinds of events in the Contiki kernel: asynchronous, synchronous, and polling events. Asynchronous events are a form of deferred procedure call. A process posts an event to the kernel which will enqueue the event and later dispatch it to the target process. Synchronous events are similar but will immediately cause the target process to be scheduled. Control will return to the posting process only after the target has finished handling the event. This is similar to the door abstraction used in the Spring operating system [13].

Polling events are high priority and are scheduled by the kernel between dispatch of each asynchronous event. Polling events are used to check for status updates of hardware devices. Polling events are executed in priority order.

The Contiki kernel uses a single shared stack for all process execution. The use of asynchronous events reduce stack space requirements as the stack is rewound between invocation of event handlers.

4.1 Two level scheduling hierarchy

All event scheduling in Contiki is done at a single level and Contiki supports the use of interrupts that preempt Contiki events. Normally, interrupts are implemented directly using hardware interrupts but may also be implemented using an underlying real-time executive, as done in e.g. RTLinux [6].

In order to support an underlying system with real-time guarantees, Contiki never disables interrupts. Therefore, Contiki does not allow interrupts to post events since that would require the kernel to disable interrupts. Instead, the kernel provides a polling flag that can be set by an interrupt handler in order to request service. When this flag is set, the kernel will schedule polling as soon as possible.
4.2 Loadable programs

Loadable programs are implemented using a run-time relocation function and a binary format that contains relocation information. When a program is loaded into the system, the loader first tries to allocate sufficient space based on information in the binary. If memory allocation fails, loading is aborted and the caller is informed of the failure.

After the program is loaded into memory, the loader calls the program’s initialization function. The initialization function registers the new process with the kernel and is thereafter ready to handle events.

4.3 Dynamic linking

In order to be able to replace libraries in deployed systems, calling Contiki libraries is somewhat complex.

Dynamic linking in Contiki is based on synchronous events. Shared libraries are loaded as ordinary programs. Library functions are indirectly called by passing synchronous events. The event contains the function to call, its arguments, and space for the return value. When the library has completed the call control returns to the calling process.

A rendezvous protocol is used to find the library that implements a particular function. The protocol consists of two messages, a request and a reply. The request is broadcast to all libraries, only the library implementing the function sends a reply containing its process ID. The process ID is necessary when posting events corresponding to library calls.

Since the process ID of a library is used as a library identifier, it is crucial that a replaced library retains the process ID of the old library. For this purpose, a special kernel function that replaces a running process with a new process with the same process ID, is used.

4.4 Context switching

Context switches in Contiki are performed when asynchronous events are dispatched, poll events are scheduled, or when synchronous events are passed between processes. Because synchronous events form the basis of the dynamic linking mechanism in Contiki, context switching time is crucial to the performance of the library calls. Since processes in Contiki are implemented as event handlers and do not run in separate protection domains, the context switching overhead is small. On our target platform, context switching is performed in 17 machine code instructions.

A context switch is nearly as lightweight as a procedure call. It involves looking up the new process ID in the list of processes, storing the ID in an internal variable and calling the event handler of the new process. For synchronous events, the process ID of the posting process is also saved on the stack before the new process is invoked.

4.5 Power save mode

In sensor networks, being able to power down the node when the network is inactive is an often required way to reduce energy consumption. Power conservation mechanisms depend on both the applications [17] and the network protocols [18]. Therefore, the Contiki kernel does not contain any explicit power save abstractions but leaves this to the application specific parts of the system. To help the application decide when to
power down the system, the event scheduler exposes its event queue. This information can be used, e.g., to power down the system when there are no scheduled events.

5 System libraries

The Contiki kernel only provides the most basic CPU multiplexing and event handling features. The rest of the system is implemented as system libraries that are optionally linked with programs. Programs can be linked with libraries in three different ways. First, programs can be statically linked with libraries that are part of the core. Second, programs can be statically linked with optional libraries that are part of the loadable program. Third, programs can call libraries through dynamic linking. Dynamically linked libraries have the advantage of being replaceable at run-time.

In the following sections we first present an example of a library that usually is dynamically linked and then an example of a library that is statically linked.

5.1 Communication support

Communication is a fundamental service in sensor networks. In Contiki, the communication stack is implemented as a library to enable dynamic replacement, even at run-time. Additionally, this provides for multiple communication stacks to be loaded simultaneously. In experimental research, this can be used to evaluate and compare different communication protocols. Furthermore, it is possible to split the communication stack into different libraries as shown in Figure 3. This enables run-time replacement of e.g. a routing algorithm, a transport protocol, or a device driver.

The communication stack and device drivers are implemented as libraries that communicate with each other and application programs using synchronous events. Because synchronous event handlers always run to completion a single buffer can be used for all communication processing and no data copying needs to be performed. A device driver reads an incoming packet into the communication buffer and then calls the upper layer library module using a synchronous event. The communication stack processes the headers of the packet and posts a synchronous event to the application program for which the packet was destined. The application program acts on the packet contents, and optionally puts a reply packet in the same buffer before it returns control to the
The communication stack prepends its headers to the outgoing packet and returns to the device driver so that the packet can be transmitted.

5.2 Preemptive multi-threading

In Contiki, preemptive multi-threading is implemented as a library optionally linked with applications that explicitly require a multi-threaded model of operation. The library is divided into two parts: a platform independent part that interfaces to the event based kernel, and a platform specific part implementing the stack switching and preemption primitives. Usually, the preemption is implemented using a timer interrupt that saves the processor registers onto the stack and switches back to the kernel stack. In practice, very little code needs to be rewritten when porting the platform specific part of the library. For reference, our platform has 31 lines of assembly code to implement this.

Unlike normal Contiki processes each thread requires a separate stack. The library provides the necessary stack management functions. Threads execute on their own stack until they either explicitly yield or are preempted. They communicate with the underlying Contiki system using events. An operation that invokes the kernel must first switch to the system stack and turn off preemption in order to avoid race conditions. For this reason, the multi-threading library provides its own functions for posting events to the kernel. Furthermore, blocking for specific events is also supported.

6 Summary

We have presented the portable lightweight Contiki operating system. One key contribution of Contiki is its support for dynamic loading of programs and libraries. The event-driven architecture of Contiki can be extended with libraries that provide higher level abstractions such as preemptive multi-threading and communication.

References

   URL: http://www.xbow.com/

   URL: http://eyes.eu.org/sensnet.htm


   URL: http://www.scatterweb.com/


