UPPSALA
UNIVERSITET

IT Licentiate theses
2011-004

Towards Adaptive Sensor Networks

NICLAS FINNE

UPPSALA UNIVERSITY
Department of Information Technology

Towards Adaptive Sensor Networks

Niclas Finne
nfi@sics.se

May 2011

Division of Computer Systems
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Swedish Institute of Computer Science | SICS

Dissertation for the degree of Licentiate of Philosophy in Computer Science

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

Wireless sensor networks consist of many small embedded devices that are equipped with sensors and a wireless communication unit. These devices, or sensor nodes, are typically low cost, resource constrained and battery-powered. Sensor network applications include environmental monitoring, industrial condition monitoring, building surveillance, and intelligent homes.

Sensor network applications today are often developed either using standard software components which enables simpler development but leads to far from optimal performance, or software customized for the specific application which complicates development, software updates, and software reuse.

We suggest that logic is separated from configuration and other information, for instance, network statistics and estimated power consumption. Software components publish their configuration and other information using a generalized programming abstraction. Configuration policies are separate modules that react on changes and reconfigure the sensor node as needed. These configuration policies are responsible for coordinating the configuration between the components and optimize the sensor network towards the application objectives.

One of our contributions is that we demonstrate the need for self-monitoring and self-configuration based on experiences from two deployed sensor networks. Our main contribution is that we devise a configuration architecture that solves the problem of cross-layer optimization for sensor network applications without tight coupling between components, thus enabling standard and easily replaceable components to be used. The configuration architecture makes it possible to reach the same level of performance as specialized cross-layer optimizations but without adding application-specific knowledge to the components.

Acknowledgments

First, I would like to thank my advisor Thiemo Voigt for his support and encouragement during this thesis. I would also like to thank my co-advisor Mats Björkman for his good advices and valuable feedback. I am also grateful to Per Gunningberg, co-advisor and former main advisor.

Many thanks to Adam Dunkels, Joakim Eriksson, Zhitao He, Joel Höglund, Daniel Gillblad, Luca Mottola, Nicolas Tsiftes, Niklas Wirström, Fredrik Österlind and my other colleagues at SICS. Furthermore, I would like to thank Sverker Janson for offering me the chance to be a member of NES/CSL at SICS.

I am grateful to the SAVE-IT industrial Ph.D. program at MdH in which I have been an industrial Ph.D. student during the thesis. I am also grateful to ABB Corporate Research in Västerås for providing me with the opportunity to participate in industrial research projects. Many thanks to Stefan Svensson, Mikael Gidlund, Jan-Erik Frej, Martin Strand, Tomas Lennvall, Lennart Balgård and Jonas Neander for interesting discussions.

Throughout the work on the thesis my research has been funded by a number of projects financed by SAVE-IT/KKS, SSF, VINNOVA, ITEA, Stiftelsen för Internetinfrastruktur, FMV, the SICS Center for Networked Systems and the Uppsala VINN Excellence Center for Wireless Sensor Networks (WISENET). My work has also been supported by the European Commission with contract FP7-2007-2-224053 (CONET) and 224282 (GINSENG). The final write-up of the thesis has been partially funded by SSF's ProInstitute Grant.

The Swedish Institute of Computer Science is sponsored by TeliaSonera, Ericsson, Saab SDS, FMV (Defence Materiel Administration), Green Cargo (Swedish freight railway operator), ABB and Bombardier Transportation.

Included Papers

This thesis is based on the following papers, which are referred in the text as Paper A, Paper B, and Paper C.

Paper A Adam Dunkels, Niclas Finne, Joakim Eriksson, Thiemo Voigt. Runtime dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (ACM SenSys 2006)*, Boulder, Colorado, USA, November 2006.

Paper B Niclas Finne, Joakim Eriksson, Adam Dunkels, Thiemo Voigt. Experiences from two sensor network deployments: self-monitoring and self-configuration keys to success. In *Proceedings of the 6th International Conference on Wired/Wireless Internet Communications (WWIC 2008)*, Tampere, Finland, May 2008.

Paper C Niclas Finne, Joakim Eriksson, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt. Improving Sensor Network Performance by Separating Protocol Configuration from Protocol Logic. In *Proceedings of the 7th European Conference on Wireless Sensor Networks (EWSN 2010)*, Coimbra, Portugal, February 2010.

Reprints were made with permission from the publishers.

Selected Papers Not Included in the Thesis

- Adam Dunkels, Luca Mottola, Nicolas Tsiftes, Fredrik Österlind, Joakim Eriksson, Niclas Finne (2011) The Announcement Layer: Beacon Coordination for the Sensornet Stack. In: 8th European Conference on Wireless Sensor Networks (EWSN 2011), 23-25 Feb 2011, Bonn, Germany.
- Fredrik Österlind, Niklas Wirström, Nicolas Tsiftes, Niclas Finne, Thiemo Voigt, Adam Dunkels (2010) StrawMAN: Making Sudden Traffic Surges Graceful in Low-Power Wireless Networks. In: ACM HotEMNETS 2010 Workshop on Hot Topics in Embedded Networked Sensors, June 2010, Killarney, Ireland.
- Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, Pedro José Marrón (2009) COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks. In: 2nd International Conference on Simulation Tools and Techniques (SIMUTools), 2-6 Mar 2009, Rome, Italy.
- Fredrik Österlind, Adam Dunkels, Thiemo Voigt, Nicolas Tsiftes, Joakim Eriksson, Niclas Finne (2009) Sensornet checkpointing: enabling repeatability in testbeds and realism in simulations. In: EWSN 2009: 6th European Conference on Wireless Sensor Networks, 11-13 Feb 2009, Cork, Ireland.
- Joakim Eriksson, Fredrik Österlind, Niclas Finne, Adam Dunkels, Thiemo Voigt, Nicolas Tsiftes (2009) Accurate, network-scale power profiling for sensor network simulators. In: EWSN 2009: 6th European Conference on Wireless Sensor Networks, 11-13 Feb 2009, Cork, Ireland.
- Thiemo Voigt, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Zhitao He, Joakim Eriksson, Adam Dunkels, Ulf Båmstedt, Jochen Schiller, Klas Hjort (2007) Sensor Networking in Aquatic Environments - Experiences and New Challenges. In: Second IEEE International Workshop

on Practical Issues in Building Sensor Network Applications, 15-18 Oct 2007, Dublin, Ireland.

- Fredrik Österlind, Erik Pramsten, Daniel Roberthson, Joakim Eriksson, Niclas Finne, Thiemo Voigt (2007) Integrating Building Automation Systems and Wireless Sensor Networks. In: 12th IEEE Conference on Emerging Technologies and Factory Automation, 25-28 September 2007, Patras, Greece.

- Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, Thiemo Voigt (2007) Cross-level sensor network simulation with Cooja. In: Real-Time in Sweden 2007, August 2007, Västerås, Sweden.

- Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, Thiemo Voigt (2007) Cross-level simulation in cooja. In: European Conference on Wireless Sensor Networks (EWSN), January 2007, Delft, The Netherlands.

- Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Österlind, Thiemo Voigt (2007) Mspsim - an extensible simulator for msp430-equipped sensor boards. In: European Conference on Wireless Sensor Networks (EWSN), January 2007, Delft, The Netherlands.

- Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, Thiemo Voigt (2006) Cross-level sensor network simulation with COOJA. In: First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006), November 2006, Tampa, Florida, USA.

Contents

Part I

Thesis

Chapter 1

Introduction

1.1 Wireless Sensor Networks

Wireless sensor networks consist of many small resource constrained computing devices equipped with a wireless communication unit, and one or more sensors or actuators. These devices, or sensor nodes, are typically battery-powered. Installing cables for communication can be extremely expensive or even infeasible in some environments as, for example, factories. Even in home environments it is often impractical to draw cables to each sensor node.

Since the nodes usually have limited energy supplies, the sensor network lifetime is directly dependent on the energy consumption and conserving energy is one of the most important issues in sensor networks.

Wireless sensor networks are used for various types of applications including environmental monitoring, industrial condition monitoring, building surveillance, and intelligent homes [35].

1.1.1 Hardware

A sensor node is typically equipped with an 8- or 16-bit CPU with 2-10 kB RAM, 30-128 kB programmable flash, and runs at 4-8 MHz (for instance, the sensor nodes Tmote Sky and Sentilla JCreate). This might change somewhat with the development of energy efficient 32-bit CPUs with higher performance such as the low-power versions of the ARM CPU family. There are already examples of this like, for instance, the Redbee-Econotag with 96 kB RAM runs at 24 MHz. Still the nodes remain very resource constrained compared to normal desktop computers or smart phones to keep down the cost and energy consumption. Figure 1.1 shows a Sentilla JCreate node and a Tmote Sky node, a sensor node commonly used in sensor network research.

For communication the nodes are usually equipped with a short range wireless radio chip. The CC2420 [37] is an well-used radio chip with data rates up to 250 kilobits per seconds and a range of a few hundred meters

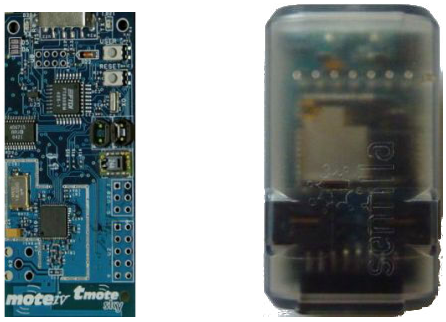


Figure 1.1: Two example sensor nodes. To the left a Tmote Sky node equipped with light, temperature and humidity sensors. To the right a Sentilla JCreate node equipped with accelerometer.

out-door. In-door the range can be much less depending on the environment where, for instance, concrete walls affect the radio signals. A short range radio is used to reduce the power consumption and the cost of the nodes.

Cost is normally an important factor when developing sensor network applications. To reduce the cost, inexpensive and low-quality components can be used for the sensor nodes. As a result each node might behave somewhat different from the other nodes. For example, the sensitivity of sensors or range of radio communication can differ between nodes and these differences need to be addressed by the software if they affect the application performance.

1.1.2 Software

With the development of more advanced energy efficient communication protocols and more capable hardware, developers are moving from programming applications directly on the hardware to programming on operating systems such as the Contiki OS [7], TinyOS [11], Mantis [2], and SOS [10]. The operating systems provide hardware abstractions, which enables the same software to be compiled for multiple platforms. This greatly helps to reuse software components between sensor network applications since software components can be developed with APIs provided by the operating systems instead of specific hardware.

Contiki is an operating system for memory-constrained embedded devices developed at the Swedish Institute of Computer Science [7]. This operating system has been ported to a number of platforms and includes implementations of various Media Access Control (MAC) and routing protocols. Also included is a software based energy measurement mechanism for online estimation of power consumption [8]. Contiki is accompanied with the network simulator COOJA/MSPSim, that is very useful for testing and

developing sensor node software [29]. The software developed during this thesis has been developed for Contiki.

The sensor nodes are usually equipped with a limited energy supply such as a battery. Therefore a lot of research effort has been focused on conserving energy. Power can be saved by turning off any hardware that currently is not needed. The CPU can be put in energy efficient sleep modes when there is no work to be done, and most peripherals such as sensors can be completely turned off. Turning hardware on and off in this manner is called duty cycling and is very important to increase the sensor network lifetime [41].

The sensor nodes can be deployed in large numbers and in locations that are not easily accessible, which makes it cumbersome to physically access the nodes once the network has been deployed. Even with a few nodes, it can be very time-consuming to retrieve and reprogram the nodes if the need arises. There are many reasons to update the nodes after deployment such as software bug fixes, adding new functionality, or changed application objectives. For this purpose, mechanisms are needed to distribute updates using radio communication and install the updates onto already deployed sensor nodes.

1.1.3 Low-Power Wireless Communication

The radio chip is one of the largest energy consumers in sensor networks. For example, the radio chip CC2420, which is used in the Tmote Sky and Sentilla JCreate motes, draws 19.7 mA when listening and 17.4 mA when transmitting [37]. These numbers show that the CC2420 actually consumes somewhat more energy when listening for radio communications compared to when transmitting. The current draw for the CPU is much lower. The low-power CPU MSP430F1611 draws 330 μA at 1 MHz and 1.1 μA in standby mode [38].

Turning off the radio as much as possible when it is not transmitting or receiving is one of the most important factors to increase node lifetime.

One approach to such radio duty cycling is Time Division Multiple Access (TDMA), in which the time is divided into slots. A schedule is used to specify in which slots each node may send data and in which slots it should listen for data. Since the radio only needs to be turned on in some slots, TDMA can be a very energy efficient way to handle the radio communication. Each schedule is a compromise between energy consumption and throughput. The fewer slots where the nodes have their radio on, the lower the energy consumption and the lower the possible throughput. The drawback with TDMA is that the nodes must synchronize their clocks to know when each slot starts. There are many solutions for clock synchronization but most of them are costly in terms of energy consumption due to the required communications [24]. TDMA also makes it difficult to adapt to varying traffic loads without updating the schedule in all involved nodes

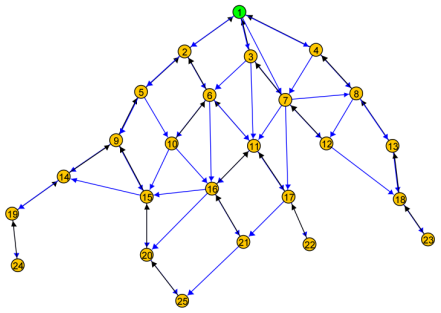


Figure 1.2: Multihop Sensor Network.

which is not easily done instantly.

Another approach to radio duty cycling is Low-Power Listening (LPL), in which the nodes periodically wake up and listen for radio transmissions [31]. In contrast to TDMA, the nodes do not know when the other nodes will be awake and ready to receive data. Instead, when a node wants to send data, it sends short radio packets in sequence until the receiving node wakes up and acknowledges the radio packet. At that time both nodes are awake and the data can be sent. How often the nodes wake up and listen for transmissions is a compromise between lifetime and latency. This approach has the advantage of being completely asynchronous and does not require any time synchronization among the nodes.

To compensate for the short communication range of the radio, the sensor nodes can forward packets between each other. By forming multihop networks (Figure 1.2), the communication range can be increased and the reliability improved. Especially in-door the connectivity between nodes is very difficult to predict beforehand because the environment interferes with the radio waves in various ways [5]. By using multihop communication the nodes can forward radio packets via other nodes to avoid areas with bad connectivity. With only single-hop all nodes must be deployed in such a way that every node can reach the base station even in harsh environments, which might complicate deployment. Multihop, however, requires more complex software because the nodes need to be able to forward packets for other nodes. Furthermore, the nodes must listen for communication from other nodes, which increases the energy consumption.

Recently, standards have emerged for using IPv6 in these types of resource constrained systems. IPv6 solves the problem of having enough globally unique addresses for each sensor node and can put the sensor nodes directly on the Internet. Furthermore, using standards like IPv6/6LoWPAN allow interoperability between manufacturers of sensor networks and enable standard network tools to be used for administrating the networks.

1.2 Configuration of Wireless Sensor Networks

Wireless sensor network applications have specific objectives such as lifetime, throughput, latency, reliability that need to be satisfied. The application objectives might also change over time for example from maximizing lifetime to throughput. In general configuration and reconfiguration aim to optimize the performance of the sensor network towards the application objectives. This section describes different configuration parameters for sensor network applications.

1.2.1 Example Applications

Below we discuss two example sensor network applications and how some configuration choices affect their performance.

Example: Condition Monitoring with Bulk Transfer

One example application for wireless sensor networks is condition monitoring of industrial motors. This example is based on a real case scenario from the ESNA project [32] where sensor nodes measured vibrations on electrical motors in a sawmill. The sensor nodes are attached to motors and periodically sample vibration data that are sent to a base station for analysis. The vibration data is typically quite large and requires a stream of radio packets to be delivered. Due to the harsh radio environment in factories, multihop communication is often needed to provide connectivity with the base station. Furthermore, a long lifetime is a requirement and the nodes must be very energy efficient because changing batteries or replacing nodes is expensive or even unfeasible in factories. Ideally the nodes should survive at least the expected lifetime of the motor.

This application raises a number of questions about how it should be configured. There are a number of basic parameters that have a direct effect on the performance. The MAC layer duty cycle, that specifies the time the radio is turned off, has a direct impact not only on the energy consumption but also on the throughput and latency. The more the radio is turned off, the lower the throughput and the higher the latency. The send rate when sending the stream of radio packets can cause congestion if the packets are sent too fast. If the packets are sent slower than necessary, the network might be kept active for a longer time resulting in higher energy consumption. The routing mechanism can be configured for stable networks where routes rarely need to be changed once setup, or for highly dynamic networks where nodes are added or removed on the fly, or where temporary radio interference force different routes to be used at different times. Highly dynamic networks might require more active communication in order to detect connectivity changes. The parameters mentioned above are only a few examples of configuration parameters for this type of sensor networks.

Example: Surveillance Network

Another example of a wireless sensor network application is surveillance of buildings where the sensor nodes are used to detect movement. In this case the most important objective is to deliver any alarms to the base station as soon as possible. Although network lifetime is important, reliability and short latency is even more important.

1.2.2 Static Configuration is Not Enough

A common approach to develop sensornet applications is simply to use a static configuration in the nodes for the expected characteristics of the environment (for example radio duty cycle, number of retransmissions) which overall works quite well but might be suboptimal in terms of network lifetime.

In the first example above for condition monitoring, the normal mode when all nodes are idle should be as energy efficient as possible. This means among other things that the radios should be turned off as much as possible because there will be little communication. After a node has performed a measurement, a stream of data packets needs to be delivered to the base station, possibly through a number of forwarding nodes. The data delivery can take a long time in the normal low-power settings. Reconfiguring the network for high throughput during a short time can be much more energy efficient since the data is delivered much faster after which the network can resume the normal low-power mode. By adapting to the traffic load, the sensor network can save a lot of energy.

Typically there will be other types of communication apart from the sensor data delivery in the network such as routing, statistics, or keep alive messages. These other types of data messages are sent infrequently and should therefore not cause the network to enter high throughput mode. This makes it more complex to implement adaptation of the network configuration. For example, the node can gather statistics in order to adapt to different throughput which is a more generic solution but this might make the adaption slower. Or information about the various message types can be embedded in the network stack in order to adapt to high throughput only for measurement data deliveries. However, this embeds application specific knowledge into the network stack which makes it harder to reuse the software and also implement changes if the application objectives change.

In the second example about surveillance, there is a similar situation. In the normal mode when there is no alarm the network should be as energy efficient as possible but with the added constraint that all alarms should be delivered quickly to the base station. Furthermore, the traffic load is dependent on the number of alarming nodes, and if many nodes try to send alarms at the same time because something has been detected, the

network might become congested, causing no alarms at all to reach the base station. Reconfiguring the network for lower latency during times with much activity would allow the sensor network to use a more energy conserving configuration during times with low activity, thus prolonging the life-time of the network.

There are many more reasons for a node to adapt its configuration based on its current conditions. For example, if a node reaches a low level of energy or finds itself in a place with bad connectivity, it might reconfigure itself to take a less active part in forwarding packets for other nodes to conserve its remaining energy. This all depends on the application objectives.

Often it is not possible to test the sensor network applications in realistic conditions before deployment. Test runs in test beds might be quite different from real-world deployments. Therefore it can be difficult to find good configuration parameters for all nodes before deployment and each node might end up with different conditions after deployment. To improve the performance we would like the configuration to adapt to the current conditions of the node where the current conditions include the surrounding environment.

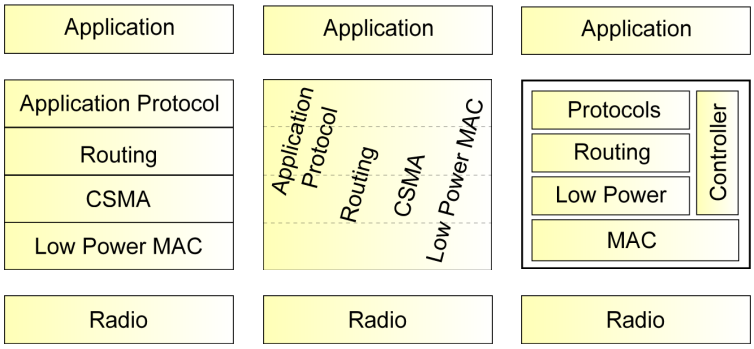
1.2.3 Approaches for Adaptive Configuration

For a sensor node to adapt to the current conditions it needs to be aware of the surrounding environment. The sensor nodes use self-monitoring to monitor the environment which might include anything from simple network statistics collection to actively probing the hardware (see Section 1.3.2). What to monitor and when depends on the available adaptations and application objectives.

