Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors

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

Abstract

Wireless sensor networks are composed of large numbers of tiny networked devices that communicate untethered. For large scale networks it is important to be able to dynamically download code into the network. In this paper we present Contiki, a lightweight operating system with support for dynamic loading and replacement of individual programs and services. Contiki is built around an event-driven kernel but provides optional preemptive multi-threading that can be applied to individual processes. We show that dynamic loading and unloading is feasible in a resource constrained environment, while keeping the base system lightweight and compact.

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. Moreover, 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.

For the designer of an operating system for sensor nodes, the challenge lies in finding lightweight mechanisms and abstractions that provide a rich enough execution environment while staying within the limitations of the constrained devices. We have developed Contiki, an operating system developed for such constrained environments. Contiki provides dynamic loading and unloading of individual programs and services. The kernel is event-driven, but the system supports preemptive multi-threading that can be applied on a per-process basis. Preemptive multi-threading is implemented as a library that is linked only with programs that explicitly require multi-threading.

Contiki is implemented in the C language and has been ported to a number of microcontroller architectures, including the Texas Instruments MSP430 and the Atmel AVR. We are currently running it on the ESB platform [5]. The ESB uses the MSP430 microcontroller with 2 kilobytes of RAM and 60 kilobytes of ROM running at 1 MHz. The microcontroller has the ability to selectively reprogram parts of the on-chip flash memory.

The contributions of this paper are twofold. Our first contribution is that we show the feasibility of loadable programs and services even in a constrained sensor device. The possibility to dynamically load individual programs leads to a very flexible architecture, which still is compact enough for resource constrained sensor nodes. Our second contribution is more general in that we show that preemptive multi-threading does not have to be implemented at the lowest level of the kernel but that it can be built as an application library on top of an event-driven kernel. This allows for thread-based programs running on top of an event-based kernel, without the overhead of reentrancy or multiple stacks in all parts of the system.

1.1. Downloading code at run-time

Wireless sensor networks are envisioned to be large scale, with hundreds or even thousands of nodes per network. When developing software for such a large sensor network, being able to dynamically download program code into the network is of great importance. Furthermore, bugs may have to be patched in an operational network [9]. In general, it is not feasible to physically collect and reprogram all sensor devices and in-situ mechanisms are required. A number of methods for distributing code in wireless sensor networks have been developed [21, 8, 17]. For such methods it is important to reduce the number of bytes sent over

the network, as communication requires a large parts of the available node energy.

Most operating systems for embedded systems require that a complete binary image of the entire system is built and downloaded into each device. The binary includes the operating system, system libraries, and the actual applications running on top of the system. In contrast, Contiki has the ability to load and unload individual applications or services at run-time. In most cases, an individual application is much smaller than the entire system binary and therefore requires less energy when transmitted through a network. Additionally, the transfer time of an application binary is less than that of an entire system image.

1.2. Portability

As the number of different sensor device platforms increases (e.g. [1, 2, 5]), it is desirable to have a common software infrastructure that is portable across hardware platforms. 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 provided by the base system is CPU multiplexing and support for loadable programs and services. As a consequence of the application specific nature of sensor networks, we believe that other abstractions are better implemented as libraries or services and provide mechanisms for dynamic service management.

1.3. Event-driven systems

In severely memory constrained environments, a multi-threaded model of operation often consumes large parts of the memory resources. Each thread must have its own stack and because it in general is hard to know in advance how much stack space a thread needs, the stack typically has to be over provisioned. Furthermore, the memory for each stack must be allocated when the thread is created. The memory contained in a stack can not be shared between many concurrent threads, but can only be used by the thread to which is was allocated. Moreover, a threaded concurrency model requires locking mechanisms to prevent concurrent threads from modifying shared resources.

To provide concurrency without the need for per-thread stacks or locking mechanisms, event-driven systems have been proposed [15]. In event-driven systems, processes are implemented as event handlers that run to completion. Because an event handler cannot block, all processes can use the same stack, effectively sharing the scarce memory resources between all processes. Also, locking mechanisms are generally not needed because two event handlers never run concurrently with respect to each other.