There are many approaches to achieve adaptation in wireless sensor networks. This section describes some approaches and concludes with a description of the configuration architecture developed during the thesis.

Software Updates

One approach to handle suboptimal configuration is to reprogram the sensor networks with updated code (see Section 1.3.1). After measuring the sensor network performance in order to find what needs to be reconfigured, updated code is pushed into the network. This usually means that all nodes use the same code and configuration. Reprogramming a sensor network is a very flexible solution but also very costly in terms of energy and might take a long time to perform depending on the underlying communication mechanism. The task of how to best reprogram sensor networks is a research challenge in itself.



(a) Layered Communication Stack. (b) Cross-layer Communication Stack. (c) Specialized Architecture.

Figure 1.3: Network Stacks.

Adaptive Communication Layers

The network stack is traditionally described as a layered communication stack where each layer implements a specific functionality such as radio duty cycling, reliable packet delivery, routing (Figure 1.3(a)). This is a nice clean model where each layer can be a reusable and interchangeable software module. Each such software module can be adaptive by itself.

One example of adaptive layer is the adaptation of radio duty cycle described earlier: in times where there is lot of communication the radio duty cycle is reconfigured to decrease latency, and in quiet situations, the duty cycle is reconfigured to decrease power consumption. X-MAC is an asynchronous low-power listening MAC protocol where the nodes periodically wake up and listen for transmissions. How often the nodes wake up determines the energy efficiency and latency in the network. Buettner et al. describe an adaptive version of X-MAC where the low-power MAC layer adapts the sleep time to the traffic load based on configured minimal and maximal sleep time [3].

Many algorithms also include internal adaptation mechanisms. For example, the dissemination protocol Trickle uses beacons to distribute the nodes' current code version, and detect when an updated version is available [25]. The beacon interval is adapted based on updates: frequent updates give short beacon intervals, infrequent updates give large beacon intervals. The configuration in this particular example consists of the minimum and maximum intervals.

These examples are single-layer adaptations. Moldeklev et al. describe how optimizations in different layers worked against each other in some situations and drastically reduced throughput in TCP/IP over ATM networks [27].

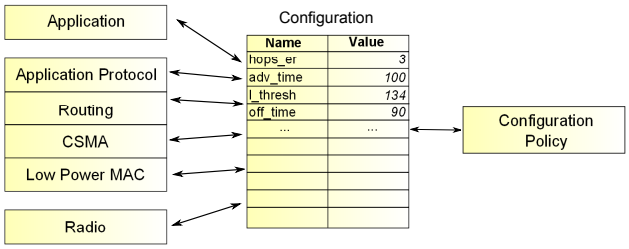


Figure 1.4: Full-system Configuration Architecture.

Cross-layer Design

The idea with cross-layer design is to allow the layers to know about each other and use information from other layers for their own adaptations (Figure 1.3(b)). Cross-layer information allows the layers to fully work together but at the cost of interweaving the layers, which makes the software more difficult to maintain and update. Cross-layer designs have been criticized as leading to "spaghetti design" [17]. Because knowledge about the different layers is integrated into the system, it is usually not possible simply to exchange a layer or re-use a system implementation for a different application.

The adaptive version of the X-MAC protocol is an example of a single-layer adaptation in the network stack. For example, if a bulk transfer temporarily causes a high-volume data transfer, X-MAC will react to the increase in traffic over time by adjusting the radio duty cycle, and afterwards adjust the duty cycle back to the normal low-power condition when the traffic decreases. If information is shared between the layers, the MAC layer could know when a high-volume data transfer is in progress. Both time and energy could be saved by directly switching between low and high throughput instead of adapting to the perceived traffic load.

Specialized Architectures

A higher-level approach with respect to cross-layer designs, is to use specialized architectures (Figure 1.3(c)). These are created for special purposes such as coordinating energy consumption. Klues et al. describe an architecture with a mechanism to coordinate the radio power management based on the application requirements [18].

Although such architectures provide a clean model where the layers are not tightly coupled, they tend to focus on specific goals such as energy efficiency by coordinating radio power management, not for general application objectives.

Architectures for Full-System Configuration

For optimization purposes, it would be preferable to be able to focus on different metrics as is possible with cross-layer designs. But from a development perspective it would be preferable to avoid cross-layer interaction to ease development, software reuse, and software updates as is possible with specialized architectures.

Paper C presents Chi, an architecture for full-system configuration. This architecture enables cross-layer optimizations but keeps the layers decoupled by separating logic from configuration and information such as network statistics. Figure 1.4 illustrates the layout of the architecture. Each layer publishes configuration parameters and system information using a generalized programming abstraction in a blackboard component that is used for both notifications and control. Separate software components called configuration policies are responsible for coordinating the sensor node configuration and optimize towards the application objectives. The configuration policies adapt to changing system information and reconfigure the system as needed. In contrast to the cross-layer designs described in Section 1.2.3, the individual layers in this approach are unaware about the other layers and individual layers can with replaced without affecting the other layers. All such knowledge is embedded in the configuration policies. Only the configuration policies need to be updated to change the sensor node configuration or adjust the network for better performance. A configuration policy might be implemented in plain C as if-then-else statements, but might also be more advanced like a rule-engine enforcing a set of rules.

1.3 Challenges and Research Questions

There are many challenges in the area of wireless sensor networks. This section discusses some of the challenges that are relevant for this thesis.

1.3.1 Software Updates

A common problem when deploying sensor networks is that the network behaves differently when deployed as compared to tests in lab settings. This could be due to different conditions such as radio environment or different deployment setup. The application objective might also change over time. Collecting the nodes to reprogram them is very time-consuming and might even be infeasible. Therefore it is important to include mechanisms that can update the code on the nodes or reconfigure the network after deployment in an energy efficient manner. The problem is twofold: distributing the updates over radio to the nodes, and reprogramming the node once the updates are available at the nodes. This thesis addresses the latter. Paper A presents a dynamic linker for modular code update and compares it to

full system-image replacement and two variants of virtual machines in terms of energy cost.

1.3.2 Self-monitoring

As mentioned earlier, inexpensive and low quality hardware can be used in the sensor nodes to reduce the cost. In deployment of a large number of nodes, some substandard nodes might be acceptable to reduce the average cost of each node. With enough nodes, other nodes can fill in for the substandard nodes while the substandard nodes can be used with reduced functionality. The problem is to identify the deficient nodes. One approach we have used is self-probing the hardware. Each hardware component is activated in turn while measuring all sensors to see if any of the sensors react on the active component.

Another problem is that some software issues are difficult to detect. Problems such as failing to turn off the radio might temporary not impair the functionality of the network. The performance might even appear to be improved but at a cost of higher energy consumption resulting in depleted energy sources earlier than expected.

Furthermore, it is difficult to predict the behaviour in real-world deployments from test runs in lab settings. Test runs tend to be smaller, run for shorter time and in a different radio environment [23]. Therefore, it is important to collect statistics after deployment in order to evaluate the performance.

Paper B highlights the need for self-monitoring in sensor networks and presents mechanisms for detecting hardware and software problems.

1.3.3 Adaptive Configuration in Wireless Sensor Networks

There are two main challenges when configuring wireless sensor networks: creating good configurations that satisfy the sensor network application objectives; and developing on-node software that utilizes the configurations. We focus on the latter: when we have a configuration, developing an architecture on the sensor nodes that uses the configuration. As discussed earlier in Section 1.2.2, adaptation is desired to improve performance. The configuration architecture should support adaptation and cross-layer optimizations on the nodes. Another important issue is how these adaptive configurations are represented. As future work, we plan to generate configurations using optimization techniques from the area of artificial intelligence.

We suggest logic is separated from configuration and system information such as network statistics. Component configuration is coordinated by configuration policies that handle both initial configuration and reconfiguration to achieve system adaptivity. Paper C describes a configuration architecture, developed based on these ideas.

1.4 Contributions and Results

The main contributions of this thesis are the concept of configuration policies and the separation of logic from configuration and information that enables cross-layer optimizations without tight coupling between components. A configuration architecture for full-system configuration has been designed, implemented and evaluated based on these ideas.

Furthermore, the need for self-configuration and self-monitoring has been highlighted based on experiences from real-world deployments. We have developed and evaluated mechanisms to address this need.

Throughout this thesis, all development has been based on the Contiki operating system. Furthermore, I have contributed to the continuous development of Contiki. Contiki is an open source operating system for resource constrained embedded systems and is used by both academia and industry.

Chapter 2

Summary of Papers

2.1 Paper A: Run-time dynamic linking for reprogramming wireless sensor networks

Adam Dunkels, Niclas Finne, Joakim Eriksson, Thiemo Voigt. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (ACM SenSys 2006)*, Boulder, Colorado, USA, November 2006.

Summary. The nodes in wireless sensor networks often need to be updated with new software to fix software bugs or add new functionality. This makes dynamic reprogramming of wireless sensor networks an important feature. This paper presents a dynamic linker and loader for updating sensor nodes during run-time, and a compact ELF format that with minimal differences towards standard ELF format reduces the module size to half. Using a basic energy efficiency model, we compare system updates with the dynamic linker to three other methods for reprogramming sensor nodes: full image replacement, an application specific virtual machine, and a Java virtual machine. We validate the comparison using energy measurements on real hardware.

Contribution. The paper shows that a standard mechanism for dynamic linking of native code is feasible even in resource constrained sensor nodes. Furthermore, the evaluation suggests that a combination of virtual machine code and native code may provide a more energy efficient alternative than using only native code or only virtual machine code.

My contribution. I assisted with the energy measurements for the evaluation. This paper is also included as a reference of my contributions to Contiki.

2.2 Paper B: Experiences from two sensor network deployments: self-monitoring and self-configuration keys to success

Niclas Finne, Joakim Eriksson, Adam Dunkels, Thiemo Voigt. In *Proceedings of the 6th International Conference on Wired/Wireless Internet Communications (WWIC 2008)*, Tampere, Finland, May 2008.

Summary. The paper describes two sensor network surveillance deployments performed together with the Swedish defense unit Markstridsskolan: The first is an in-door deployment in a factory complex and the second is a combined in-door and out-door deployment in an urban environment. Both deployments spread over an area that required multihop networking to transfer the sensor data from the sensor nodes to the base station. During the deployments we also found that the sensors on a few nodes reacted on radio transmissions and caused unwanted false alarms, something we had not previously seen. Based on experiences from these deployments, two architectures are presented and evaluated: hardware self-test to detect and adapt for hardware problems, and energy profiling to detect software problems during run-time. The first architecture is a self-monitoring mechanism that probes the hardware for problems and separates the nodes with and without hardware problems. The nodes with hardware problems can then configure themselves to handle the problem, which usually means somewhat reduced functionality to avoid interfering. The second architecture uses the built-in energy estimator in the Contiki OS to monitor the energy usage and compare it to an application specific energy profile. If a software problem causes the CPU or radio to remain awake, the problem is detected when the energy consumption deviates from the energy profile. This type of problem is normally not easily detected because the application might function as normal until it runs out of energy.

Contribution. The paper highlights the importance of self-monitoring and self-configuration in wireless sensor networks, and presents mechanisms to address this.

My contribution. I performed both deployments together with Joakim Eriksson, SICS and Mikael Axelsson, Swedish Defence. I wrote parts of the paper and worked on the implementation, design and experiments. I presented the paper at WWIC 2008.

2.3 Paper C: Improving sensornet performance by separating system configuration from system logic

Niclas Finne, Joakim Eriksson, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt. In *Proceedings of the 7th European Conference on Wireless Sensor Networks (EWSN 2010)*, Coimbra, Portugal, February 2010.

Summary. The paper presents Chi, an architecture for full-system configuration and policy-based optimization in sensor networks. In Chi cross-layer optimization is done using generalized programming abstractions. Components such as the low-power MAC layer, routing layer, publish their configuration and system state parameters to a central configuration store. The main design principle is that the components do not need to be aware of the parameters of other components. Instead, all such knowledge is handled by external configuration policies that are used to coordinate configuration and reconfiguration. These configuration policies optimize the system towards the application objectives of which the components do not need to be aware. This keeps a clean separation between the components and only the configuration policies need to be changed if the application objectives are changed. Chi's dynamic properties make it possible to switch configuration policies and to add new configuration and state parameters during run-time. The paper shows that Chi solves the problem of cross-layer optimization for sensornet systems using a generalized programming abstraction. The abilities of Chi are demonstrated in three case studies: condition monitoring, TCP over a power-saving MAC protocol, and aggregation of multiple duty cycles.

Contribution. The paper shows how a separation of configuration from logic can improve the sensor network performance as much as specialized architectures, while maintaining a clear separation of concerns.

My contribution. I worked on the implementation, design and experiments. I wrote parts of the paper and presented the paper at EWSN 2010.

Chapter 3

Related Work

The related work discussion is divided in three sections: software updates, self-monitoring, and adaptation in sensor networks.

3.1 Software Updates

Reprogramming the nodes in a sensor network can be done in several ways. Full system image replacement is a common way where a new system image is sent to each node to replace the existing system [13, 15]. This is an easy approach, but requires that the full system image is distributed even if only parts of the system have been modified. There are mechanisms that only distribute the difference to reduce the data size and reconstruct the full system image on the nodes, but these mechanisms require that all nodes have the same version of the system [14, 20, 33]. In contrast, our approach as described in Paper A only distributes and updates the modules that have changed. Marron et al. [30] also update only changed modules but unlike our approach that makes use of standard ELF format, their approach requires specialized tools and formats.

3.2 Self-monitoring

Self-monitoring is a necessity for both detecting problems and adapting for better performance. Minor software bugs and misconfigurations can radically decrease the lifetime of the sensornet application. Langendoen et al. describe experiences from a pilot sensor deployment for precision agriculture [23]. Like us, they highlight how important it is to monitor the performance and gather system statistics to detect problems early on. Arora et al. describe experiences from experimental deployments for intrusion detection [1]. Sensor nodes handled roughly and exposed to harsh environmental conditions like heat and moisture, fail frequently and in complex ways. Faults do not always result in lost nodes, but rather unexpected behavior

like increased radio communication or sending false alarms. In contrast to their work, we do not only identify problems but also suggest and implement mechanisms that detect both hardware and software problems.

Self-monitoring is not only needed for self-adaptation and system statistics. Many mechanisms inherently depend on updated information which require monitoring of the closest neighborhood. For instance, it is common to monitor the link quality to neighbors and use the quality estimations to make routing decisions and to handle failing routes. In RPL, the proposed standard routing protocol for IPv6 in low-power networks, an objective function ranks the neighbors based on some metrics [39]. Different objective functions use different metrics to rank the neighbors, which controls how the routing topology is built. It is often desirable to aggregate information from several layers. Fonseca et al. present a link estimator that uses information from several layers in the network stack to provide a link quality estimation to the neighbors [9].

3.3 Adaptation in Wireless Sensor Networks

3.3.1 Adaptive Layers

As already discussed in Section 1.2.3, the dissemination protocol Trickle adapts its advertisement beacons based on the update frequency [25]. Buettner et al. describe an adaptive version of X-MAC where the MAC layer adapts the radio duty cycle based on the traffic load [3]. Other examples are the scheduled channel polling MAC protocol, which adapts to varying traffic load [42]; the MintRoute protocol, which adapts its routing based on the communication conditions [40]; and the Z-MAC protocol, which reconfigures itself between TDMA and CSMA based on the channel conditions [34].

3.3.2 Cross-layer Design

Multiple forms of cross-layer designs have been proposed to improve sensornet performance. For instance, Cui et al. propose a combined optimization over the link, MAC, and networking layer [6]. Van Hoesel et al. [12] and Madan et al. [26] propose approaches with tight integration of medium access and routing for higher energy efficiency.

In vertically integrated systems the layers are designed to coordinate with each other. Koala [28] and Dozer [4] are examples of two such systems with duty cycles below 1% in low rate data gathering applications. These systems, however, are designed for specific network patterns, which makes them difficult to use for other traffic patterns.

Cross-layer designs have also been developed for specific applications. For instance, Song et al. propose a cross-layer approach for target tracking by which the cross-layer interactions includes the application [36].

3.3.3 Cross-layer Frameworks

Many frameworks have been presented for cross-layer interactions and information sharing to allow cross-layer optimizations.

As described earlier, Klues et al. describe an architecture for coordinating the radio power management [18]. Jurdak et al. present a cross-layer framework for optimizing the global power consumption [16].

Lachenmann et al. describe Levels, a programming abstraction that helps to coordinate the energy consumption in order to meet the application lifetime goals [22]. The software components describe their energy consumption for different energy levels and the system assigns an optimal energy level for each component based on the remaining lifetime.

TinyXXL is a language and framework that supports cross-layer interaction [21]. The components describe the parameters and system state that they wish to share with other components using an extension of the programming language nesC.

Köpke et al. suggest using a blackboard for component-based interactions [19]. The information is stored in a central information repository and accessed via an abstract API to decouple the components.

Although these approaches provide programming abstractions that help to decouple the software components while supporting interaction between the components, they lack the dynamic properties that would allow the adaptive configuration to be switched and new configuration parameters to be added during run-time, something that our approach allows as described in Paper C.

Chapter 4

Conclusions and Future Work

Our experiences from real-world deployments highlight the need for self-monitoring and self-configuration. In this thesis, we developed Chi, a configuration architecture for adaptive full-system configuration. The software components export their configuration parameters using a programming abstraction and are not required to have any knowledge of the parameters exported by other components. Instead, all such knowledge is in separate components that enforce configuration policies. This thesis shows that Chi can improve the sensor network performance as much as cross-layer optimizations without adding dependencies between the layers, thus enabling the use of standard and easily replaceable components. Furthermore, we show that dynamic module replacement using a standard mechanism for dynamic linking is feasible even in resource constrained sensor nodes which enables efficient software updates.

The concept of configuration policies is the foundation for the next step: how to generate adaptive configurations that can be used on the sensor nodes. Adaptive configurations can be changed simply by replacing a configuration policy while the rest of the system remains unchanged. We will continue with this approach and implement a more advanced rule engine for configuration policies. The configuration policies will be generated using optimization techniques from the area of artificial intelligence.

To optimize the sensor network performance, feedback is required both on the node for self-adaptation and outside the network for offline optimizations. Energy efficient self-monitoring is needed to provide the necessary feedback and is something we will study further.

Bibliography

[1] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks*, 46(5):605–634, 2004.

[2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *CM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, 10(4), August 2005.

[3] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 307–320, Boulder, Colorado, USA, 2006.

[4] N. Burri, P. von Rickenbach, and R. Wattenhofer. Dozer: ultra-low power data gathering in sensor networks. In *IPSN '07*, 2007.

[5] O. Chipara, G. Hackmann, C. Lu, W. D. Smart, and G. Roman. Practical modeling and prediction of radio coverage of indoor sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 339–349, New York, NY, USA, 2010. ACM.

[6] S. Cui, R. Madan, A. Goldsmith, and S. Lall. Joint routing, MAC, and link layer optimization in sensor networks with energy constraints. In *IEEE International Conference on Communications (ICC)*, Seoul, Korea, May 2005.

[7] A. Dunkels. The Contiki Operating System. Web page. Visited 2011-01-20. `http://www.sics.se/contiki/`

[8] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based online energy estimation for sensor nodes. In *Proceedings of the Fourth*

Workshop on Embedded Networked Sensors (Emnets IV), Cork, Ireland, June 2007.

[9] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis. Four-bit wireless link estimation. In *Sixth Workshop on Hot Topics in Networks (ACM HotNets-VI)*, Atlanta, Georgia, USA, November 2007.

[10] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. SOS: A dynamic operating system for sensor networks. In *MobiSys '05*, 2005.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[12] L. Van Hoesel, T. Nieberg, J. Wu, and P. Havinga. Prolonging the Lifetime of Wireless Sensor Networks by Cross-Layer Interaction. *IEEE Wireless Communications*, 11(6):78–86, 2004.

[13] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. SenSys'04*, Baltimore, Maryland, USA, November 2004.

[14] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *IEEE SECON*, October 2004.

[15] J. Jeong, S. Kim, and A. Broad. Network reprogramming. TinyOS documentation, 2003. Visited 2006-04-06. `http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf`

[16] R. Jurdak, P. Baldi, and C. Videira Lopes. Adaptive low power listening for wireless sensor networks. *IEEE Transactions on Mobile Computing*, 6(8):988–1004, 2007. ISSN: 1536-1233

[17] V. Kawadia and P. Kumar. A cautionary perspective on cross-layer design. *Wireless Communications, IEEE*, 12(1):3–11, 2005. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1404568`

[18] K. Klues, G. Xing, and C. Lu. Link layer support for unified radio power management in wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN '07)*, pages 460–469, Cambridge, Massachusetts, USA, 2007.

[19] A. Köpke, V. Handziski, J.-H. Hauer, and H. Karl. Structuring the information flow in component-based protocol implementations for wireless sensor nodes. In *Proc. of Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN)*, Berlin, Germany, January 2004.

[20] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proc. EWSN'05*, 2005.

[21] A. Lachenmann, P. Marrón, D. Minder, M. Gauger, O. Saukh, and K. Rothermel. TinyXXL: Language and runtime support for cross-layer interactions. In *Proceedings of the Third Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, pages 178–187, 2006.

[22] A. Lachenmann, P. Marrón, D. Minder, and K. Rothermel. Meeting lifetime goals with energy levels. In *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems*, 2007.

[23] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, April 2006. IEEE.

[24] C. Lenzen, P. Sommer, and R. Wattenhofer. Optimal Clock Synchronization in Networks. In *7th ACM Conference on Embedded Networked Sensor Systems (SenSys), Berkeley, California, USA*, November 2009.

[25] 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'04*, March 2004.

[26] R. Madan, S. Cui, S. Lall, and A. Goldsmith. Cross-layer design for lifetime maximization in interference-limited wireless sensor networks. In *IEEE INFOCOM*, March 2005.

[27] K. Moldeklev and P. Gunningberg. How a large ATM MTU causes deadlocks in TCP data transfers. *IEEE/ACM Trans. Netw.*, 3:409–422, August 1995. ISSN: 1063-6692`http://dx.doi.org/10.1109/90.413215`

[28] R. Musaloiu-E., C-J. M. Liang, and A. Terzis. Koala: Ultra-Low Power Data Retrieval in Wireless Sensor Networks. In *IPSN '08*, 2008.

[29] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Tampa, Florida, USA, November 2006.

[30] P. Marrón et al. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *European Workshop on Wireless Sensor Networks*, 2006.

[31] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM Press.

[32] The ESNA project. European sensor network architecture. Web page. Visited 2011-05-06. `http://www.sics.se/esna/`

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

[34] I. Rhee, A. Warrier, M. Aia, and J. Min. Z-MAC: a hybrid MAC for wireless sensor networks. *ACM SenSys*, pages 90–101, 2005.

[35] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004.

[36] L. Song and D. Hatzinakos. A cross-layer architecture of wireless sensor networks for target tracking. *IEEE/ACM Transactions on Networking (TON)*, 15(1):145–158, 2007.

[37] Texas Instruments. CC2420 Datasheet (rev. 1.4), 2007. `http://focus.ti.com/docs/prod/folders/print/cc2420.html`

[38] Texas Instruments. MSP430F1611 Datasheet, 2009. `http://focus.ti.com/docs/prod/folders/print/msp430f1611.html`

[39] T. Winter (Ed.), P. Thubert (Ed.), and RPL Author Team. RPL: IPv6 Routing Protocol for Low power and Lossy Networks. Internet Draft draft-ietf-roll-rpl-19, work in progress.

[40] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of ACM SenSys'03*, Los Angeles, California, USA, 2003.

[41] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, New York, NY, USA, June 2002.

[42] W. Ye, F. Silva, and J. Heidemann. Ultra-low duty cycle mac with scheduled channel polling. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 321–334, New York, NY, USA, 2006. ACM Press.

Part II

Included Papers

Chapter 5

Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks

Adam Dunkels, Niclas Finne, Joakim Eriksson, Thiemo Voigt
Swedish Institute of Computer Science, Box 1263,
SE-16429 Kista, Sweden
adam@sics.se, nfi@sics.se, joakime@sics.se, thiemo@sics.se

5.1 Abstract

From experience with wireless sensor networks it has become apparent that dynamic reprogramming of the sensor nodes is a useful feature. The resource constraints in terms of energy, memory, and processing power make sensor network reprogramming a challenging task. Many different mechanisms for reprogramming sensor nodes have been developed ranging from full image replacement to virtual machines.

We have implemented an in-situ run-time dynamic linker and loader that use the standard ELF object file format. We show that run-time dynamic linking is an effective method for reprogramming even resource constrained wireless sensor nodes. To evaluate our dynamic linking mechanism we have implemented an application-specific virtual machine and a Java virtual machine and compare the energy cost of the different linking and execution models. We measure the energy consumption and execution time overhead on real hardware to quantify the energy costs for dynamic linking.

Our results suggest that while in general the overhead of a virtual machine is high, a combination of native code and virtual machine code provide good energy efficiency. Dynamic run-time linking can be used to update the native code, even in heterogeneous networks.

5.2 Introduction

Wireless sensor networks consist of a collection of programmable radio-equipped embedded systems. The behavior of a wireless sensor network is encoded in software running on the wireless sensor network nodes. The software in deployed wireless sensor network systems often needs to be changed, both to update the system with new functionality and to correct software bugs. For this reason dynamically reprogramming of wireless sensor network is an important feature. Furthermore, when developing software for wireless sensor networks, being able to update the software of a running sensor network greatly helps to shorten the development time.

The limitations of communication bandwidth, the limited energy of the sensor nodes, the limited sensor node memory which typically is on the order of a few thousand bytes large, the absence of memory mapping hardware, and the limited processing power make reprogramming of sensor network nodes challenging.

Many different methods for reprogramming sensor nodes have been developed, including full system image replacement [12, 14], approaches based on binary differences [13, 15, 29], virtual machines [16, 17, 18], and loadable native code modules in the first versions of Contiki [3] and SOS [10]. These methods are either inefficient in terms of energy or require non-standard data formats and tools.

The primary contribution of this paper is that we investigate the use of standard mechanisms and file formats for reprogramming sensor network nodes. We show that in-situ dynamic run-time linking and loading of native code using the ELF file format, which is a standard feature on many operating systems for PC computers and workstations, is feasible even for resource-constrained sensor nodes. Our secondary contribution is that we measure and quantify the energy costs of dynamic linking and execution of native code and compare it to the energy cost of transmission and execution of code for two virtual machines: an application-specific virtual machine and the Java virtual machine.

We have implemented a dynamic linker in the Contiki operating system that can link, relocate, and load standard ELF object code files. Our mechanism is independent of the particular microprocessor architecture on the sensor nodes and we have ported the linker to two different sensor node platforms with only minor modifications to the architecture dependent module of the code.

To evaluate the energy costs of the dynamic linker we implement an application specific virtual machine for Contiki together with a compiler for a subset of Java. We also adapt the Java virtual machine from the lejOS system [6] to run under Contiki. We measure the energy cost of reprogramming and executing a set of program using dynamic linking of native code and the two virtual machines. Using the measurements and

a simple energy consumption model we calculate break-even points for the energy consumption of the different mechanisms. Our results suggest that while the execution time overhead of a virtual machine is high, a combination of native code and virtual machine code may give good energy efficiency.

The remainder of this paper is structured as follows. In Section 5.3 we discuss different scenarios in which reprogramming is useful. Section 5.4 presents a set of mechanisms for executing code inside a sensor node and in Section 5.5 we discuss loadable modules and the process of linking, relocating, and loading native code. Section 5.6 describes our implementation of dynamic linking and our virtual machines. Our experiments and the results are presented in Section 5.7 and discuss the results in Section 5.8. Related work is reviewed in Section 5.9. Finally, we conclude the paper in Section 5.10.

5.3 Scenarios for Software Updates

Software updates for sensor networks are necessary for a variety of reasons ranging from implementation and testing of new features of an existing program to complete reprogramming of sensor nodes when installing new applications. In this section we review a set of typical reprogramming scenarios and compare their qualitative properties.

5.3.1 Software Development

Software development is an iterative process where code is written, installed, tested, and debugged in a cyclic fashion. Being able to dynamically reprogram parts of the sensor network system helps shorten the time of the development cycle. During the development cycle developers typically change only one part of the system, possibly only a single algorithm or a function. A sensor network used for software development may therefore see large amounts of small changes to its code.

5.3.2 Sensor Network Testbeds

Sensor network testbeds are an important tool for development and experimentation with sensor network applications. New applications can be tested in a realistic setting and important measurements can be obtained [36]. When a new application is to be tested in a testbed the application typically is installed in the entire network. The application is then run for a specified time, while measurements are collected both from the sensors on the sensor nodes, and from network traffic.

For testbeds that are powered from a continuous energy source, the energy consumption of software updates is only of secondary importance. Instead, qualitative properties such as ease of use and flexibility of the software

update mechanism are more important. Since the time required to make an update is important, the throughput of a network-wide software update is of importance. As the size of the transmitted binaries impact the throughput, the binary size still can be used as an evaluation metric for systems where throughput is more important than energy consumption.

5.3.3 Correction of Software Bugs

The need for correcting software bugs in sensor networks was early identified [5]. Even after careful testing, new bugs can occur in deployed sensor networks caused by, for example, an unexpected combination of inputs or variable link connectivity that stimulate untested control paths in the communication software [28].

Software bugs can occur at any level of the system. To correct bugs it must therefore be possible to reprogram all parts of the system.

5.3.4 Application Reconfiguration

In an already installed sensor network, the application may need to be reconfigured. This includes change of parameters, or small changes in the application such as changing from absolute temperature readings to notification when thresholds are exceeded [24]. Even though reconfiguration not necessarily include software updates [23], application reconfiguration can be done by reprogramming the application software. Hence software updates can be used in an application reconfiguration scenario.

5.3.5 Dynamic Applications

There are many situations where it is useful to replace the application software of an already deployed sensor network. One example is the forest fire detection scenario presented by Fok et al. [7] where a sensor network is used to detect a fire. When the fire detection application has detected a fire, the fire fighters might want to run a search and rescue application as well as a fire tracking application. While it may possible to host these particular applications on each node despite the limited memory of the sensor nodes, this approach is not scalable [7]. In this scenario, replacing the application on the sensor nodes leads to a more scalable system.

5.3.6 Summary

Table 5.1 compares the different scenarios and their properties. *Update fraction* refers to what amount of the system that needs to be updated for every update, *update level* to at what levels of the system updates are likely to occur, and *program longevity* to how long an installed program will be expected to reside on the sensor node.

Scenario	Update frequency	Update fraction	Update level	Program longevity
Development	Often	Small	All	Short
Testbeds	Seldom	Large	All	Long
Bug fixes	Seldom	Small	All	Long
Reconfig.	Seldom	Small	App	Long
Dynamic Application	Often	Small	App	Long

Table 5.1: Qualitative comparison between different reprogramming scenarios.

5.4 Code Execution Models and Reprogramming

Many different execution models and environments have been developed or adapted to run on wireless sensor nodes. Some with the notion of facilitating programming [1], others motivated by the potential of saving energy costs for reprogramming enabled by the compact code representation of virtual machines [17]. The choice of the execution model directly impacts the data format and size of the data that needs to be transported to a node. In this section we discuss three different mechanisms for executing program code inside each sensor node: script languages, virtual machines, and native code.

5.4.1 Script Languages

There are many examples of script languages for embedded systems, including BASIC variants, Python interpreters [20], and TCL machines [1]. However, most script interpreters target platforms with much more resources than our target platforms and we have therefore not included them in our comparison.

5.4.2 Virtual Machines

Virtual machines are a common approach to reduce the cost of transmitting program code in situations where the cost of distributing a program is high. Typically, program code for a virtual machine can be made more compact than the program code for the physical machine. For this reason virtual machines are often used for programming sensor networks [16, 17, 18, 21].

While many virtual machines such as the Java virtual machine are generic enough to perform well for a variety of different types of programs, most virtual machines for sensor networks are designed to be highly configurable in order to allow the virtual machine to be tailored for specific applications. In effect, this means that parts of the application code is implemented as virtual machine code running on the virtual machine, and other parts of the

application code is implemented in native code that can be used from the programs running on the virtual machine.

5.4.3 Native Code

The most straightforward way to execute code on sensor nodes is by running native code that is executed directly by the microcontroller of the sensor node. Installing new native code on a sensor node is more complex than installing code for a virtual machine because the native code uses physical addresses which typically need to be updated before the program can be executed. In this section we discuss two widely used mechanisms for reprogramming sensor nodes that execute native code: full image replacement and approaches based on binary differences.

Full Image Replacement

The most common way to update software in embedded systems and sensor networks is to compile a complete new binary image of the software together with the operating system and overwrite the existing system image of the sensor node. This is the default method used by the XNP and Deluge network reprogramming software in TinyOS [11].

The full image replacement does not require any additional processing of the loaded system image before it is loaded into the system, since the loaded image resides at the same, known, physical memory address as the previous system image. For some systems, such as the Scatterweb system code [31], the system contains both an operating system image and a small set of functions that provide functionality for loading new operating system images. A new operating system image can overwrite the existing image without overwriting the loading functions. The addresses of the loading functions are hard-coded in the operating system image.

Diff-based Approaches

Often a small update in the code of the system, such as a bugfix, will cause only minor differences between in the new and old system image. Instead of distributing a new full system image the binary differences, deltas, between the modified and original binary can be distributed. This reduces the amount of data that needs to be transferred. Several types of diff-based approaches have been developed [13, 15, 29] and it has been shown that the size of the deltas produced by the diff-based approaches is very small compared to the full binary image.

5.5 Loadable Modules

A less common alternative to full image replacement and diff-based approaches is to use loadable modules to perform reprogramming. With loadable modules, only parts of the system need to be modified when a single program is changed. Typically, loadable modules require support from the operating system. Contiki and SOS are examples of systems that support loadable modules and TinyOS is an example of an operating system without loadable module support.

A loadable module contains the native machine code of the program that is to be loaded into the system. The machine code in the module usually contains references to functions or variables in the system. These references must be resolved to the physical address of the functions or variables before the machine code can be executed. The process of resolving those references is called linking. Linking can be done either when the module is compiled or when the module is loaded. We call the former approach pre-linking and the latter dynamic linking. A pre-linked module contains the absolute physical addresses of the referenced functions or variables whereas a dynamically linked module contains the symbolic names of all system core functions or variables that are referenced in the module. This information increases the size of the dynamically linked module compared to the pre-linked module. The difference is shown in Figure 5.1. Dynamic linking has not previously been considered for wireless sensor networks because of the perceived run-time overhead, both in terms of execution time, energy consumption, and memory requirements.

The machine code in the module usually contains references not only to functions or variables in the system, but also to functions or variables within the module itself. The physical address of those functions will change depending on the memory address at which the module is loaded in the system. The addresses of the references must therefore be updated to the physical address that the function or variable will have when the module is loaded. The process of updating these references is known as relocation. Like linking, relocation can be done either at compile-time or at run-time.

When a module has been linked and relocated the program loader loads the module into the system by copying the linked and relocated native code into a place in memory from where the program can be executed.

5.5.1 Pre-linked Modules

The machine code of a pre-linked module contains absolute addresses of all functions and variables in the system code that are referenced by the module. Linking of the module is done at compile time and only relocation is performed at run-time. To link a pre-linked module, information about the physical addresses of all functions and variables in the system into which

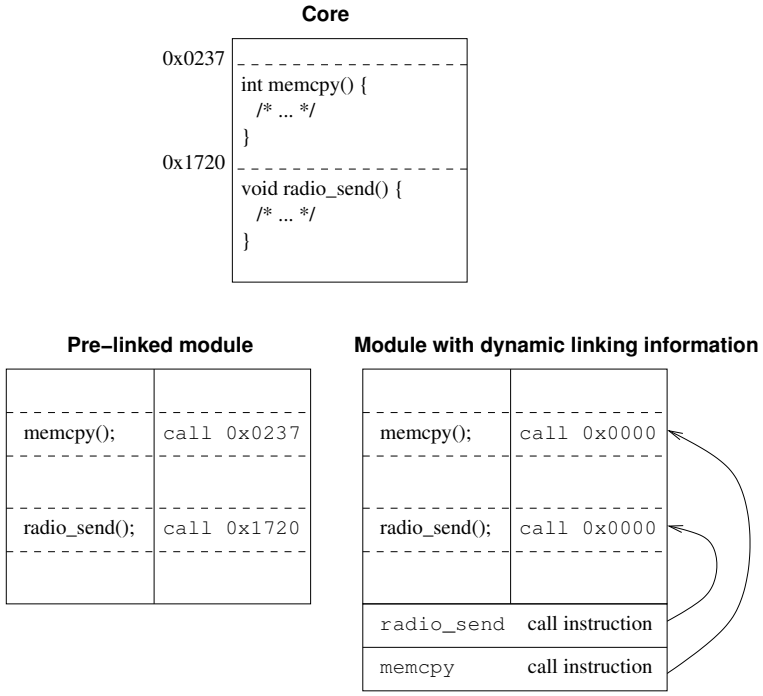


Figure 5.1: The difference between a pre-linked module and a module with dynamic linking information: the pre-linked module contains physical addresses whereas the dynamically linked module contains symbolic names.

the module is to be loaded must be available at compile time.

There are two benefits of pre-linked modules over dynamically linked modules. First, pre-linked modules are smaller than dynamically linked modules which results in less information to be transmitted. Second, the process of loading a pre-linked module into the system is less complex than the process of linking a dynamically linked module. However, the fact that all physical addresses of the system core are hard-coded in the pre-linked module is a severe drawback as a pre-linked module can only be loaded into a system with the exact same physical addresses as the system that was to generate the list of addresses that was used for linking the module.

In the original Contiki system [3] we used pre-linked binary modules for dynamic loading. When compiling the Contiki system core, the compiler generated a map file containing the mapping between all globally visible functions and variables in the system core and their addresses. This list of addresses was used to pre-link Contiki modules.

We quickly noticed that while pre-linked binary modules worked well for small projects with a homogeneous set of sensor nodes, the system quickly became unmanageable when the number of sensor nodes grew. Even a small

change to the system core of one of the sensor nodes would make it impossible to load binary a module into the system bedcase the addresses of variables and functions in the core were different from when the program was linked. We used version numbers to guard against this situation. Version numbers did help against system crashes, but did not solve the general problem: new modules could not be loaded into the system.

5.5.2 Dynamic Linking

With dynamic linking, the object files do not only contain code and data, but also names of functions are variables of the system core that are referenced by the module. The code in the object file cannot be executed before the physical addresses of the referenced variables and functions have been filled in. This process is done at run time by a dynamic linker.

In the Contiki dynamic linker we use two file formats for the dynamically linked modules, ELF and Compact ELF.

ELF - Executable and Linkable Format

One of the most common object code format for dynamic linking is the Executable and Linkable Format (ELF) [34]. It is a standard format for object files and executables that is used for most modern Unix-like systems. An ELF object file include both program code and data and additional information such as a symbol table, the names of all external unresolved symbols, and relocation tables. The relocation tables are used to locate the program code and data at other places in memory than for which the object code originally was assembled. Additionally, ELF files can hold debugging information such as the line numbers corresponding to specific machine code instructions, and file names of the source files used when producing the ELF object.

ELF is also the default object file format produced by the GCC utilities and for this reason there are a number of standard software utilities for manipulating ELF files available. Examples include debuggers, linkers, converters, and programs for calculating program code and data memory sizes. These utilities exist for a wide variety of platforms, including MS Windows, Linux, Solaris, and FreeBSD. This is a clear advantage over other solutions such as FlexCup [25], which require specialized utilities and tools.

Our dynamic linker in Contiki understands the ELF format and is able to perform dynamic linking, relocation, and loading of ELF object code files. The debugging features of the ELF format are not used.

CELF - Compact ELF

One problem with the ELF format is the overhead in terms of bytes to be transmitted across the network, compared to pre-linked modules. There are

a number of reasons for the extra overhead. First, ELF, as any dynamically relocatable file format, includes the symbolic names of all referenced functions or variables that need to be linked at run-time. Second, and more important, the ELF format is designed to work on 32-bit and 64-bit architectures. This causes all ELF data structures to be defined with 32-bit data types. For 8-bit or 16-bit targets the high 16 bits of these fields are unused.