While event-driven system designs have been found to work well for many kinds of sensor network applications [18] they are not without problems. The state driven programming model can be hard to manage for programmers [17]. Also, not all programs are easily expressed as state machines. One example is the lengthy computation required for cryptographic operations. Typically, such operations take several seconds to complete on CPU constrained platforms [22]. In a purely event-driven operating system a lengthy computation completely monopolizes the CPU, making the system unable to respond to external events. If the operating system instead was based on preemptive multi-threading this would not be a problem as a lengthy computation could be preempted.

To combine the benefits of both event-driven systems and preemptible threads, Contiki uses a hybrid model: the system is based on an event-driven kernel where preemptive multi-threading is implemented as an application library that is *optionally* linked with programs that *explicitly* require it.

The rest of this paper is structured as follows. Section 2 reviews related work and Section 3 presents an overview of the Contiki system. We describe the design of the Contiki kernel in Section 4. The Contiki service concept is presented in Section 5. In the following section, we describe how Contiki handles libraries and communication support is discussed in Section 7. We present the implementation of preemptive multi-threading in Section 8 and our experiences with using the system is discussed in Section 9. Finally, the paper is concluded in Section 10.

2. Related work

TinyOS [15] 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 [17]. 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 [17], a virtual machine for TinyOS devices. Code for the virtual machine can be downloaded into the system at run-time. The virtual ma-

chine 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. Similarly, the EmStar environment [13] is designed for less resource constrained systems. Reijers and Langendoen [21] 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 [23]. Unlike the lightweight fibers, Contiki does not limit the number of concurrent threads to two. Furthermore, unlike fibers, threads in Contiki support preemption.

As Exokernel [11] and Nemesis [16], 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 and complexity, as well as to preserve flexibility. Unlike Exokernel, Contiki do not support any protection mechanisms since the hardware for which Contiki is designed do not support memory protection.

3. System overview

A running Contiki system consists of the kernel, libraries, the program loader, and a set of processes. A process may be either an application program or a *service*. A service implements functionality used by more than one application process. All processes, both application programs and services, can be dynamically replaced at run-time. Communication between processes always goes through the kernel. The kernel does not provide a hardware abstraction layer, but lets device drivers and applications communicate directly with the hardware.

A process is defined by an event handler function and an optional poll handler function. The process state is held in the process' private memory and the kernel only keeps a pointer to the process state. On the ESB 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 done by posting events.

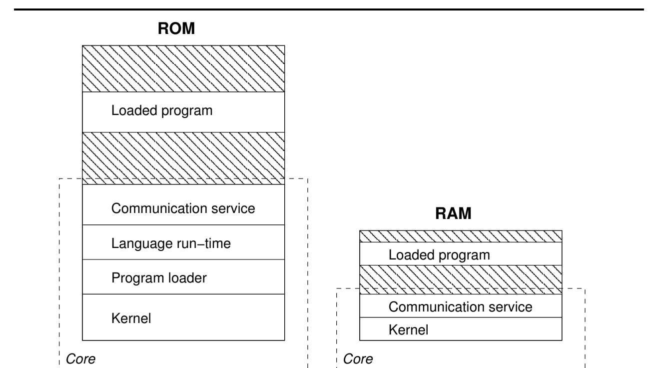

Figure 1. Partitioning into core and loaded programs.

A Contiki system is partitioned into two parts: the *core* and the *loaded programs* as shown in Figure 1. 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, the most commonly used parts of the language run-time and support libraries, 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. The core is generally not modified after deployment, even though it should be noted that it is possible to use a special boot loader to overwrite or patch the core.

Programs are loaded into the system by the program loader. The program loader may obtain the program bina-

ries either by using the communication stack or by using directly attached storage such as EEPROM. Typically, programs to be loaded into the system are first stored in EEPROM before they are programmed into the code memory.

4. Kernel architecture

The Contiki kernel consists of a lightweight event scheduler that dispatches events to running processes and periodically calls processes' polling handlers. All program execution is triggered either by events dispatched by the kernel or through the polling mechanism. The kernel does not preempt an event handler once it has been scheduled. Therefore, event handlers must run to completion. As shown in Section 8, however, event handlers may use internal mechanisms to achieve preemption.

The kernel supports two kind of events: *asynchronous* and *synchronous* events. Asynchronous events are a form of deferred procedure call: asynchronous events are enqueued by the kernel and are dispatched to the target process some time later. Synchronous events are similar to asynchronous but immediately causes the target process to be scheduled. Control returns to the posting process only after the target has finished processing the event. This can be seen as an inter-process procedure call and is similar to the door abstraction used in the Spring operating system [14].

In addition to the events, the kernel provides a *polling* mechanism. Polling can be seen as high priority events that are scheduled in-between each asynchronous event. Polling is used by processes that operate near the hardware to check for status updates of hardware devices. When a poll is scheduled all processes that implement a poll handler are called, in order of their priority.

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 each invocation of event handlers.

4.1. Two level scheduling hierarchy

All event scheduling in Contiki is done at a single level and events cannot preempt each other. Events can only be preempted by interrupts. Normally, interrupts are implemented using hardware interrupts but may also be implemented using an underlying real-time executive. The latter technique has previously been used to provide real-time guarantees for the Linux kernel [6].

In order to be able to support an underlying real-time executive, Contiki never disables interrupts. Because of this, Contiki does not allow events to be posted by interrupt handlers as that would lead to race-conditions in the event handler. Instead, the kernel provides a polling flag that it used to request a poll event. The flag provides interrupt handlers with a way to request immediate polling.

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 memory space based on information provided by the binary. If memory allocation fails, program loading is aborted.

After the program is loaded into memory, the loader calls the program's initialization function. The initialization function may start or replace one or more processes.

4.3. 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 [18] and the network protocols [20]. The Contiki kernel contains no explicit power save abstractions, but lets the the application specific parts of the system implement such mechanisms. To help the application decide when to power down the system, the event scheduler exposes the size of the event queue. This information can be used to power down the processor when there are no events scheduled. When the processor wakes up in response to an interrupt, the poll handlers are run to handle the external event.

5. Services

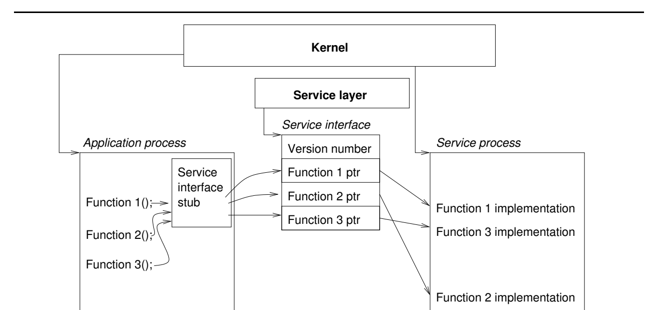

Figure 2. An application function calling a service.

In Contiki, a *service* is a process that implements functionality that can be used by other processes. A service can be seen as a form of a shared library. Services can be dynamically replaced at run-time and must therefore be

dynamically linked. Typical examples of services includes communication protocol stacks, sensor device drivers, and higher level functionality such as sensor data handling algorithms.

Services are managed by a *service layer* conceptually situated directly next to the kernel. The service layer keeps track of running services and provides a way to find installed services. A service is identified by a textual string that describes the service. The service layer uses ordinary string matching to querying installed services.

A service consists of a *service interface* and a process that implements the interface. The service interface consists of a version number and a function table with pointers to the functions that implement the interface.

Application programs using the service use a stub library to communicate with the service. The stub library is linked with the application and uses the service layer to find the service process. Once a service has been located, the service stub caches the process ID of the service process and uses this ID for all future requests.

Programs call services through the service interface stub and need not be aware of the fact that a particular function is implemented as a service. The first time the service is called, the service interface stub performs a service lookup in the service layer. If the specified service exists in the system, the lookup returns a pointer to the service interface. The version number in the service interface is checked with the version of the interface stub. In addition to the version number, the service interface contains pointers to the implementation of all service functions. The function implementations are contained in the service process. If the version number of the service stub match the number in the service interface, the interface stub calls the implementation of the requested function.