To quantify the overhead of the ELF format we devise an alternative to the ELF object code format that we call CELF - Compact ELF. A CELF file contains the same information as an ELF file, but represented with 8 and 16-bit datatypes. CELF files typically are half the size of the corresponding ELF file. The Contiki dynamic loader is able to load CELF files and a utility program is used to convert ELF files to CELF files.

It is possible to further compress CELF files using lossless data compression. However, we leave the investigation of the energy-efficiency of this approach to future work.

The drawback of the CELF format is that it requires a special compressor utility is for creating the CELF files. This makes the CELF format less attractive for use in many real-world situations.

5.5.3 Position Independent Code

To avoid performing the relocation step when loading a module, it is in some cases possible to compile the module into position independent code. Position independent code is a type of machine code which does not contain any absolute addresses to itself, but only relative references. This is the approach taken by the SOS system.

To generate position independent code compiler support is needed. Furthermore, not all CPU architectures support position independent code and even when supported, programs compiled to position independent code typically are subject to size restrictions. For example, the AVR microcontroller supports position independent code but restricts the size of programs to 4 kilobytes. For the MSP430 no compiler is known to fully support position independent code.

5.6 Implementation

We have implemented run-time dynamic linking of ELF and CELF files in the Contiki operating system [3]. To evaluate dynamic linking we have implemented an application specific virtual machine for Contiki together with a compiler for a subset of Java, and have ported a Java virtual machine to Contiki.

5.6.1 The Contiki Operating System

The Contiki operating system was the first operating system for memory-constrained sensor nodes to support dynamic run-time loading of native code modules. Contiki is built around an event-driven kernel and has very low memory requirements. Contiki applications run as extremely lightweight protothreads [4] that provide blocking operations on top of the event-driven kernel at a very small memory cost. Contiki is designed to be highly portable and has been ported to over ten different platforms with different CPU architectures and using different C compilers.

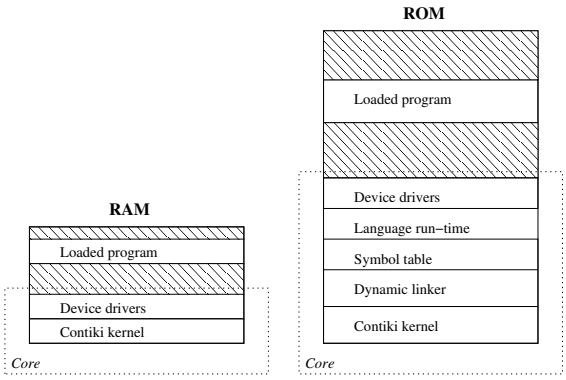


Figure 5.2: Partitioning in Contiki: the core and loadable programs in RAM and ROM.

A Contiki system is divided into two parts: the core and the loadable programs as shown in Figure 5.2. The core consists of the Contiki kernel, device drivers, a set of standard applications, parts of the C language library, and a symbol table. Loadable programs are loaded on top of the core and do not modify the core.

The core has no information about the loadable programs, except for information that the loadable programs explicitly register with the core. Loadable programs, on the other hand, have full knowledge of the core and may freely call functions and access variables that reside in the core. Loadable programs can call each other by going through the kernel. The kernel dispatches calls from one loaded program to another by looking up the target program in an in-kernel list of active processes. This one-way dependency makes it possible to load and unload programs at run-time without needing to patch the core and without the need for a reboot when a module has been loaded or unloaded.

While it is possible to replace the core at run-time by running a special loadable program that overwrites the current core and reboots the system, experience has shown that this feature is not often used in practice.

5.6.2 The Symbol Table

The Contiki core contains a table of the symbolic names of all externally visible variable and function names in the Contiki core and their corresponding addresses. The table includes not only the Contiki system, but also the C language run-time library. The symbol table is used by the dynamic linker when linking loaded programs.

The symbol table is created when the Contiki core binary image is compiled. Since the core must contain a correct symbol table, and a correct symbol table cannot be created before the core exists, a three-step process is required to compile a core with a correct symbol table. First, an intermediary core image with an empty symbol table is compiled. From the intermediary core image an intermediary symbol table is created. The intermediary symbol table contains the correct symbols of the final core image, but the addresses of the symbols are incorrect. Second, a second intermediary core image that includes the intermediary symbol table is created. This core image now contains a symbol table of the same size as the one in the final core image so the addresses of all symbols in the core are now as they will be in the final core image. The final symbol table is then created from the second intermediary core image. This symbol table contains both the correct symbols and their correct addresses. Third, the final core image with the correct symbol table is compiled.

The process of creating a core image is automated through a simple `make` script. The symbol table is created using a combination of standard ELF tools.

For a typical Contiki system the symbol table contains around 300 entries which amounts to approximately 4 kilobytes of data stored in flash ROM.

5.6.3 The Dynamic Linker

We implemented a dynamic linker for Contiki that is designed to link, relocate, and load either standard ELF files [34] and CELF, Compact ELF, files. The dynamic linker reads ELF/CELF files through the Contiki virtual filesystem interface, CFS, which makes the dynamic linker unaware of the physical location of the ELF/CELF file. Thus the linker can operate on files stored either in RAM, on-chip flash ROM, external EEPROM, or external ROM without modification. Since all file access to the ELF/CELF file is made through the CFS, the dynamic linker does not need to concern itself with low-level filesystem details such as wear-leveling or fragmentation [2] as this is better handled by the CFS.

The dynamic linker performs four steps to link, relocate and load an ELF/CELF file. The dynamic linker first parses the ELF/CELF file and extracts relevant information about where in the ELF/CELF file the code, data, symbol table, and relocation entries are stored. Second, memory for

the code and data is allocated from flash ROM and RAM, respectively. Third, the code and data segments are linked and relocated to their respective memory locations, and fourth, the code is written to flash ROM and the data to RAM.

Currently, memory allocation for the loaded program is done using a simple block allocation scheme. More sophisticated allocation schemes will be investigated in the future.

Linking and Relocating

The relocation information in an ELF/CELF file consists of a list of relocation entries. Each relocation entry corresponds to an instruction or address in the code or data in the module that needs to be updated with a new address. A relocation entry contains a pointer to a symbol, such as a variable name or a function name, a pointer to a place in the code or data contained in the ELF/CELF file that needs to be updated with the address of the symbol, and a relocation type which specifies how the data or code should be updated. The relocation types are different depending on the CPU architecture. For the MSP430 there is only one single relocation type, whereas the AVR has 19 different relocation types.

The dynamic linker processes a relocation entry at a time. For each relocation entry, its symbol is looked up in the symbol table in the core. If the symbol is found in the core's symbol table, the address of the symbol is used to patch the code or data to which the relocation entry points. The code or data is patched in different ways depending on the relocation type and on the CPU architecture.

If the symbol in the relocation entry was not found in the symbol table of the core, the symbol table of the ELF/CELF file itself is searched. If the symbol is found, the address that the symbol will have when the program has been loaded is calculated, and the code or data is patched in the same way as if the symbol was found in the core symbol table.

Relocation entries may also be relative to the data, BSS, or code segment in the ELF/CELF file. In that case no symbol is associated with the relocation entry. For such entries the dynamic linker calculates the address that the segment will have when the program has been loaded, and uses that address to patch the code or data.

Loading

When the linking and relocating is completed, the text and data have been relocated to their final memory position. The text segment is then written to flash ROM, at the location that was previously allocated. The memory allocated for the data and BSS segments are used as an intermediate storage for transferring text segment data from the ELF/CELF file before it is

written to flash ROM. Finally, the memory allocated for the BSS segment is cleared, and the contents of the data segment is copied from the ELF/CELF file.

Executing the Loaded Program

When the dynamic linker has successfully loaded the code and data segments, Contiki starts executing the program.

The loaded program may replace an already running Contiki service. If the service that is to be replaced needs to pass state to the newly loaded service, Contiki supports the allocation of an external memory buffer for this purpose. However, experience has shown that this mechanism has been very scarcely used in practice and the mechanism is likely to be removed in future versions of Contiki.

Portability

Since the ELF/CELF format is the same across different platforms, we designed the Contiki dynamic linker to be easily portable to new platforms. The loader is split into one architecture specific part and one generic part. The generic part parses the ELF/CELF file, finds the relevant sections of the file, looks up symbols from the symbol table, and performs the generic relocation logic. The architecture specific part does only three things: allocates ROM and RAM, writes the linked and relocated binary to flash ROM, and understands the relocation types in order to modify machine code instructions that need adjustment because of relocation.

Alternative Designs

The Contiki core symbol table contains all externally visible symbols in the Contiki core. Many of the symbols may never need to be accessed by loadable programs, thus causing ROM overhead. An alternative design would be to let the symbol table include only a handful of symbols, entry points, that define the only ways for an application program to interact with the core. This would lead to a smaller symbol table, but would also require a detailed specification of which entry points that should be included in the symbol table. The main reason why we did not chose this design, however, is that we wish to be able to replace modules at any level of the system. For this reason, we chose to provide the same amount of symbols to an application program as it would have, would it have been compiled directly into the core. However, we are continuing to investigate this alternative design for future versions of the system.

5.6.4 The Java Virtual Machine

We ported the Java virtual machine (JVM) from lejOS [6], a small operating system originally developed for the Lego Mindstorms. The Lego Mindstorms are equipped with an Hitachi H8 microcontroller with 32 kilobytes of RAM available for user programs such as the JVM. The lejOS JVM works within this constrained memory while featuring preemptive threads, recursion, synchronization and exceptions. The Contiki port required changes to the RAM-only model of the lejOS JVM. To be able to run Java programs within the 2 kilobytes of RAM available on our hardware platform, Java classes needs to be stored in flash ROM rather than in RAM. The Contiki port stores the class descriptions including bytecode in flash ROM memory. Static class data and class flags that denote if classes have been initialized are stored in RAM as well as object instances and execution stacks. The RAM requirements for the Java part of typical sensor applications are a few hundred bytes.

Java programs can call native code methods by declaring native Java methods. The Java virtual machine dispatches calls to native methods to native code. Any native function in Contiki may be called, including services that are part of a loaded Contiki program.

5.6.5 CVM - the Contiki Virtual Machine

We designed the Contiki Virtual Machine, CVS, to be a compromise between an application-specific and a generic virtual machine. CVM can be configured for the application running on top of the machine by allowing functions to be either implemented as native code or as CVM code. To be able to run the same programs for the Java VM and for CVM, we developed a compiler that compiles a subset of the Java language to CVM bytecode.

The design of CVM is intentionally similar to other virtual machines, including Maté [17], VM* [16], and the Java virtual machine. CVM is a stack-based machine with separated code and data areas. The CVM instruction set contains integer arithmetic, unconditional and conditional branches, and method invocation instructions. Method invocation can be done in two ways, either by invocation of CVM bytecode functions, or by invocation of functions implemented in native code. Invocation of native functions is done through a special instruction for calling native code. This instruction takes one parameter, which identifies the native function that is to be called. The native function identifiers are defined at compile time by the user that compiles a list of native functions that the CVM program should be able to call. With the native function interface, it is possible for a CVM program to call any native functions provided by the underlying system, including services provided by loadable programs.

Native functions in a CVM program are invoked like any other function.

The CVM compiler uses the list of native functions to translate calls to such functions into the special instruction for calling native code. Parameters are passed to native functions through the CVM stack.

5.7 Evaluation

To evaluate dynamic linking of native code we compare the energy costs of transferring, linking, relocating, loading, and executing a native code module in ELF format using dynamic linking with the energy costs of transferring, loading, and executing the same program compiled for the CVM and the Java virtual machine. We devise a simple model of the energy consumption of the reprogramming process. Thereafter we experimentally quantify the energy and memory consumption as well as the execution overhead for the reprogramming, the execution methods and the applications. We use the results of the measurements as input into the model which enables us to perform a quantitative comparison of the energy-efficiency of the reprogramming methods.

We use the ESB board [31] and the Telos Sky board [27] as our experimental platforms. The ESB is equipped with an MSP430 microcontroller with 2 kilobytes of RAM and 60 kilobytes of flash ROM, an external 64 kilobyte EEPROM, as well as a set of sensors and a TR1001 radio transceiver. The Telos Sky is equipped with an MSP430 microcontroller with 10 kilobytes of RAM and 48 kilobytes of flash ROM together with a CC2420 radio transceiver. We use the ESB to measure the energy of receiving, storing, linking, relocating, loading and executing loadable modules and the Telos Sky to measure the energy of receiving loadable modules.

We use three Contiki programs to measure the energy efficiency and execution overhead of our different approaches. Blinker, the first of the two programs, is shown in Figure 5.3. It is a simple program that toggles the LEDs every second. The second program, Object Tracker, is an object tracking application based on abstract regions [35]. To allow running the programs both as native code, as CVM code, and as Java code we have implemented these programs both in C and Java. A schematic illustration of the C implementation is in Figure 5.4. To support the object tracker program, we implemented a subset of the abstract regions mechanism in Contiki. The Java and CVM versions of the program call native code versions of the abstract regions functions. The third program is a simple 8 by 8 vector convolution calculation.

5.7.1 Energy Consumption

We model the energy consumption E of the reprogramming process with

$$E = E_p + E_s + E_l + E_f$$

```
PROCESS_THREAD(test_blink, ev, data)
{
  static struct etimer t;
  PROCESS_BEGIN();

  etimer_set(&t, CLOCK_SECOND);

  while(1) {
    leds_on(LEDS_GREEN);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));
    etimer_reset(&t);

    leds_off(LEDS_GREEN);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));
    etimer_reset(&t);
  }

  PROCESS_END();
}
```

Figure 5.3: Example Contiki program that toggles the LEDs every second.

where E_p is the energy spent in transferring the object over the network, E_s the energy cost of storing the object on the device, E_l the energy consumed by linking and relocating the object, and E_f the required energy for of storing the linked program in flash ROM. We use a simplified model of the network propagation energy where we assume a propagation protocol where the energy consumption E_p is proportional to the size of the object to be transferred. Formally,

$$E_p = P_p s_o$$

where s_o is the size of the object file to be transfered and P_p is a constant scale factor that depends on the network protocol used to transfer the object. We use similar equations for E_s (energy for storing the binary) and E_l (energy for linking and relocating). The equation for E_f (the energy for loading the binary to ROM) contains the size of the compiled code size of the program instead of the size of the object file. This model is intentionally simple and we consider it good enough for our purpose of comparing the energy-efficiency of different reprogramming schemes.

Lower Bounds on Radio Reception Energy

We measured the energy consumption of receiving data over the radio for two different radio transceivers: the TR1001 [30], that is used on the ESB board, and the CC2420 [33], that conforms to the IEEE 802.15.4 standard [9] and is used on the Telos Sky board. The TR1001 provides a very low-level interface to the radio medium. The transceiver decodes data at the bit level and transmits the bits in real-time to the CPU. Start bit detection, framing, MAC layer, checksums, and all protocol processing must be done in software

```
PROCESS_THREAD(use_regions_process, ev, data)
{
  PROCESS_BEGIN();

  while(1) {

    value = pir_sensor.value();

    region_put(reading_key, value);
    region_put(reg_x_key, value * loc_x());
    region_put(reg_y_key, value * loc_y());
    if(value > threshold) {
      max = region_max(reading_key);

      if(max == value) {
        sum = region_sum(reading_key);
        sum_x = region_sum(reg_x_key);
        sum_y = region_sum(reg_y_key);
        centroid_x = sum_x / sum;
        centroid_y = sum_y / sum;
        send(centroid_x, centroid_y);
      }
    }

    etimer_set(&t, PERIODIC_DELAY);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));

  }

  PROCESS_END();
}
```

Figure 5.4: Schematic implementation of an object tracker based on abstract regions.

running on the CPU. In contrast, the interface provided by the CC2420 is at a higher level. Start bits, framing, and parts of the MAC protocol are handled by the transceiver. The software driver handles incoming and outgoing data on the packet level.

Since the TR1001 operates at the bit-level, the communication speed of the TR1001 is determined by the CPU. We use a data rate of 9600 bits per second. The CC2420 has a data rate of 250 kilobits per second, but also incurs some protocol overhead as it provides a more high-level interface.

Figure 5.5 shows the current draw from receiving 1000 bytes of data with the TR1001 and CC2420 radio transceivers. These measurements constitute a lower bound on the energy consumption for receiving data over the radio, as they do not include any control overhead caused by a code propagation protocol. Nor do they include any packet headers. An actual propagation protocol would incur overhead because of both packet headers and control traffic. For example, the Deluge protocol has a control packet overhead of approximately 20% [12]. This overhead is derived from the total number of

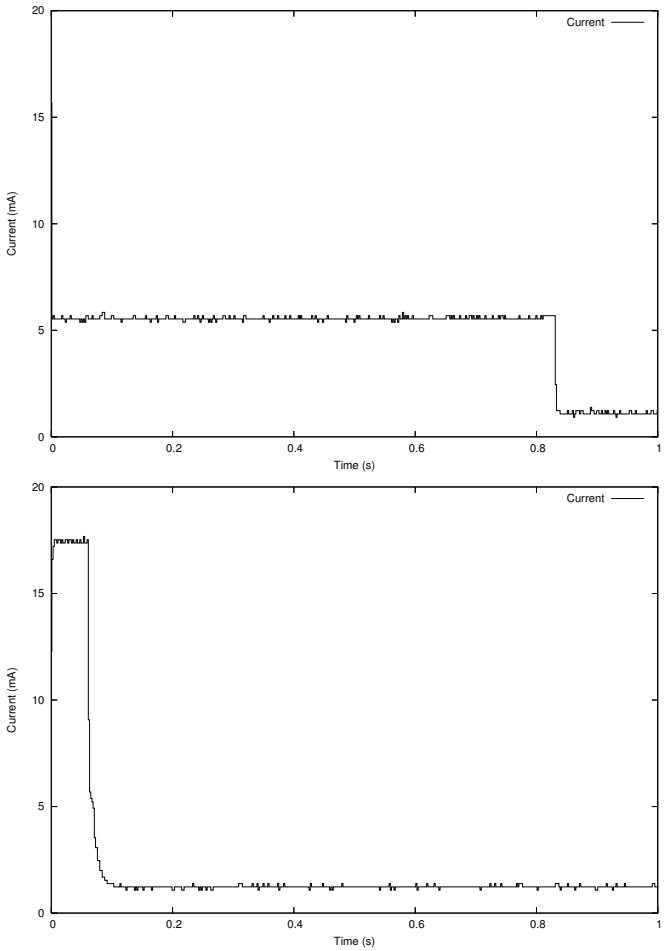


Figure 5.5: Current draw for receiving 1000 bytes with the TR1001 and CC2420, respectively.

control packets and the total number of data packets in a sensor network. The average overhead in terms of number of excessive data packets received is 3.35 [12]. In addition to the actual code propagation protocol overhead, there is also overhead from the MAC layer, both in terms of packet headers and control traffic.

The TR1001 provides a low-level interface to the CPU, which enabled us to measure only the current draw of the receiver. We first measured the time required for receiving one byte of data from the radio. To produce the graph in the figure, we measured the current draw of an ESB board which we had programmed to turn on receive mode and busy-wait for the time corresponding to the reception time of 1000 bytes.

When measuring the reception current draw of the CC2420, we could

Transceiver	Time (s)	Energy (mJ)	Time per byte (s)	Energy per byte (mJ)
TR1001	0.83	21	0.0008	0.021
CC2420	0.060	4.8	0.00006	0.0048

Table 5.2: Lower bounds on the time and energy consumption for receiving 1000 bytes with the TR1001 and CC2420 transceivers. All values are rounded to two significant digits.

not measure the time required for receiving one byte because the CC2420 does not provide an interface at the bit level. Instead, we used two Telos Sky boards and programmed one to continuously send back-to-back packets with 100 bytes of data. We programmed the other board to turn on receive mode when the on-board button was pressed. The receiver would receive 1000 bytes of data, corresponding to 10 packets, before turning the receiver off. We placed the two boards next to each other on a table to avoid packet drops. We produced the graph in Figure 5.5 by measuring the current draw of the receiver Telos Sky board. To ensure that we did not get spurious packet drops, we repeated the measurement five times without obtaining differing results.

Table 5.2 shows the lower bounds on the time and energy consumption for receiving data with the TR1001 and CC2420 transceivers. The results show that while the current draw of the CC2420 is higher than that of the TR1001, the energy efficiency in terms of energy per byte of the CC2420 is better because of the shorter time required to receive the data.