5.1. Service replacement

Like all processes, services may be dynamically loaded and replaced in a running Contiki system. Because the process ID of the service process is used as a service identifier, it is crucial that the process ID is retained if the service process is replaced. For this reason, the kernel provides special mechanism for replacing a process and retaining the process ID.

When a service is to be replaced, the kernel informs the running version of the service by posting a special event to the service process. In response to this event, the service must remove itself from the system.

Many services have an internal state that may need to be transfered to the new process. The kernel provides a way to pass a pointer to the new service process, and the service can produce a state description that is passed to the new process. The memory for holding the state must be allocated from a shared source, since the process memory is deallocated when the old process is removed.

The service state description is tagged with the version number of the service, so that an incompatible version of the same service will not try to load the service description.

6. 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 libraries that are part of the loadable program. Third, programs can call services implementing a specific library. Libraries that are implemented as services can be dynamically replaced at runtime.

Typically, run-time libraries such as often-used parts of the language run-time libraries are best placed in the Contiki core. Rarely used or application specific libraries, however, are more appropriately linked with loadable programs. Libraries that are part of the core are always present in the system and do not have to be included in loadable program binaries.

As an example, consider a program that uses the `memcpy()` and `atoi()` functions to copy memory and to convert strings to integers, respectively. The `memcpy()` function is a frequently used C library function, whereas `atoi()` is used less often. Therefore, in this particular example, `memcpy()` has been included in the system core but not `atoi()`. When the program is linked to produce a binary, the `memcpy()` function will be linked against its static address in the core. The object code for the part of the C library that implements the `atoi()` function must, however, be included in the program binary.

7. Communication support

Communication is a fundamental concept in sensor networks. In Contiki, communication is implemented as a service in order to enable run-time replacement. Implementing communication as a service also provides for multiple communication stacks to be loaded simultaneously. In experimental research, this can be used to evaluate and compare different communication protocols. Furthermore, the communication stack may be split into different services as shown in Figure 3. This enables run-time replacement of individual parts of the communication stack.

Communication services use the service mechanism to call each other and synchronous events to communicate

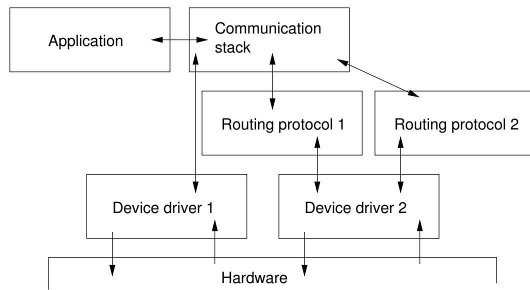

Figure 3. Loosely coupled communication stack.

with application programs. Because synchronous event handlers are required to be run to completion, it is possible to use a single buffer for all communication processing. With this approach, no data copying has to be performed. A device driver reads an incoming packet into the communication buffer and then calls the upper layer communication service using the service mechanisms. 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 in the buffer before it returns control to the communication stack. The communication stack prepends its headers to the outgoing packet and returns control to the device driver so that the packet can be transmitted.

8. Preemptive multi-threading

In Contiki, preemptive multi-threading is implemented as a library on top of the event-based kernel. The library is 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 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, the implementation for the MSP430 consists of 25 lines of C code.

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.

The API of the multi-threading library is shown in Figure 4. It consists of four functions that can be called from a running thread (`mt_yield()`, `mt_post()`, `mt_wait()`, and `mt_exit()`) and two functions that are called to setup and run a thread (`mt_start()` and `mt_exec()`). The `mt_exec()` function performs the actual scheduling of a thread and is called from an event handler.

`mt_yield();`
Yield from the running thread.

`mt_post(id, event, dataptr);`
Post an event from the running thread.

`mt_wait(event, dataptr);`
Wait for an event to be posted to the running thread.

`mt_exit();`
Exit the running thread.

`mt_start(thread, functionptr, dataptr);`
Start a thread with a specified function call.

`mt_exec(thread);`
Execute the specified thread until it yields or is preempted.

Figure 4. The multi-threading library API.