Energy Consumption of Dynamic Linking

To evaluate the energy consumption of dynamic linking, we measure the energy required for the Contiki dynamic linker to link and load two Contiki programs. Normally, Contiki loads programs from the radio network but to avoid measuring any unrelated radio or network effects, we stored the loadable object files in flash ROM before running the experiments. The loadable objects were stored as ELF files from which all debugging information and symbols that were not needed for run-time linking was removed. At boot-up, one ELF file was copied into an on-board EEPROM from where the Contiki dynamic linker linked and relocated the ELF file before it loaded the program into flash ROM.

Figure 5.6 shows the current draw when loading the Blinker program, and Figure 5.7 shows the current draw when loading the Object Tracker program. The current spikes seen in both graphs are intentionally caused by blinking the on-board LEDs. The spikes delimit the four different steps that the loader is going through: copying the ELF object file to EEPROM, linking and relocating the object code, copying the linked code to flash

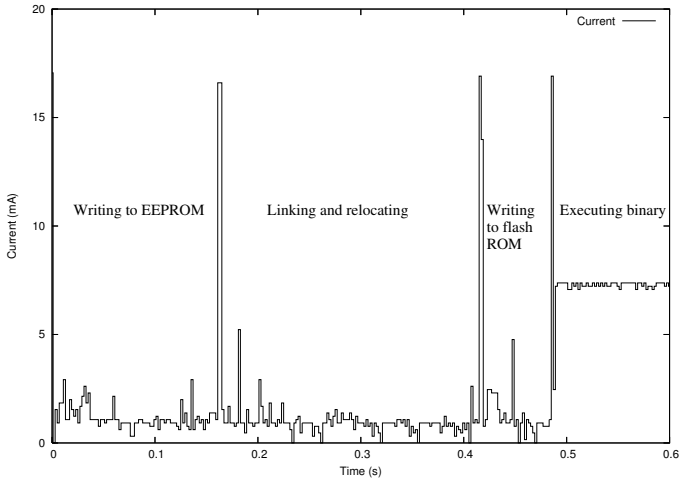


Figure 5.6: Current draw for writing the Blinker ELF file to EEPROM (0 - 0.166 s), linking and relocating the program (0.166 - 0.418 s), writing the resulting code to flash ROM (0.418 - 0.488 s), and executing the binary (0.488 s and onward). The current spikes delimit the three steps and are intentionally caused by blinking on-board LEDs. The high energy consumption when executing the binary is caused by the green LED.

ROM, and finally executing the loaded program. The current draw of the green LED is slightly above 8 mA, which causes the high current draw when executing the blinker program (Figure 5.6). Similarly, when the object tracking application starts, it turns on the radio for neighbor discovery. This causes the current draw to rise to around 6 mA in Figure 5.7, and matches the radio current measurements in Figure 5.5.

Table 5.3 shows the energy consumption of loading and linking the Blinker program. The energy was obtained from integration of the curve from Figure 5.6 and multiplying it by the voltage used in our experiments (4.5 V). We see that the linking and relocation step is the most expensive in terms of energy. It is also the longest step.

To evaluate the energy overhead of the ELF file format, we compare the energy consumption for receiving four different Contiki programs using the ELF and CELF formats. In addition to the two programs from Figures 5.3 and 5.4 we include the code for the Contiki code propagation mechanism and a network publish/subscribe program that performs periodic flooding and converging of information. The two latter programs are significantly larger. We calculate an estimate of the required energy for receiving the files by using the measured energy consumption of the CC2420 radio transceiver and multiply it by the average overhead by the Deluge code propagation protocol, 3.35 [12]. The results are listed in Table 5.4 and show that radio

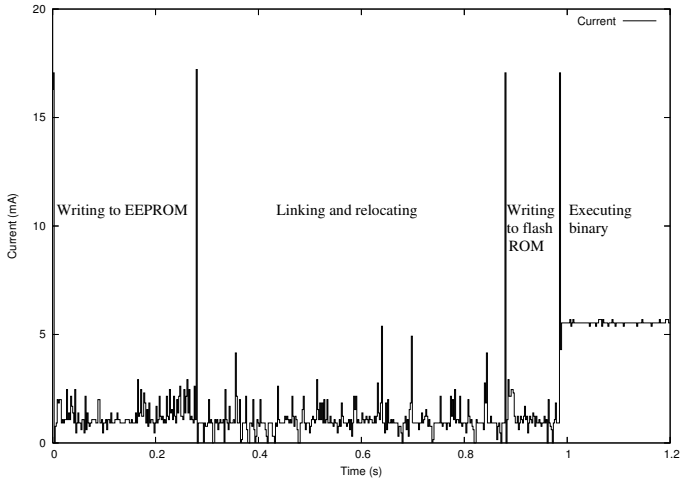


Figure 5.7: Current draw for writing the Object Tracker ELF file to EEPROM (0 - 0.282 s), linking and relocating the program (0.282 - 0.882 s), writing the resulting code to flash ROM (0.882 - 0.988 s), and executing the binary (0.988 s and onward). The current spikes delimit the three steps and are intentionally caused by blinking on-board LEDs. The high current draw when executing the binary comes from the radio being turned on.

reception is more energy consuming than linking and loading a program, even for a small program. Furthermore, the results show that the relative average size and energy overhead for ELF files compared to the code and data contained in the files is approximately 4 whereas the relative CELF overhead is just under 2.

5.7.2 Memory Consumption

Memory consumption is an important metric for sensor nodes since memory is a scarce resource on most sensor node platforms. The ESB nodes feature only 2 KB RAM and 60 KB ROM while Mica2 motes provide 128 KB of program memory and 4 KB of RAM. The less memory required for reprogramming, the more is left for applications and support for other important tasks such as security which may also require a large part of the available memory [26].

Table 5.5 lists the memory requirements of the static linker, the dynamic linker and loader, the CVM and the Java VM. The dynamic linker needs to keep a table of all core symbols in the system. For a complete Contiki system with process management, networking, the dynamic loader, memory allocation, Contiki libraries, and parts of the standard C library, the symbol table requires about 4 kilobytes of ROM. This is included in the ROM size

Step	Blinker time (s)	Energy (mJ)	Obj. Tr. time (s)	Energy (mJ)
Wrt. EEPROM	0.164	1.1	0.282	1.9
Link & reloc	0.252	1.2	0.600	2.9
Wrt. flash ROM	0.070	0.62	0.106	0.76
Total	0.486	2.9	0.988	5.5

Table 5.3: Measured energy consumption of the storing, linking and loading of the 1056 bytes large Blinker binary and the 1824 bytes large Object Tracker binary. The size of the Blinker code is 130 bytes and the size of the Object Tracker code is 344 bytes.

Program	Code size	Data size	ELF file size	ELF file size overhead	ELF radio reception energy (mJ)	CELF file size	CELF file size overhead	CELF radio reception energy (mJ)
Blinker	130	14	1056	7.3	17	361	2.5	5.9
Object tracker	344	22	1668	5.0	29	758	2.0	12
Code propagator	2184	10	5696	2.6	92	3686	1.7	59
Flood/converge	4298	42	8456	1.9	136	5399	1.2	87

Table 5.4: The overhead of the ELF and CELF file formats in terms of bytes and estimated reception energy for four Contiki programs. The reception energy is the lower bound of the radio reception energy with the CC2420 chip, multiplied by the average Deluge overhead (3.35).

for the dynamic linker.

5.7.3 Execution Overhead

To measure the execution overhead of the application specific virtual machine and the Java virtual machine, we implemented the object tracking program in Figure 5.4 in C and Java. We compiled the Java code to CVM code and Java bytecode. We ran the compiled code on the MSP430-equipped ESB board. The native C code was compiled with the MSP430 port of GCC version 3.2.3. The MSP430 digitally-controlled oscillator was set to clock

Module	ROM	RAM
Static loader	670	0
Dynamic linker, loader	5694	18
CVM	1344	8
Java VM	13284	59

Table 5.5: Memory requirements, in bytes. The ROM size for the dynamic linker includes the symbol table. The RAM figures do not include memory for programs running on top of the virtual machines.

Execution type	Execution time (ms)	Energy (mJ)
Native	0.479	0.00054
CVM	0.845	0.00095
Java VM	1.79	0.0020

Table 5.6: Execution times and energy consumption of one iteration of the tracking program.

Execution type	Execution time (ms)	Energy (mJ)
Native	0.67	0.00075
CVM	58.52	0.065
Java VM	65.6	0.073

Table 5.7: Execution times and energy consumption of the 8 by 8 vector convolution.

the CPU at a speed of 2.4576 MHz. We measured the execution time of the three implementations using the on-chip timer A1 that was set to generate a timer interrupt 1000 times per second. The execution times are averaged over 5000 iterations of the object tracking program.

The results in Table 5.6 show the execution time of one run of the object tracking application from Figure 5.4. The execution time measurements are averaged over 5000 runs of the object tracking program. The energy consumption is calculated by multiplying the execution time with the average energy consumption when a program is running with the radio turned off. The table shows that the overhead of the Java virtual machine is higher than that of the CVM, which is turn is higher than the execution overhead of the native C code.

All three implementations of the tracker program use the same abstract regions library which is compiled as native code. Thus much of the execution time in the Java VM and CVM implementations of the object tracking program is spent executing the native code in the abstract regions library. Essentially, the virtual machine simply acts as a dispatcher of calls to various native functions. For programs that spend a significant part of their time executing virtual machine code the relative execution times are significantly higher for the virtual machine programs. To illustrate this, Table 5.7 lists the execution times of a convolution operation of two vectors of length 8. Convolution is a common operation in digital signal processing where it is used for algorithms such as filtering or edge detection. We see that the execution time of the program running on the virtual machines is close to ten times that of the native program.

Step	Dynamic linking (mJ)	Full image replacement (mJ)
Receiving	17	330
Wrt. EEPROM	1.1	22
Link & reloc	1.4	-
Wrt. flash ROM	0.45	72
Total	20	424

Table 5.8: Comparison of energy-consumption of reprogramming the blinker application using dynamic linking with an ELF file and full image replacement methods.

5.7.4 Quantitative Comparison

Using our model from Section 5.7.1 and the results from the above measurements, we can calculate approximations of the energy consumption for distribution, reprogramming, and execution of native and virtual machine programs in order to compare the methods with each other. We set P_p, the scale factor of the energy consumption for receiving an object file, to the average Deluge overhead of 3.35.

Dynamic Linking vs Full Image Replacement

We first compare the energy costs for the two native code reprogramming models: dynamic linking and full image replacement. Table 5.8 shows the results for the energy consumption of reprogramming the blinker application. The size of blinker application including the operating system is 20 KB which is about 20 times the size of the blinker application itself. Even though no linking needs to be performed during the full image replacement, this method is about 20 times more expensive to perform a whole image replacement compared to a modular update using the dynamic linker.

Dynamic Linking vs Virtual Machines

We use the tracking application to compare reprogramming using the Contiki dynamic linker with code updates for the CVM and the Java virtual machine. CVM programs are typically very small and are not stored in EEPROM, nor are they linked or written to flash. Java uncompressed class files are loaded into flash ROM before they are executed. Table 5.9 shows the sizes of the corresponding binaries and the energy consumption of each reprogramming step.

As expected, the process of updating sensor nodes with native code is less energy-efficient than updating with a virtual machine. Also, as shown in Table 5.6, executing native code is more energy-efficient than executing code for the virtual machines.

Step	ELF	CELF	CVM	Java
Size (bytes)	1824	968	123	1356
Receiving	29	12	2.0	22
Wrt. EEPROM	1.9	0.80	-	-
Link & reloc	2.5	2.5	-	-
Wrt. flash ROM	1.2	1.2	-	4.7
Total	35	16.5	2.0	26.7

Table 5.9: Comparison of energy-consumption in mJ of reprogramming for the object tracking application using the four different methods.

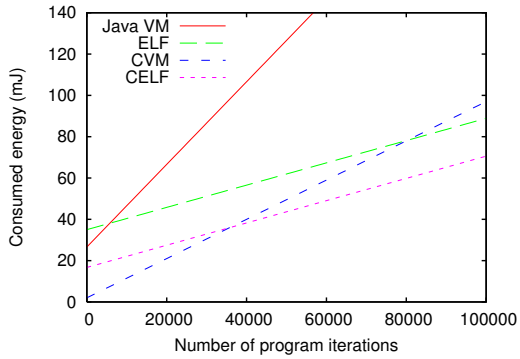


Figure 5.8: Break-even points for the object tracking program implemented with four different linking and execution methods.

By combining the results in Table 5.6 and Table 5.9, we can compute break-even points for how often we can execute native code as opposed to virtual machine code for the same energy consumption. That is, after how many program iterations do the cheaper execution costs outweigh the more expensive code updates.

Figure 5.8 shows the modeled energy consumption for executing the Object Tracking program using native code loaded with an ELF object file, native code loaded with an CELF object file, CVM code, and Java code. We see that the Java virtual machine is expensive in terms of energy and will always require more energy than native code loaded with a CELF file. For native code loaded with an ELF file the energy overhead due to receiving the file makes the Java virtual machine more energy efficient until the program is repeated a few thousand times. Due to the small size of the CVM code it is very energy efficient for small numbers of program iterations. It takes about 40000 iterations of the program before the interpretation overhead outweigh the linking and loading overhead of same program running as native code and loaded as a CELF file. If the native program was loaded with an ELF file, however, the CVM program needs to be run approximately

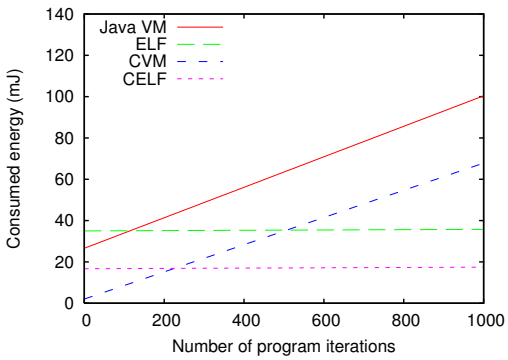


Figure 5.9: Break-even points for the vector convolution implemented with four different linking and execution methods.

80000 iterations before the energy costs are the same. At the break-even point, the energy consumption is only about one fifth of the energy consumption for loading the blink program using full image replacement as shown in Table 5.8.

In contrast with Figure 5.8, Figure 5.9 contains the break-even points from the vector convolution in Table 5.7. We assume that the convolution algorithm is part of a program with the same size as in Figure 5.8 so that the energy consumption for reprogramming is the same. In this case the break-even points are drastically lower than in Figure 5.8. Here the native code loaded with an ELF file outperforms the Java implementation already at 100 iterations. The CVM implementation has spent as much energy as the native ELF implementation after 500 iterations.

5.7.5 Scenario Suitability

We can now apply our results to the software update scenarios discussed in Section 5.3. In a scenario with frequent code updates, such as the dynamic application scenario or during software development, a low loading overhead is to prefer. From Figure 5.8 we see that both an application-specific virtual machine and a Java machine may be good choices. Depending on the type of application it may be beneficial to decide to run the program on top of a more flexible virtual machine such as the Java machine. The price for such a decision is higher energy overhead.

In scenarios where the update frequency is low, e.g. when fixing bugs in installed software or when reconfiguring an installed application, the higher price for dynamic linking may be worth paying. If the program is continuously run for a long time, the energy savings of being able to use native code outweigh the energy cost of the linking process. Furthermore, with a virtual machine it may not be possible to make changes to all levels of the

Module	Lines of code, total	Lines of code, relocation function
Generic linker	292	
MSP430-specific	45	8
AVR-specific	143	104

Table 5.10: Number of lines of code for the dynamic linker and the microcontroller-specific parts.

system. For example, a bug in a low-level driver can usually only be fixed by installing new native code. Moreover, programs that are computationally heavy benefit from being implemented as native code as native code has lower energy consumption than virtual machine code.

The results from Figures 5.8 and 5.9 suggest that a combination of virtual machin code and native code can be energy efficient. For many situations this may be a viable alternative to running only native code or only virtual machine code.

5.7.6 Portability

Because of the diversity of sensor network platforms, the Contiki dynamic linker is designed to be portable between different microcontrollers. The dynamic linker is divided into two modules: a generic part that parses and analyzes the ELF/CELF that is to be loaded, and a microcontroller-specific part that allocates memory for the program to be loaded, performs code and data relocation, and writes the linked program into memory.

To evaluate the portability of our design we have ported the dynamic linker to two different microcontrollers: the TI MSP430 and the Atmel AVR. The TI MSP430 is used in several sensor network platforms, including the Telos Sky and the ESB. The Atmel AVR is used in the Mica2 motes.

Table 5.10 shows the number of lines of code needed to implement each module. The dramatic difference between the MSP430-specific module and the AVR-specific module is due to the different addressing modes used by the machine code of the two microcontrollers. While the MSP430 has only one addressing mode, the AVR has 19 different addressing modes. Each addressing mode must be handled differently by the relocation function, which leads to a larger amount of code for the AVR-specific module.

5.8 Discussion

Standard file formats. Our main motivation behind choosing the ELF format for dynamic linking in Contiki was that the ELF format is a standard file format. Many compilers and utilities, including all GCC utilities, are able to produce and handle ELF files. Hence no special software is needed

to compile and upload new programs into a network of Contiki nodes. In contrast, FlexCup [25] or diff-based approaches require the usage of specially crafted utilities to produce meta data or diff scripts required for uploading software. These special utilities also need to be maintained and ported to the full range of development platforms used for software development for the system.

Operating system support. Dynamic linking of ELF files requires support from the underlying operating system and cannot be done on monolithic operating systems such as TinyOS. This is a disadvantage of our approach. For monolithic operating systems, an approach such as FlexCup is better suited.

Heterogeneity. With diff-based approaches a binary diff is created either at a base station or by an outside server. The server must have knowledge of the exact software configuration of the sensor nodes on which the diff script is to be run. If sensor nodes are running different versions of their software, diff-based approaches do not scale.

Specifically, in many of our development networks we have witnessed a form of *micro heterogeneity* in the software configuration. Many sensor nodes, which have been running the exact same version of the Contiki operating system, have had small differences in the address of functions and variables in the core. This micro heterogeneity comes from the different core images being compiled by different developers, each having slightly different versions of the C compiler, the C library and the linker utilities. This results in small variations of the operating system image depending on which developer compiled the operating system image. With diff-based approaches micro heterogeneity poses a big problem, as the base station would have to be aware of all the small differences between each node.

Combination of native and virtual machine code. Our results suggest that a combination of native and virtual machine code is an energy efficient alternative to pure native code or pure virtual machine code approaches. The dynamic linking mechanism can be used to load the native code that is used by the virtual machine code by the native code interfaces in the virtual machines.

5.9 Related Work

Because of the importance of dynamic reprogramming of wireless sensor networks there has been a lot of effort in the area of software updates for sensor nodes both in the form of system support for software updates and execution environments that directly impact the type and size of updates as well as distribution protocols for software updates.

Mainwaring et al. [24] also identified the trade-off between using virtual machine code that is more expensive to run but enables more energy-efficient

updates and running native code that executes more efficiently but requires more costly updates. This trade-off has been further discussed by Levis and Culler [17] who implemented the Maté virtual machine designed to both simplify programming and to leverage energy-efficient large-scale software updates in sensor networks. Maté is implemented on top of TinyOS.

Levis and Culler later enhanced Maté by application specific virtual machines (ASVMs) [18]. They address the main limitations of Maté: flexibility, concurrency and propagation. Whereas Maté was designed for a single application domain only, ASVM supports a wide range of application domains. Further, instead of relying on broadcasts for code propagation as Maté, ASVM uses the trickle algorithm [19].

The MagnetOS [21] system uses the Java virtual machine to distribute applications across an ad hoc network of laptops. In MagnetOS, Java applications are partitioned into distributed components. The components transparently communicate by raising events. Unlike Maté and Contiki, MagnetOS targets larger platforms than sensor nodes such as PocketPC devices. SensorWare [1] is another script-based proposal for programming nodes that targets larger platforms. VM* is a framework for runtime environments for sensor networks [16]. Using this framework Koshy and Pandey have implemented a subset of the Java Virtual Machine that enables programmers to write applications in Java, and access sensing devices and I/O through native interfaces.

Mobile agent-based approaches extend the notion of injected scripts by deploying dynamic, localized and intelligent mobile agents. Using mobile agents, Fok et al. have built the Agilla platform that enables continuous reprogramming by injecting new agents into the network [7].

TinyOS uses a special description language for composing a system of smaller components [8] which are statically linked with the kernel to a complete image of the system. After linking, modifying the system is not possible [17] and hence TinyOS requires the whole image to be updated even for small code changes.

Systems that offer loadable modules besides Contiki include SOS [10] and Impala [22]. Impala features an application updater that enables software updates to be performed by linking in updated modules. Updates in Impala are coarse-grained since cross-references between different modules are not possible. Also, the software updater in Impala was only implemented for much more resource-rich hardware than our target devices. The design of SOS [10] is very similar to the Contiki system: SOS consists of a small kernel and dynamically-loaded modules. However, SOS uses position independent code to achieve relocation and jump tables for application programs to access the operating system kernel. Application programs can register function pointers with the operating system for performing interprocess communication. Position independent code is not available for all platforms, however, which limits the applicability of this approach.

FlexCup [25] enables run-time installation of software components in TinyOS and thus solves the problem that a full image replacement is required for reprogramming TinyOS applications. In contrast to our ELF-based solution, FlexCup uses a non-standard format and is less portable. Further, FlexCup requires a reboot after a program has been installed, requiring an external mechanism to save and restore the state of all other applications as well as the state of running network protocols across the reboot. Contiki does not need to be rebooted after a program has been installed.

FlexCup also requires a complete duplicate image of the binary image of the system to be stored in external flash ROM. The copy of the system image is used for constructing a new system image when a new program has been loaded. In contrast, the Contiki dynamic linker does not alter the core image when programs are loaded and therefore no external copy of the core image is needed.

Since the energy consumption of distributing code in sensor networks increases with the size of the code to be distributed several attempts have been made to reduce the size of the code to be distributed. Reijers and Langendoen [29] produce an edit script based on the difference between the modified and original executable. After various optimizations including architecture-dependent ones, the script is distributed. A similar approach has been developed by Jeong and Culler [13] who use the rsync algorithm to generate the difference between modified and original executable. Koshy and Pandey's diff-based approach [15] reduces the amount of flash rewriting by modifying the linking procedure so that functions that are not changed are not shifted.

XNP [14] was the previous default reprogramming mechanism in TinyOS which is used by the multi-hop reprogramming scheme MOAP (Multihop Over-the-Air Programming) developed to distribute node images in the sensor network. MOAP distributes data to a selective number of nodes on a neighbourhood-by-neighbourhood basis that avoids flooding [32]. In Trickle [19] virtual machine code is distributed to a network of nodes. While Trickle is restricted to single packet dissemination, Deluge adds support for the dissemination of large data objects [12].

5.10 Conclusions

We have presented a highly portable dynamic linker and loader that uses the standard ELF file format and compared the energy-efficiency of run-time dynamic linking with an application specific virtual machine and a Java virtual machine. We show that dynamic linking is feasible even for constrained sensor nodes.

Our results also suggest that a combination of native and virtual machine code provide an energy efficient alternative to pure native code or pure

virtual machine approaches. The native code that is called from the virtual machine code can be updated using the dynamic linker, even in heterogeneous systems.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and the European Commission under contract IST-004536-RUNES. Thanks to our paper shepherd Feng Zhao for reading and commenting on the paper.

Bibliography

[1] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, May 2003.

[2] H. Dai, Michael N., and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *2nd international conference on Embedded networked sensor systems (ACM SenSys)*, Baltimore, MD, USA, November 2004.

[3] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[4] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[5] D. Estrin (editor). *Embedded everywhere: A research agenda for networked systems of embedded computers.* National Academy Press, 1st edition, October 2001. ISBN: 0309075688

[6] G. Ferrari, J. Stuber, A. Gombos, and D. Laverde, editors. *Programming Lego Mindstorms with Java with CD-ROM.* Syngress Publishing, 2002. ISBN: 1928994555

[7] C. Fok, G. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, June 2005.

[8] 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*, pages 1–11, 2003.

[9] J. A. Gutierrez, M. Naeve, E. Callaway, M. Bourgeois, V. Mitter, and B. Heile. IEEE 802.15.4: A developing standard for low-power low-cost wireless personal area networks. *IEEE Network*, 15(5):12–19, September/October 2001.

[10] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. SOS: A dynamic operating system for sensor networks. In *MobiSys '05*, 2005.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[12] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. SenSys'04*, Baltimore, Maryland, USA, November 2004.

[13] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *IEEE SECON*, October 2004.

[14] J. Jeong, S. Kim, and A. Broad. Network reprogramming. TinyOS documentation, 2003. Visited 2006-04-06. http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf

[15] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proc. EWSN'05*, 2005.

[16] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *Proc. SenSys'05*, San Diego, CA, USA, November 2005.

[17] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS-X*, San Jose, CA, USA, October 2002.

[18] P. Levis, D. Gay, and D Culler. Active sensor networks. In *Proc. USENIX/ACM NSDI'05*, Boston, MA, USA, May 2005.

[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'04*, March 2004.

[20] J. Lilius and I. Paltor. Deeply embedded python, a virtual machine for embedded systems. Web page. 2006-04-06. http://www.tucs.fi/magazin/output.php?ID=2000.N2.LilDeEmPy

[21] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. Gün Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys*, pages 149–162, 2005.

[22] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proc. Second Intl. Conference on Mobile Systems, Applications and Services (MOBISYS 2004)*, June 2004.

[23] G. Mainland, L. Kang, S. Lahaie, D. C. Parkes, and M. Welsh. Using virtual markets to program global behavior in sensor networks. In *Proceedings of the 2004 SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[24] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, USA, September 2002.

[25] P. Marrón et al. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *European Workshop on Wireless Sensor Networks*, 2006.

[26] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: security protocols for sensor netowrks. In *Mobile Computing and Networking*, pages 189–199, 2001.

[27] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

[28] N. Ramanathan, E. Kohler, and D. Estrin. Towards a debugging system for sensor networks. *International Journal for Network Management*, 3(5), 2005.

[29] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proc. WSNA'05*, 2003.

[30] RF Monolithics. 868.35 MHz Hybrid Transceiver TR1001, 1999. http://www.rfm.com

[31] J. Schiller, H. Ritter, A. Liers, and T. Voigt. Scatterweb - low power nodes and energy aware routing. In *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, USA, 2005.

[32] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, UCLA, Center for Embedded Networked Computing, Nov. 2003.

[33] Texas Instruments. CC2420 Datasheet (rev. 1.4), 2007. http://focus.ti.com/docs/prod/folders/print/cc2420.html

[34] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

[35] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. USENIX/ACM NSDI'04*, San Francisco, CA,, March 2004.

[36] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

Chapter 6

Experiences from Two Sensor Network Deployments — Self-Monitoring and Self-Configuration Keys to Success

Niclas Finne, Joakim Eriksson, Adam Dunkels, Thiemo Voigt.
Swedish Institute of Computer Science, SICS.
{nfi,joakime,adam,thiemo}@sics.se

6.1 Abstract

Despite sensor network protocols being self-configuring, sensor network deployments continue to fail. We report our experience from two recently deployed IP-based multi-hop sensor networks: one in-door surveillance network in a factory complex and a combined out-door and in-door surveillance network. Our experiences highlight that adaptive protocols alone are not sufficient, but that an approach to self-monitoring and self-configuration that covers more aspects than protocol adaptation is needed. Based on our experiences, we design and implement an architecture for self-monitoring of sensor nodes. We show that the self-monitoring architecture detects and prevents the problems with false alarms encountered in our deployments. The architecture also detects software bugs by monitoring actual and expected duty-cycle of key components of the sensor node. We show that the energy-monitoring architecture detects bugs that cause the radio chip to be active longer than expected.

6.2 Introduction

Surveillance is one of the most prominent application domains for wireless sensor networks. Wireless sensor networks enable rapidly deployed surveillance applications in urban terrain. While most wireless sensor network mechanisms are self-configuring and designed to operate in changing conditions [12, 16], the characteristics of the deployment environment often cause additional and unexpected problems [8, 9, 11]. In particular, Langendoen et al. [8] point out the difficulties posed by, e.g., hardware not working as expected.

To contribute to the understanding of the problems encountered in real-world sensor network deployments, we report on our experience from recent deployments of two surveillance applications: one in-door surveillance application in a factory complex, and one combined out-door and in-door surveillance network. Both applications covered a large area and therefore required multi-hop networking.

Our experiences highlight that adaptive protocols alone are not sufficient, but that an approach to self-monitoring and self-configuration that covers more aspects than protocol adaptation is needed. An example where we have experienced the need for self-monitoring of sensor nodes is when the components used in low-cost sensor nodes behave differently on different nodes. In many of our experiments, radio transmissions triggered the motion detector on a subset of our nodes while other nodes did not experience this problem.

Motivated by the observation that self-configuration and adaptation is not sufficient to circumvent unexpected hardware and software problems, we design and implement a self-monitoring architecture for detecting hardware and software problems. Our architecture consists of pairs of probes and activators where the activators start up an activity that is suspected to trigger problems and the probes measure if sensor components react to the activator's activity. Callback functions enable a node to self-configure its handling of a detected problem. We experimentally demonstrate that our approach solves the observed problem of packet transmissions triggering the motion detector.

To find software problems, we integrate Contiki's software-based on-line energy estimator [6] into the self-monitoring architecture. This allows us to detect problems such as the CPU not going into the correct low power mode, a problem previously encountered by Langendoen et al. [8]. With two examples we demonstrate the effectiveness of the self-monitoring architecture. Based on our deployment experiences, we believe this tool to be very valuable for both application developers and system developers.

The rest of the paper is structured as follows. The setup and measurements for the two deployments are described in Section 6.3. In Section 6.4 we present our experiences from the deployments, including unexpected be-

havior. Section 6.5 describes our architecture for self-monitoring while the following section evaluates it. Finally, we describe related work in Section 6.7 and our conclusions in Section 6.8.

6.3 Deployments

We have deployed two sensor network surveillance applications in two different environments. The first network was deployed indoors in a large factory complex setting with concrete floors and walls, and the second in a combined outdoor and indoor setting in an urban environment.

In both experiments, we used ESB sensor nodes [14] consisting of a MSP430 microprocessor with 2kB RAM, 60kB flash, a TR1001 868 MHz radio and several sensors. During the deployments, we used the ESB's motion detector (PIR) and vibration sensor.

We implemented the applications on top of the Contiki operating system [5] that features the uIP stack, the smallest RFC-compliant TCP/IP stack [4]. All communication uses UDP broadcast and header compression that reduces the UDP/IP header down to only six bytes: the full source IP address and UDP port, as well as a flag field that indicates whether or not the header is compressed.

We used three different types of messages: *Measurement messages* to send sensor data to the sink, *Path messages* to report forwarding paths to the sink, and *Alarm messages* that send alarms about detected activity.

We used two different protocols during the deployment. In the first experiment, we used a single-hop protocol where all nodes broadcast messages to the sink. In the second experiment, we used a multi-hop protocol where each node calculates the number of hops to the sink and transmits messages with a limit on hops to the sink. A node only forwards messages for nodes it has accepted to be relay node for. A message can take several paths to the sink and arrive multiple times. During the first deployment only a few nodes were configured to forward messages, but in the second deployment any node could configure itself to act as relay node.

After a sensor has triggered an alarm, an alarm message is sent towards the sink. Alarm messages are retransmitted up to three times unless the node hears an explicit acknowledgment message or overhears that another node forwards the message further. Only the latest alarm from each node is forwarded.

6.3.1 First Deployment: Factory Complex

The first deployment of the surveillance sensor network was performed in a factory complex. The main building was about 250 meters times 25 meters in size and three floors high. Both floors and most walls were made of concrete but there were sections with office-like rooms that were separated

by wooden walls. Between the bottom floor and first floor there was a smaller half-height floor. The largest distance between the sink and the most distant nodes was slightly less than 100 meters.

The sensor network we deployed consisted of 25 ESB nodes running a surveillance application. All nodes were either forwarding messages to the sink or monitored their environment using the PIR sensor and the vibration detector. We made several experiments ranging from a single hop network for measuring communication quality to a multi-hop surveillance network.

Single-Hop Network Experiment

We made the first experiment to understand the limitations of communication range and quality in the building. All nodes communicated directly with the sink and sent measurement packets at regular intervals.

Node	Distance (meter)	Walls	Received	Sent (expected)	Sent (actual)	Reception ratio (percent)	Signal strength (avg,max)	
2	65	1 C	92	621	639	15%	1829	2104
3	21	1 W	329	587	588	56%	1940	2314
4	55	1 C	72	501	517	14%	1774	1979
5	33	2 W	114	611	613	19%	1758	1969
6	18	1 W	212	580	590	37%	1866	2230
7	26	2 W	347	587	588	59%	2102	2568
8	15	1 W	419	584	585	71%	2131	2643
9	25	1 W	194	575	599	34%	1868	2218
10	23	2 W	219	597	599	37%	1815	2106
11	17	1 W	331	591	593	56%	2102	2582
50	27	2 W	230	587	594	39%	1945	2334

Table 6.1: Communication related measurements for the first experiment.

Table 6.1 shows the results of the measurements. The columns from left are node id, distance from the sink in meters, number of concrete and wooden walls between node and the sink, number of messages received at the sink from the node, number of messages sent by the node (calculated on sequence number), actual number of messages sent (read from a log stored in each node), percentage of successfully delivered messages, and signal strength measured at the sink. Table 6.1 shows that as expected the ratio of received messages decreases with increasing distance from the sink. As full sensor coverage of the factory was not possible using a single-hop network, we performed the other experiments with multi-hop networks.

Multi-Hop Network Experiments

After performing some experiments to understand the performance of a multi-hop sensor network with respect to communication and surveillance coverage, we performed the final experiment in the first deployment during a MOUT (Military Operation on Urban Terrain) exercise with 15-20 soldiers moving up and down the stairs and running in the office complex at the top level of the building.

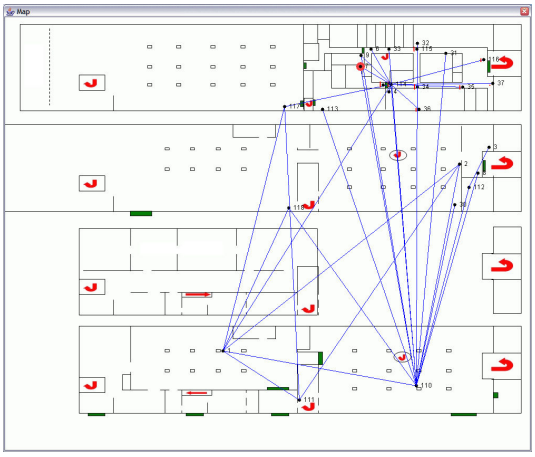


Figure 6.1: Screenshot from the final experiment in the factory deployment illustrating placement of the sensor nodes during the surveillance and the paths used to transfer messages. All levels of the factory are shown with the top level at the top of the screenshot and the basement at the bottom. The office complex is on the right side at the top level in the figure.

Node 110 (see Figure 6.1) was the most heavily loaded forwarding node in the network. It had a direct connection to the sink and forwarded 10270 messages from other nodes during the three hour long experiment. During the experiment the sink received 604 alarms generated by node 110. 27 percent of these alarms were received several times due to retransmissions. Node 8 had seven paths to the sink, one direct connection with the sink, and six paths via different forwarding nodes. The sink received 1270 unique alarms from node 8 and 957 duplicates. Most of the other nodes' messages multi-hopped over a few alternative paths to the sink, with similar or smaller delays than those from node 110 and 8. This indicates that the network was reliable and that most of the alarms got to the server; in many cases via several paths. With the 25 sensor nodes we achieved coverage of the most important passages of the factory complex, namely doors, stairs, and corridors.

6.3.2 Second Deployment: Combined in-door and out-door urban terrain

The second deployment was made in an artificial town built for MOUT exercises. It consisted of a main street and a crossing with several wooden buildings on both sides of the streets. At the end of the main street there were some concrete buildings. The distance between the sink and the nodes at the edge of the network was about 200 meters, making the network more

than twice as long as in the first deployment.

The surveillance system was improved in two important ways. First, the network was more self-configuring in that there was no need for manually configuring which role each node should have (relay node or sensor node). Each node configured itself for relaying if the connectivity to sink was above a threshold. Second, alarm messages also included path information so that information of the current configuration of the network was constantly updated as messages arrived to the sink. Even with the added path information in the alarm messages, the response times for alarms in the network were similar to the response times in the first deployment despite that the distant nodes were three or four hops away from the sink rather than two or three. Using 25 nodes we achieved fairly good sensor coverage of the most important areas.

6.4 Deployment Experiences

When we deployed the sensor network application we did not know what to expect in terms of deployment speed, communication quality, applicability of sensors, etc. Both deployments were made in locations that were new to us. This section reports on the various experiences we made during the deployments.

Network Configuration

During the first deployment the configuration needed to make a node act as a relay node was done manually. This made it very important to plan the network carefully and make measurements on connectivity at different locations in order to get an adequate number of forwarding nodes. This was one of the largest problems with the first deployment. During the second deployment the network's self-configuration capabilities made deployment a faster and easier task.

The importance of self-configuration of the network routing turned out to be higher than we expected since we suddenly needed to move together with the sink to a safer location during the second deployment, where we did not risk being fired at. This happened while the network was deployed and active.

Unforeseen Hardware Problems

During radio transmissions a few of the sensor nodes triggered sensor readings which cause unwanted false alarms. Since we detected and understood this during the first deployment, we rewrote the application to turn off sensing on the nodes that had this behavior. Our long term solution is described in the next section.

Parameter Configuration

In the implementation of the communication protocols and surveillance application there are a number of parameters with static values set during early testing with small networks. Many of these parameters need to be optimized for better application performance. Due to differences in the environment, this optimization can only partly be done before deployment. Examples of such parameters are retransmission timers, alarm triggering delays, radio transmission power level, and time before refreshing a communication link.

Ground Truth

It is important for understanding the performance of a sensor network deployment to compare the sensed data to ground truth. In our deployments, we did not have an explicit installation of a parallel monitoring system to obtain ground truth, but in both deployments we received a limited amount of parallel feedback.

During the first deployment, the sensor networks alerted us of movements in various parts of the factory but since we did not have any information about the soldiers' current locations it was difficult to estimate the time between detection by the sensor nodes and the alarm at the sink. Sometimes the soldiers threw grenades powerful enough to trigger the vibration sensors on the nodes. This way, we could estimate the time between the grenade explosions and the arrival of the vibration alarm at the sink. During the second deployment we received a real time feed from a wireless camera and used it to compare the soldiers' path with the alarms from the sensor network.

Radio Transmission

During the first deployment we placed forwarding nodes in places where we expected good radio signal strength (less walls, and floors). We expected the most used path to the sink via forwarding nodes in the stairwells. However, most messages took a path straight through two concrete floors via a node placed at the ground floor below the office rooms where the sensors were deployed.

Instant Feedback

One important feature of the application during deployment was that when started, a node visualized its connection. When it connected, the node beeped and flashed all its leds before being silent. Without this feature we would have been calling the person at the sink all the time just to see if the node had connected to the network. This way, we could also estimate a node's link quality. The longer time for the node to connect to the network,

the worse it was connected. We usually moved nodes that required more than 5 - 10 seconds to connect to a position with better connectivity.

6.5 A Self-Monitoring Architecture for Detecting Hardware and Software Problems

To ensure automatic detection of the nodes that have hardware problems, we design a self-monitoring architecture that probes potential hardware problems. Our experiences show that the nodes can be categorized into two types: those with a hardware problem and those without. The self-probing mechanism could therefore possibly be run at start-up, during deployment, or even prior to deployment.

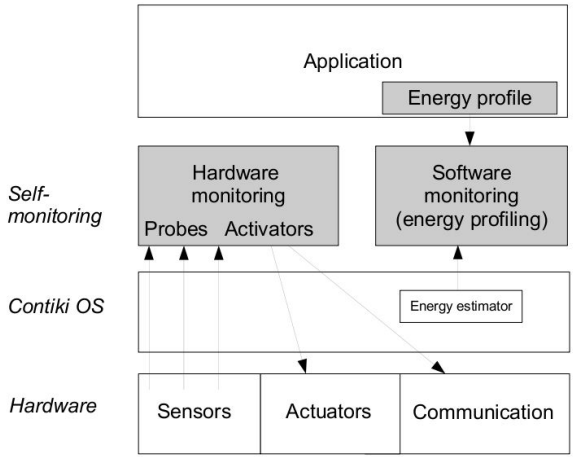


Figure 6.2: Architecture for performing self-monitoring of both hardware and software. Self-tests of hardware are performed at startup or while running the sensor network application. Monitoring of the running software is done continuously using the built-in energy estimator in Contiki.