9. Discussion

We have used the Contiki operating system to implement a number of sensor network applications such as multi-hop routing, motion detection with distributed sensor data logging and replication, and presence detection and notification.

9.1. Over-the-air programming

We have implemented a simple protocol for over-the-air programming of entire networks of sensors. The protocol transmits a single program binary to selected concentrator nodes using point-to-point communication. The binary is stored in EEPROM and when the entire program has been received, it is broadcasted to neighboring nodes. Packet loss is signaled by neighbors using negative acknowledgments. Repairs are made by the concentrator node. We intend to implement better protocols, such as the Trickle algorithm [19], in the future.

During the development of one network application, a 40-node dynamic distributed alarm system, we used both over-the-air reprogramming and manual wired reprogramming of the sensor nodes. At first, the program loading mechanism was not fully functional and we could not use it during our development. The object code size of our application was approximately 6 kilobytes. Together with the Contiki core and the C library, the complete system image was nearly 30 kilobytes. Reprogramming of an individual sensor node took just over 30 seconds. With 40 nodes, reprogramming the entire network required at least 30 min-

utes of work and was therefore not feasible to do often. In contrast, over-the-air reprogramming of a single component of the application was done in about two minutes—a reduction in an order of magnitude—and could be done with the sensor nodes placed in the actual test environment.

9.2. Code size

An operating system for constrained devices must be compact in terms of both code size and RAM usage in order to leave room for applications running on top of the system. Table 1 shows the compiled code size and the RAM usage of the Contiki system compiled for two architectures: the Texas Instruments MSP430 and the Atmel AVR. The numbers report the size of both core components and an example application: a sensor data replicator service. The replicator service consists of the service interface stub for the service as well as the implementation of the service itself. The program loader is currently only implemented on the MSP430 platform.

The code size of Contiki is larger than that of TinyOS [15], but smaller than that of the Mantis system [3]. Contiki's event kernel is significantly larger than that of TinyOS because of the different services provided. While the TinyOS event kernel only provides a FIFO event queue scheduler, the Contiki kernel supports both FIFO events and poll handlers with priorities. Furthermore, the flexibility in Contiki requires more run-time code than for a system like TinyOS, where compile time optimization can be done to a larger extent.

Module	Code size (AVR)	Code size (MSP430)	RAM usage
Kernel	1044	810	10 + + $4e$ + $2p$
Service layer	128	110	0
Program loader	-	658	8
Multi-threading	678	582	8 + s
Timer library	90	60	0
Replicator stub	182	98	4
Replicator	1752	1558	200
Total	3874	3876	230 + $4e$ + + $2p$ + s

Table 1. Size of the compiled code, in bytes.

The RAM requirement depends on the maximum number of processes that the system is configured to have (p), the maximum size of the asynchronous event queue (e) and, in the case of multi-threaded operation, the size of the thread stacks (s).

9.3. Preemption

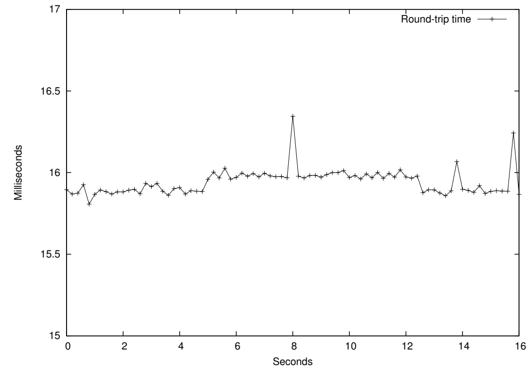

Figure 5. A slight increase in response time during a preemptible computation.

The purpose of preemption is to facilitate long running computations while being able to react on incoming events such as sensor input or incoming communication packets. Figure 5 shows how Contiki responds to incoming packets during an 8 second computation running in a preemptible thread. The curve is the measured round-trip time of 200 "ping" packets of 40 bytes each. The computation starts after approximately 5 seconds and runs until 13 seconds have passed. During the computation, the round-trip time increases slightly but the system is still able to produce replies to the ping packets.