6.5.1 Hardware Self-Test

Detection of hardware problems is done using a self-test at node start-up. The goal of the self-test is to make it possible to detect if a node has any hardware problems.

The self-test architecture consists of pairs of probes and activators. The activators start up an activity that is suspected to trigger problems and the probes measure if sensors or hardware components react to the activator's activity. An example of a probe/activator pair is measuring PIR interrupts when sending radio traffic. The API for the callbacks from the self tester

for probing for problems, executing activators and handling the results are shown in Figure 6.3.

```
int probe();
void execute_activator();
void report(int activator, int probe, int percentage);
```

Figure 6.3: The hardware self-test API.

With a few defined probes and activators, it is possible to call a self-test function that will run all probe/activator pairs. If a probe returns anything else than zero, this is an indication that a sensor or hardware component has reacted to the activity caused by the activator. The code in Figure 6.4 shows the basic algorithm for the self-test. The code assumes that the activator takes the time it needs for triggering potential problems, and the probes just read the data from the activators. This causes the self-test to monopolize the CPU, so the application can only call it when there is time for a self-test.

```
/* Do a self-test for each activator */
for(i = 0; i < activator_count; i++) {
  /* Clear the probes before running the activator */
  for(p = 0; p < probe_count; p++) {
    probe[p]->probe();
    probe_data[p] = 0;
  }
  for(t = 0; t < TEST_COUNT; t++) {
    /* run the activator and probe all the probes */
    activator[i]->execute_activator();
    for(p = 0; p < probe_count; p++)
      probe_data[p] += probe[p]->probe() ? 1 : 0;
  }
  /* send a report on the results for this activator-probe pair */
  for(p = 0; p < probe_count; p++)
    report(i, p, (100 * probe_data[p]) / TEST_COUNT);
}
```

Figure 6.4: Basic self-test algorithm expressed in C-code.

The self-test mechanism can either be built-in into the OS or a part of the application code. For the experiments we implement a self-test component in Contiki on the ESB platform.

A component on a node can break during the network's execution (we experienced complete breakdown of a node due to severe physical damage). In such case the initial self-test will not automatically detect the failure. Most sensor network applications have moments of low action, and in these cases it is possible to re-run the tests or parts of the tests to ensure that no new hardware errors have occurred.

6.5.2 Software Self-Monitoring

Monitoring the hardware for failure is taking care of some of the potential problem in a sensor network node. Some bugs in the software can also cause unexpected problems, such as the inability to put the CPU into low power mode [8]. This can be monitored using Contiki's energy estimator [6] combined with energy profiles described by the application developer.

```
ENERGY_PROFILE(60 * CLOCK_SECOND,      /* Check profile every 60 seconds */
               energy_profile_warning, /* Call this function if mismatch */
               EP(CPU, 0, 20),         /* CPU 0%-20% duty cycle */
               EP(TRANSMIT, 0, 20),    /* Transmit 0%-20% duty cycle */
               EP(LISTEN, 0, 10));     /* Listen 0%-10% duty cycle */
```

Figure 6.5: An energy profile for an application with a maximum CPU duty cycle of 20 percent and a listen duty cycle between 0 and 10 percent. The profile is checked every 60 seconds. Each time the system deviates from the profile, a call to the function *energy_profile_warning* is made.

6.5.3 Self-Configuration

Based on the information collected from the hardware and software monitoring the application and the operating system can re-configure to adapt to problems. In the case of the surveillance application described above the application can turn off the PIR sensor during radio transmissions if a PIR hardware problem is detected.

6.6 Evaluation

We evaluate the self-monitoring architecture by performing controlled experiments with nodes that have hardware defects and nodes without defects. We also introduce artificial bugs into our software that demonstrate the effectiveness of our software self-monitoring approach.

6.6.1 Detection of Hardware Problems

For the evaluation of the hardware self-testing we use one probe measuring PIR interrupts, and activators for sending data over radio, sending over RS232, blinking leds and beeping the beeper. The probes and activators are used to run the tests on ten ESB nodes of which two are having the hardware problems.

A complete but simplified set of probes, activators and report functions is shown in Figure 6.6. In this case, the results are only printed instead of used for deciding how the specific node should be configured.

```
/* A basic PIR sensor probe */
static int probe_pir(void) {
  static unsigned int lastpir;
  unsigned int value = lastpir;
  lastpir = (unsigned int) pir_sensor.value(0);
  return lastpir - value;
}

/* A basic activator for sending data over radio */
static void activator_send(void) {
  /* send packet */
  rimebuf_copyfrom(PACKET_DATA, sizeof(PACKET_DATA));
  abc_send(&abc);
}

/* Print out the report */
static void report(int activator, int probe, int trigged_percent) {
  printf("Activator %u Probe %u: %u%%\n", activator, probe, trigged_percent);
}
```

Figure 6.6: A complete set of callback functions for a self test of radio triggered PIR sensor.

As experienced in our two deployments, the PIR sensor triggers when transmitting data over the radio during the tests on a problem node. On other nodes the PIR sensors remain untriggered. Designing efficient activators and probes is important for the self-monitoring system. Figure 6.7 illustrates the variations in detection ratio when varying the number of transmitted packets and packet sizes during execution of the activator. Based on these results a good activator for the radio seems to be sending three 50 bytes packets.

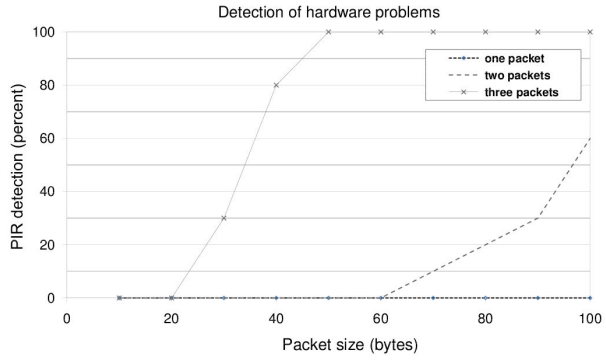


Figure 6.7: Results of varying the activator for sending radio packets on a problem node. Sending only one packet does not trigger the PIR probe, while sending three packets with a packet size larger than 50 bytes always triggers the problem. On a good node no PIR triggerings occur.

6.6.2 Detection of Software and Configuration Problems

Some of the problems encountered during development and deployment of sensor network software are related to minor software bugs and misconfigurations that decrease the lifetime of the network [8]. Bugs such as missing to power down a sensor or the radio chip when going into sleep mode, or a missed frequency divisor and therefore higher sample rate in an interrupt driven A/D based sensor can decrease the network's expected lifetime. We explicitly create two software problems causing this type of behavior.

The first problem consists of failing to turn off the radio in some situations. Figure 6.8 shows the duty cycle of the radio for an application that periodically sends data. XMAC [2] is used as MAC protocol to save energy. XMAC periodically turns on the radio to listen for transmissions and when sending it sends several packet preambles to wake up listeners. Using the profile from Figure 6.5 a warning is issued when the radio listen duty cycle drastically increases due to the software problem being triggered.

In the second problem the sound sensor is misconfigured causing the A/D converter to run at twice the desired sample rate. The node then consumes almost 50% more CPU time than normal. This misbehavior is also detected by the software self-monitoring component.

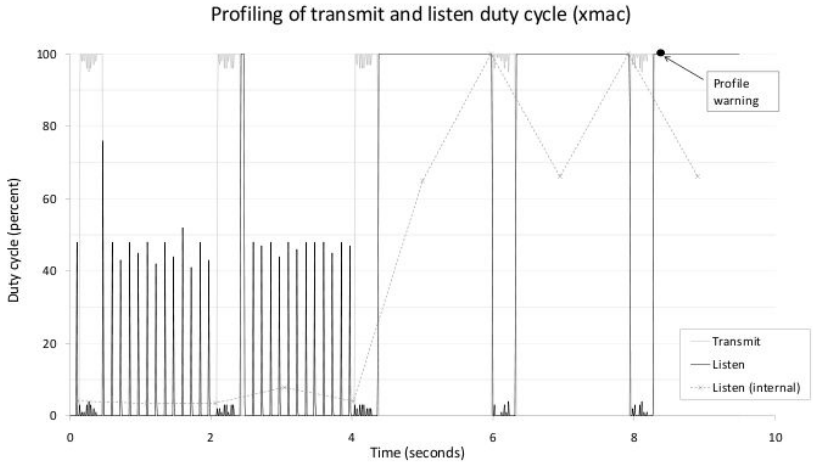


Figure 6.8: Duty-cycle for listen and transmit. The left part shows the application's normal behavior with a low duty cycle on both listen and transmit. The right part shows the behavior after triggering a bug that causes the radio chip to remain active. The dashed line shows the listen duty cycle estimated using the same mechanism as the energy profiler but sampled more often. The next time the energy profile is checked the deviation is detected and a warning issued.

6.7 Related Work

During recent years several wireless sensor networks have been deployed the most prominent being probably the one on Great Duck Island [9]. Other efforts include glacier [11] and water quality monitoring [3]. For an overview on wireless sensor network deployments, see Römer and Mattern [13].

Despite efforts to increase adaptiveness and self-configuration in wireless sensor networks [10, 12, 16], sensor network deployments still encounter severe problems: Werner-Allen et al. have deployed a sensor network to monitor a volcano in South America [15]. They encountered several bugs in TinyOS after the deployment. For example, one bug caused a three day outage of the entire network, other bugs made nodes lose time synchronization. We have already mentioned the project by Langendoen that encountered severe problems [8]. Further examples include a surveillance application called "A line in the sand" [1] where some nodes would detect false events exhausting their batteries early and Lakshman et al.'s network that included nodes with a hardware problem that caused highly spatially correlated failures [7]. Our work has revealed additional insights such as a subset of nodes having a specific hardware problem where packet transmissions triggered the motion detector.

6.8 Conclusions

In this paper we have reported results and experiences from two sensor network surveillance deployments. Based on our experiences we have designed, implemented and evaluated an architecture for detecting both hardware and software problems. The evaluation demonstrates that our architecture detects and handles hardware problems we experienced and software problems experienced during deployments of other researchers.

Acknowledgments

This work was funded by the Swedish Defence Materiel Administration and the Swedish Agency for Innovation Systems, VINNOVA.

Bibliography

[1] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks*, 46(5):605–634, 2004.

[2] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 307–320, Boulder, Colorado, USA, 2006.

[3] T.L. Dinh, W. Hu, P. Sikka, P. Corke, L. Overs, and S. Brosnan. Design and Deployment of a Remote Robust Sensor Network: Experiences from an Outdoor Water Quality Monitoring Network. *IEEE Congf. on Local Computer Networks*, pages 799–806, 2007.

[4] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, May 2003.

[5] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[6] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based online energy estimation for sensor nodes. In *Proceedings of the Fourth Workshop on Embedded Networked Sensors (Emnets IV)*, pages 28–32, Cork, Ireland, June 2007.

[7] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea. In *ACM SenSys*, pages 64–75, 2005.

[8] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, April 2006. IEEE.

[9] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, USA, September 2002.

[10] P. Marron, A. Lachenmann, D. Minder, J. Hahner, R. Sauter, and K. Rothermel. TinyCubus: a flexible and adaptive framework for sensor networks. *EWSN 2005*, 2005.

[11] P. Padhy, K. K. Martinez, A. Riddoch, H. Ong, and J. Hart. Glacial environment monitoring using sensor networks. In *Proc. of the Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, Stockholm, Sweden, June 2005.

[12] I. Rhee, A. Warrier, M. Aia, and J. Min. Z-MAC: a hybrid MAC for wireless sensor networks. *ACM SenSys*, pages 90–101, 2005.

[13] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004.

[14] J. Schiller, H. Ritter, A. Liers, and T. Voigt. Scatterweb - low power nodes and energy aware routing. In *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, USA, 2005.

[15] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation 2006*, Seattle, November 2006.

[16] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, New York, NY, USA, 2003. ACM Press.

Chapter 7

Improving Sensornet Performance by Separating System Configuration from System Logic

Niclas Finne, Joakim Eriksson, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt.
Swedish Institute of Computer Science, SICS.
{nfi,joakime,nvt,adam,thiemo}@sics.se

7.1 Abstract

Many sensor network protocols are self-configuring, but independent self-configuration at different layers often results in suboptimal performance. We present Chi, a full-system configuration architecture that separates system logic from system configuration. Drawing from concepts in artificial intelligence, Chi allows full-system configuration that meets both changing application demands and changing environmental conditions. We show that configuration policies using Chi can improve throughput and energy efficiency without adding dependencies between layers. Our results show that sensornet systems can use Chi to adapt to changing conditions at all layers of the system, thus meeting the requirements of heterogeneous and continuously changing system conditions.

7.2 Introduction

The sensornet community is moving toward modular architectures that allow a clean separation of concerns [3, 6, 7, 9, 20]. So far, however, the perfor-

mance of such modularized designs has been dissatisfying due to problems with cross-layer interactions. For example, Kim et al. write [12]: *"[...] there is still a large performance gap to the raw radio bandwidth that would require a cross-layer design and integration with the MAC and the packet processing in the OS."* Similarly, experience from sensornet deployments [17] has shown the need for configuration across modules and for gathering system statistics.

Cross-layer optimizations have primarily been implemented by coupling the programming interfaces of different components. Hence, using cross-layer designs typically requires that we sacrifice system modularity to improve system performance. As a remedy to this problem, researchers have proposed specialized architectures for cross-layer optimization [13, 16, 10]. These specialized architectures enable applications to be configured to meet energy efficiency goals. We argue, however, that since energy efficiency is not the only objective of a sensornet, a cross-layer optimization architecture should be able to focus on other metrics as well.

We present Chi, a lightweight architecture that enables cross-layer optimizations in sensornet systems without requiring an unmodular cross-layer design. The main design principle of Chi is that components must not be required to have any knowledge of parameters exported by other components. Instead, all such knowledge is in separate components that enforce configuration policies. By using Chi, we maintain the separation of concerns between layers while providing the same performance as integrated cross-layer optimizations. In contrast to previous work, Chi takes a generalized approach to configuration that enables systems to be optimized not only to meet energy objectives but also to meet other objectives such as latency, throughput, and sensor coverage.

The Contiki operating system separates protocol logic from protocol headers to achieve network protocol modularity with retained execution-time efficiency [6]. Similarly, Chi provides system configuration modularity with low run-time overhead. We draw from the work made within the autonomic computing community on blackboard systems [4]. The central component in Chi is a blackboard that holds the system configuration and relevant parts of the system state. Storing the configuration in one component simplifies updates made by external configuration modules, as illustrated in Figure 7.1. In addition to the blackboard component, Chi accommodates configuration policies written to optimize a sensornet toward different objectives.

Our contribution is to show that Chi solves the problem of cross-layer optimization for sensornet systems using a generalized programming abstraction. We demonstrate the abilities of Chi in three case studies, showing that an application using holistic configuration outperforms the same application when using constant parameter settings of the protocol layers, as well as when using several adaptive layers. We show that by using Chi the

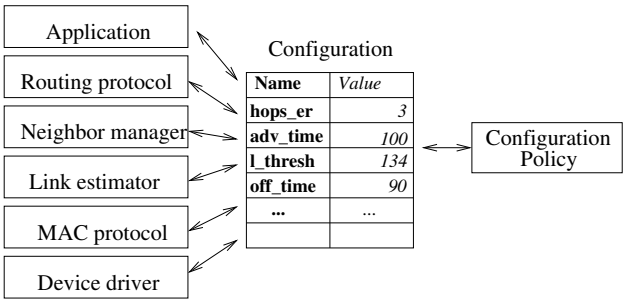


Figure 7.1: Chi keeps the configuration data of different system layers in a central component. The configuration can be changed by a configuration policy. Unlike existing cross-layer architectures, individual protocol layers are unaware of the configuration of the other layers.

TCP performance increases by an order of magnitude without putting any cross-layer logic within the networking layers. Instead, the cross-layer optimizations are put into separate configuration policies and are decoupled from all protocol implementations.

The rest of this paper is structured as follows. We present the background of cross-layer design, holistic configuration and blackboard systems in Section 7.3. We describe Chi in Section 7.4 and its implementation in the Contiki operating system in Section 7.5. We evaluate the architecture in Section 7.6, present related work in Section 7.7, and conclude the paper in Section 7.8.

7.3 Background

Layering separates different concerns in a network architecture and reduces the design complexity. The plethora of routing, transport, and medium access control protocols for sensor networks has made layering the prevalent design choice also in sensor network architectures. The high modularity of layering, however, restricts the collaboration of different layers that could potentially benefit from sharing each others unique information.

7.3.1 Performance Improvements through Cross-Layer Design

Achieving near-optimal throughput and energy consumption is difficult when using independently designed protocols in different layers. This is easier in vertically integrated systems that benefit from a coordinated design of several layers. Koala [19] and Dozer [2] are examples of vertically integrated systems with duty cycles less than 1% in low power data gathering applications. Their specific target in design, however, restricts them from achieving the same efficiency when adapting to different network traffic patterns.

Cross-layer approaches make existing protocols more adaptive to different workload patterns by allowing the layers to interact and share information. Previous research on cross-layer design for sensor networks has led to improvements in energy-efficiency [8, 18]. The negative consequence is that cross-layer design, by definition, increases the coupling between modules.

7.3.2 Cross-Layer Design Breaks Layering

Cross-layer designs have been criticized as leading to "spaghetti design" [11]. Without layering it is difficult to achieve adequate separation of concerns which may lead to stability problems and negative impact on performance. Additionally, tight coupling decreases the modularity of the architecture and makes it complex to replace modules.

7.3.3 Holistic Configuration

By holistic configuration, we mean that parameters in all parts of the system can be configured through a separate configuration component, called a configuration policy in Chi. While layered systems usually store configuration parameters in module variables, a separate configuration policy enables simultaneous configuration of the whole system. This makes it possible to add external algorithms that optimize the system using information from multiple layers. The optimization objectives can for instance be low energy consumption, minimum delay, or maximum throughput. Changing the applications objectives is simple because only the configuration policy needs to be updated.

A configuration policy can be implemented in plain C using basic if-then-else statements that optimize the configuration toward the application objectives. It can also consist of a rule engine and a set of rules that are triggered when the system state changes. If the application objectives change, the configuration policy can be replaced to reflect the new objectives.

7.3.4 Blackboard Systems

A blackboard [4] is a concept used within the artificial intelligence community. Conceptually, a blackboard is a tool used by a group of experts to solve a complex problem. In a software system, the blackboard is a component that typically stores key-value pairs, and has a mechanism for notifying interested components when a value changes. The classic blackboard design consists of a blackboard, a set of independent knowledge sources, and a number of control components. The knowledge sources have both the knowledge and the algorithms needed to solve a specific problem, whereas the control components steer the execution order and triggering. The traits of the blackboard makes it a suitable solution for the requirements of a holistic configuration architecture.

7.4 Chi: A Full-System Configuration Architecture

We have designed Chi[1], a full-system configuration architecture that uses a blackboard to enable cross-layer information sharing despite keeping modules decoupled. The blackboard provides a shared variable abstraction that is accessed in an independent module in each sensor node. Beside providing a programming interface for accessing variables, the blackboard has a notification process for subscribers of value modifications. As illustrated in Figure 7.1, modules export their configuration parameters through the blackboard. Configuration policies can then use any of the available parameters to optimize the system for any objectives determined by the application.

Chi is designed to be a dynamic configuration architecture. In a modular system such as Contiki, in which software modules can be loaded at runtime, it is necessary that also the configuration architecture can accommodate new parameters and configuration policies. New insights on protocol optimizations are easily integrated into deployed networks because the configuration policies are replaceable.

7.4.1 Separating Configuration from Logic

Chi separates system logic from system configuration to make it possible to alter the configuration without having to change the logic. Moreover, system modules do not need to contain any logic for updating their configuration: this service is provided by Chi and the configuration policies used in the system.