The packets are sent over a 57600 kbit/s serial line with a spacing of 200 ms from a 1.4 GHz PC to an ESB node running Contiki. The packets are transmitted over a serial line rather than over the wireless link in order to avoid radio effects such as bit errors and MAC collisions. The computation consists of an arbitrarily chosen sequence of multiplications and additions that are repeated for about 8 seconds. The cause for the increase in round-trip time during the computation is the cost of preempting the computation and restoring the kernel context before the incoming packet can be handled. The jitter and the spikes of about 0.3 milliseconds seen in the curve can be contributed to activity in other poll handlers, mostly the radio packet driver.

9.4. Portability

We have ported Contiki to a number of architectures, including the Texas Instruments MSP430 and the Atmel AVR. Others have ported the system to the Hitachi SH3 and the Zilog Z80. The porting process consists of writing the boot up code, device drivers, the architecture specific parts of the

program loader, and the stack switching code of the multi-threading library. The kernel and the service layer does not require any changes.

Since the kernel and service layer does not require any changes, an operational port can be tested after the first I/O device driver has been written. The Atmel AVR port was made by ourselves in a couple of hours, with help of publicly available device drivers. The Zilog Z80 port was made by a third party, in a single day.

10. Conclusions

We have presented the Contiki operating system, designed for memory constrained systems. In order to reduce the size of the system, Contiki is based on an event-driven kernel. The state-machine driven programming of event-driven systems can be hard to use and has problems with handling long running computations. Contiki provides preemptive multi-threading as an application library that runs on top of the event-driven kernel. The library is *optionally* linked with applications that *explicitly* require a multi-threaded model of computation.

A running Contiki system is divided into two parts: a core and loaded programs. The core consists of the kernel, a set of base services, and parts of the language run-time and support libraries. The loaded programs can be loading and unloading individually, at run-time. Shared functionality is implemented as *services*, a form of shared libraries. Services can be updated or replaced individually, which leads to a very flexible structure.

We have shown that dynamic loading and unloading of programs and services is feasible in a resource constrained system, while keeping the base system lightweight and compact. Even though our kernel is event-based, preemptive multi-threading can be provided at the application layer on a per-process basis.

Because of its dynamic nature, Contiki can be used to multiplex the hardware of a sensor network across multiple applications or even multiple users. This does, however, require ways to control access to the reprogramming facilities. We plan to continue our work in the direction of operating system support for secure code updates.

References

[1] Berkeley mica motes. Web page. Visited 2004-06-22. http://www.xbow.com/

[2] Eyes prototype sensor node. Web page. Visited 2004-06-22. http://eyes.eu.org/sensnet.htm

[3] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: system support for MultimodAl NeTworks of In-Situ sensors. In *Proc. WSNA'03*, 2003.

[4] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proc. USENIX*, 2002.

[5] CST Group at FU Berlin. Scatterweb Embedded Sensor Board. Web page. Visited 2004-06-22. http://www.scatterweb.com/

[6] M. Barabanov. A Linux-based RealTime Operating System. Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[7] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(2), 2002.

[8] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. MOBISYS'03*, May 2003.

[9] D. Estrin (editor). *Embedded everywhere: A research agenda for networked systems of embedded computers*. National Academy Press, 2001.

[10] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proc. HotOS-V*, May 1995.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc SOSP '95*, December 1995.

[12] 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 *Proc. SIGPLAN'03*, 2003.

[13] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks. In *Proc. USENIX*, 2004.

[14] G. Hamilton and P. Kougiouris. The spring nucleus: A microkernel for objects. In *Proc. Usenix Summer Conf.*, 1993.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. ASPLOS-IX*, November 2000.

[16] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, 14(7):1280–1297, 1996.

[17] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. ASPLOS-X*, October 2002.

[18] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proc. NSDI*, 2004.

[19] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. NSDI*, 2004.

[20] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava. Energy aware wireless microsensor networks. *IEEE Signal Processing Magazine*, 19(2):40–50, 2002.

[21] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proc. WSNA'03*, 2003.

[22] F. Stajano. *Security for Ubiquitous Computing*. Wiley, 2002.

[23] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proc. NSDI*, 2004.