Existing mechanisms conflate logic and configuration by storing configuration parameters as module variables. To change the configuration of a module, the internal variables need to be changed. Thus the module must contain logic for storing and retrieving configuration parameters. Without a consistent interface for storage and retrieval of configuration parameters, every module provides its own mechanism for doing so, leading to systems that are difficult to reconfigure. Chi provides a consistent interface for storing and retrieving configuration parameters from the blackboard, thus making it possible to reconfigure the entire system using a single interface.

Configuration and data sharing abstractions that are aware of the details in specific protocols can be implemented by using the generalized Chi architecture. For instance, a network-based data sharing abstraction such as Hood [22] could store reflected data variables locally in Chi. The need for creating parameters dynamically–which is possible in Chi–is highlighted in

[1]The name Chi comes from the Greek letter χ, representing the cross-layer information sharing that the architecture enables.

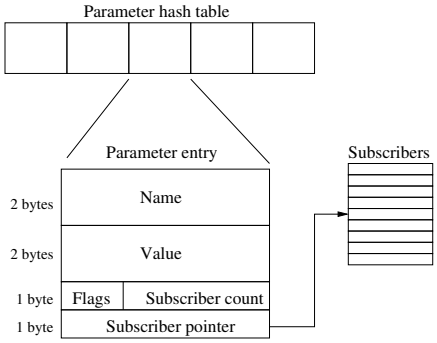


Figure 7.2: The memory layout in the blackboard. Parameters are stored in a hash table for fast lookup. Subscribers are listed as function pointers in a separate table.

the case of deploying a heterogeneous sensor network where different types of sensor nodes may want to share different parameters.

Sensornet protocols are in general unaware of which specific configuration parameters that protocols at other layers make available. A configuration policy, as previously depicted in Figure 7.1, therefore handles the protocol-specific optimization. If the protocols change, for example by code dissemination and dynamic loading in a deployed system, a corresponding update must be made with the configuration policy. In the evaluation (Section 7.6), we will show three configuration policies in practice.

7.4.2 Inter-Layer Information Hiding

To achieve a meaningful separation of concerns, it is important that different layers of the system are independent of each other. If modules in adjacent layers would depend on having information about each other, it would be difficult to replace them. Instead of depending on inter-layer information sharing to enable cross-layer optimization, Chi moves the information sharing from the inter-layer interface into the blackboard to keep the layers separate. This ensures that inter-layer interfaces focus on the abstractions provided by each layer and not on information sharing between layers.

7.4.3 State Monitoring

State information can serve as input for configuration policies. Components monitor different parts of the node state and publish the state information in the blackboard. State can be collected either in the nodes through active monitoring, or it may require communication with neighboring nodes to gain a complete picture of the surroundings of a node.

In some cases, the information published by one component may not be

directly usable by others. Chi allows reusable components to be plugged in to process data and to produce a refined output. For example, a network statistics component can take the raw packet statistics produced by the network stack and calculate whether the node is in bulk traffic mode or in passive mode. This information can then be used by a configuration policy to optimize the system based on a refined input.

The parameter subscription mechanism in Chi ensures that interested parties are notified if a parameter value changes. Just holding the shared state would have required that modules periodically poll the blackboard for changes, which would add latency to the information exchange and increase the processing energy. In particular, configuration policies regularly require that components must react within a limited time after the event occurs.

7.5 Implementation

We have implemented the Chi architecture in Contiki, but the architecture is general enough to be portable to other systems. Figure 7.2 illustrates the memory layout. The blackboard component uses two tables: the parameter hash table and the subscriber table. Parameters are represented by a name pointer, a value, a set of flags, the number of subscribers, and a pointer to the first subscriber in the subscriber table. We restrict the values to be of integer type to have a concise API. Moreover, we have not identified any need for other types. When compiled for 16-bit computing architectures such as the MSP430, each parameter requires six bytes, whereas the subscriber table requires two bytes per subscriber to store pointers to callback functions.

Chi's API consists of eight functions. The configuration parameters are denoted by textual names, such as "mac.off_time" and "measure.period". The *set* function assigns a value to a configuration parameter. The *get* function obtains the last set value. Whether or not a value has been set is checked with the *exists* function. The *subscribe* function registers a callback function as a subscriber to a specified configuration parameter. The callback function is called whenever the parameter is changed using the *set* function. The *unsubscribe* function removes a previously registered callback.

Parameter values can be read and written without subsequent parameter lookups in the hash table by holding a one-byte index for the parameter. The *lookup* function returns an index value if the parameter exists. Thereafter, the *entry_get* and *entry_set* functions will provide significantly faster access to the parameter by using the index value.

7.6 Evaluation

We evaluate Chi through a series of experiments to determine whether it achieves the same performance improvements as those of typical cross-layer

designs, and whether the dynamic properties of the architecture results in any performance penalty. The first experiment is inspired by a condition monitoring application that we evaluate by comparing the performance of different optimization principles. In the second experiment we show that the separation of configuration and logic makes it possible to reuse optimizations in configuration policies between different applications and communication stacks. The third experiment demonstrates that Chi improves the communication performance as much as that of a specialized architecture for application feedback to the MAC layer. Lastly, we evaluate Chi through a set of micro benchmarks where we measure the cycle count of each operation.

7.6.1 Case Study: Condition Monitoring with Bulk Transfer

To quantify the effectiveness of Chi, we implement a condition monitoring application in Contiki using a holistic configuration policy. We compare the performance with three other types of network stack designs: constant configuration, adaptive layers, and cross-layer optimizations. The application has the same communication behavior as applications for condition monitoring of industrial motors: it samples large chunks of vibration data periodically, and sends the data to a sink for processing and analysis. A data chunk is typically a few kilobytes large. We apply the configuration policy on the X-MAC protocol [1] and the Rime communication stack [6]. To support bulk transfers over a multi-hop network, we implement a bulk transport layer using Rime's data collection abstraction. We use three Tmote Sky nodes forming a two-hop network.

Constant Configuration

The version with constant parameters uses 20 ms wake time and 480 ms sleep time, resulting in a duty cycle of 4%. On the layers above the MAC layer, we set the routing advertisement interval to 60 s, the data packet transmission interval to 1 s, and the retransmission timeout to 1 s.

Adaptive Layers

Adaptive layers means that each layer optimizes itself using internal knowledge. In this experiment, we implement the version of the X-MAC protocol [1] that adapts to the traffic load using only information about the packets sent and received.

The adaptive layers setup is similar to that of the constant configuration setup, but to avoid collisions, the routing advertisement rate is adapted by delaying the next advertisement by 10 s after receiving a packet. The transport layer adapts the send rate in order to increase the throughput. If there is a large delay between a sent packet and its corresponding acknowledgement, we reduce the send rate. If the delay is below a certain threshold, we

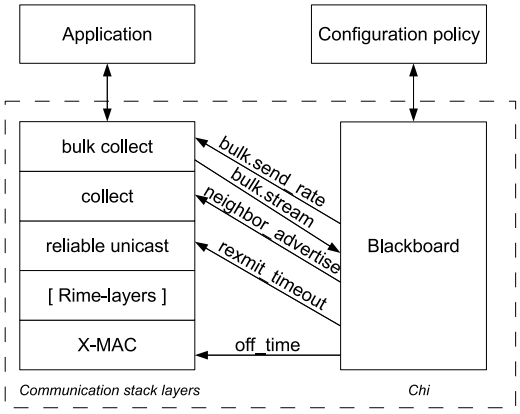


Figure 7.3: The condition monitoring application and its parts.

increase the send rate as much as possible while keeping the delay below the threshold. A large delay indicates retransmissions in the lower layers and that the next node in the route may be overloaded.

A Cross-Layer Design

The cross-layer design in this experiment combines information from the transport layer and the link layer. The consequence is that the bulk transport module must be coupled directly with the MAC protocol implementation. The bulk transport module must know of MAC-specific variables or functions that will no longer be valid if the MAC protocol would be switched. Moreover, the data collection module is coupled directly with the bulk transport module to know when not to send routing advertisements.

In this experiment, the transport layer sets a flag as a global variable and reconfigures the MAC layer at the beginning and end of a bulk transfer. The send rate is fixed at 20 packets per second. While a bulk transport is active, the data collection module withholds routing advertisements, and the reliable unicast layer uses a shorter retransmission timeout of 0.5 s.

The Chi Design

The Chi design consists of a policy that optimizes the system through a set of parameters in multiple layers. Whereas the cross-layer design described in the previous section modifies the communication stack to fit application requirements, the Chi design uses the communication stack without changing any interfaces or intra-layer logic.

Figure 7.3 shows the configuration policy that manages the reconfiguration, and Figure 7.4 depicts the corresponding code. Reconfiguration decisions are primarily affected by the bulk transfer parameter. When the

```
/* This function is called when "bulk.stream" changes */
void stream_changed(const char *name, int value) {
  if(value != 0) {
    set("mac.off-time", 0);
    set("bulk.send-rate", 20);
    set("collect.routing-advertisements", 0);
    set("runicast.rexmit-time", CLOCK_SECOND / 2);
  } else {
    /* Restore default configuration */
    set("mac.off-time", MAC_DEFAULT_OFF_TIME);
    set("bulk.send-rate", 1);
    set("collect.routing-advertisements", 1);
    set("runicast.rexmit-time", DEFAULT_REXMIT_TIME);
  }
}
/* Register a callback for "bulk.stream" */
subscribe("bulk.stream", stream_changed);
```

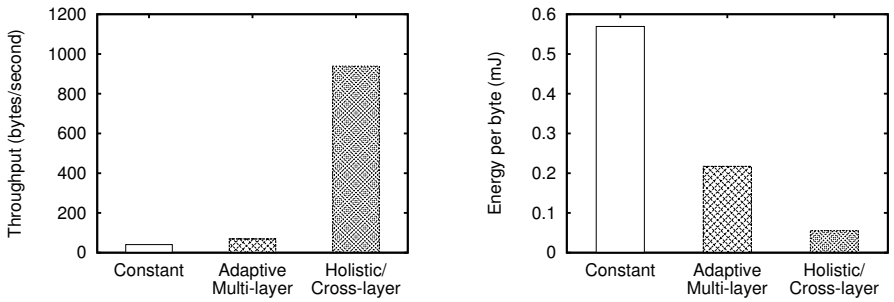
Figure 7.4: The configuration policy reconfigures the communication stack based on routing information from the network layer. If the system indicates that a bulk transfer is about to occur, the policy sets the system in high throughput mode.

application switches between regular transfer and bulk transfer, it causes a global reconfiguration of the communication stack. This cannot be done with a constant configuration since it would be optimized for either energy efficiency or throughput. Our approach is to use configuration policies optimized for the specific application and reconfigure the communication stack when needed.

Throughput and energy consumption

We show the results in Figure 7.5. Since the results achieved with the cross-layer design and the Chi design are identical, they are shown as one result in the table. When using constant parameters, the throughput is quite low and the power consumption is moderate. The result depends on the chosen parameters that, as discussed above, lead to a duty cycle of around 4%. Although a higher duty cycle leads to higher throughput, the power consumption increases. The adaptive layers design is considerably more efficient than the design with constant parameters. The power consumption decreases to about one third compared with constant parameter setting.

Both the Chi design and the cross-layer design yield a high throughput and a low power consumption. The reason is that the nodes can immediately switch from low power mode to high throughput mode when a bulk transfer begins since the application demands are known by the configuration policy. In the cross-layer design, the same demands are hard-wired into the different layers.



(a) Throughput (b) Energy

Figure 7.5: The holistic configuration outperforms the constant configuration and the multi-layer self-adaptive configuration both in throughput and energy

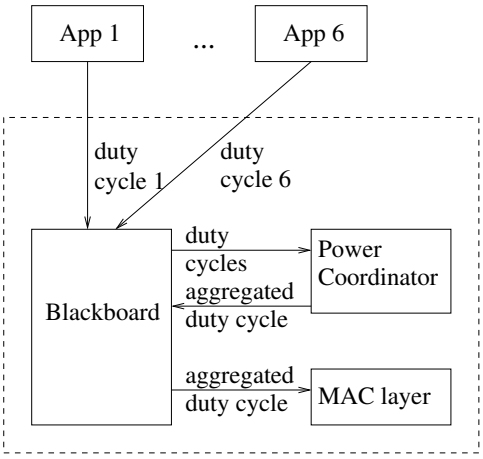


Figure 7.6: The power coordinator uses Chi to coordinate duty cycles.

7.6.2 Case Study: TCP Optimization over a Power-Saving MAC Protocol

The second case study shows that we can reuse optimizations of a Chi configuration policy in a different type of application using another communication stack. The application studied is a web-service application running HTTP over TCP/IPv6 [7] with X-MAC as the MAC protocol. The scenario consists of a client that connects to a server, transfers some data, and then closes the connection. For this purpose we use two Tmote Sky nodes that are able to communicate directly with each other.

This scenario differs from the previous scenario since this is a request-response scenario and the first was a bulk transfer. There are also similarities, however, since both scenarios transfer many packets and use X-MAC. By re-using the MAC optimization of the configuration policy from the first

```
/* This function is called when "tcp.connection.count" changes */
void connections_changed(const char *name, int value) {
  if(value > 0)
    set("mac.off-time", 0);
  else
    set("mac.off-time", MAC_DEFAULT_OFF_TIME);
}
/* Register a callback for "tcp.connection.count" */
subscribe("tcp.connection.count", connections_changed);
```

Figure 7.7: The configuration policy reconfigures the MAC layer for high throughput when at least one TCP connection is active.

case, we can get a much higher performance for the web-service request. The configuration policy is illustrated in Figure 7.7. The TCP layer publishes the parameter "tcp.connection.count" and the value of this parameter is determined by counting the open TCP connections in the IP stack. The TCP layer needs no knowledge of the MAC layer and vice versa when using Chi–all cross-layer logic is put into the configuration policy.

Figure 7.8 shows the result of running the experiment with and without configuration policy optimization. As expected, the overhead of setting up a TCP connection decreases in relation to the payload size when more data is transmitted. The data rate increases with an order of magnitude when using the configuration policy. The low performance when using no optimization is caused by TCP waiting for acknowledgments for each sent packet, and both TCP segments and acknowledgments are delayed depending on where in the X-MAC duty cycle each node is. The optimization yields such a high performance by using a 100% duty cycle in X-MAC when there are active TCP connections.

7.6.3 Case Study: Aggregation of Multiple Duty Cycles

Data from different sensors can require different communication patterns depending on the deployment. Klues et al. have presented the Unified Power Management Architecture (UPMA) [13], which separates power management from MAC level functionality. UPMA is able to coordinate the duty cycles of multiple applications. Applications store their duty cycles in a Power Management Table, and UPMA uses a configured policy to coordinate the duty cycles.

In this experiment, we emulate the behavior of the aforementioned Power Management Table by using Chi. The experimental setup consists of multiple applications that periodically transmit data according to a configured duty cycle. Applications insert their duty cycles into Chi, as shown in Figure 7.6. The power coordinator subscribes to the duty cycle parameters in Chi. When an application submits its duty cycle, the power coordinator computes the aggregate duty cycle and assigns this value to the duty cycle

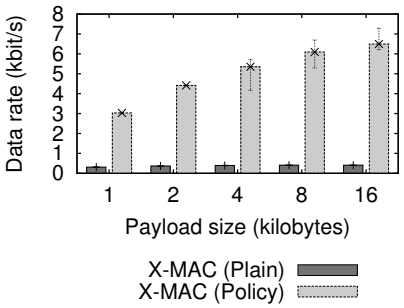


Figure 7.8: The data transfer rate of TCP/IPv6 over X-MAC with and without configuration policy optimization.

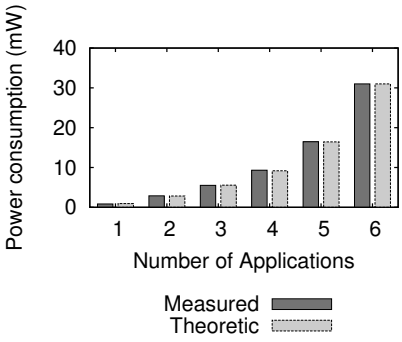


Figure 7.9: The measured power consumption of the aggregated duty cycles of multiple applications matches the theoretical values.

parameter in Chi used by the MAC protocol. We use the same duty cycles as Klues et al. in our experiment. The radio on-time is 200 ms for all applications, and the radio off-times are 12.6 s, 6 s, 3 s, 1.4 s, 600 ms, and 200 ms.

In contrast with Klues et al., we do not measure the duty cycle, which is an indirect metric for power consumption, but instead we directly measure the radio power consumption using Contiki's software-based on-line energy estimation method [5]. Using this method, we measured the power consumption of the CC2420 radio as 59.92 mW. The theoretical radio on-time for the six applications with the duty cycles as defined by Klues et al. is slightly below 52% (31.16 mW), matching our measured value of approximately 31 mW.

The results in Figure 7.9 are similar to those of Klues et al. (Figure 10, [13]) in that the theoretical values match the measured ones. The results confirm that our generalized configuration architecture achieves the same optimized radio power management as that of a specialized architecture such as UPMA.

7.6.4 Operations Benchmark

We execute a benchmark that measures the required time for the blackboard operations in Chi. We use the internal clock of the MSP430F1611 processor in a Tmote Sky node to count clock cycles. The blackboard is set up with 20 parameters and uses a hash table size with 32 entries. Despite increasing the risk of collisions, the hash table uses a size of 2^n instead of a prime number. Our experiments have shown that the performance degradation of these extra collisions is less severe than the degradation caused by the expensive modulo operation that we avoid this way.

Figure 7.10 shows the numbers of clock cycles used by the main black-

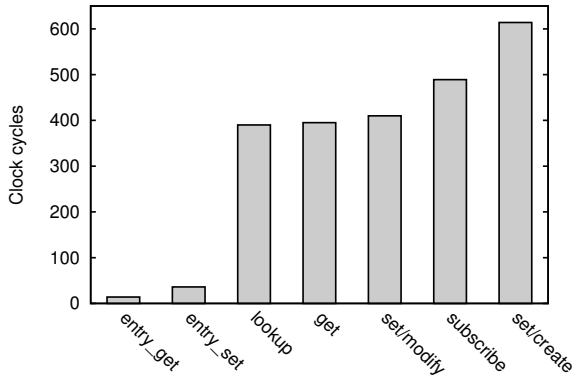


Figure 7.10: A micro benchmark for the Chi operations. The *entry_get* and *entry_set* operations are the most commonly used and are therefore optimized for execution-time efficiency.

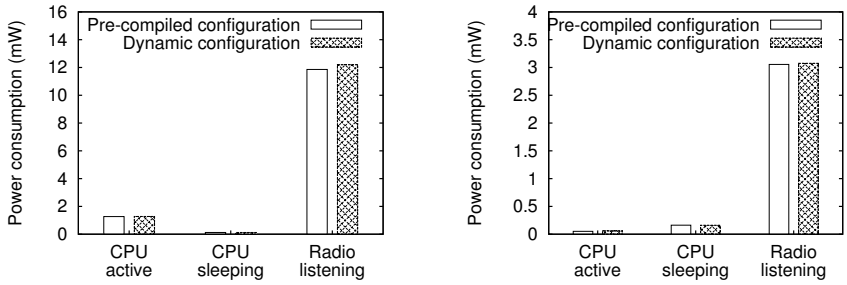
board operations. The *set* and *get* operations have a simplified interface, but require a parameter lookup in the hash table at each call. In the rare cases where parameters must be accessed several hundred times per second, the *entry_get* and *entry_set* functions provide a shorter path to the parameter by holding a pointer to it in the table. Thus the need to do a lookup is eliminated and the performance becomes comparable to that of a pre-compiled configuration approach such as TinyXXL [15].

7.6.5 Network Power Consumption

While the micro benchmark gives a clear view of the cost of configuration operations in terms of clock cycles, it is the effect on power consumption in a typical sensor network application that is the key issue. To quantify the power consumption in a sensor network, we conduct two experiments with a data-collection application. We compare the pre-compiled, constant configuration setting with the use of Chi. We measure the power using an online energy-estimation method [5], and collect the energy data when the experiment has finished. The sensor nodes communicate using Rime [6]. When running the experiment with Chi, we substitute calls to Chi for the constant configuration variables in Rime. In addition, we store communication statistics in Chi instead of in memory variables. The experiments are conducted with both a TDMA-based protocol that is specialized for data collection, and a data collection protocol using the more generic X-MAC protocol underneath.

An X-MAC-based data collection protocol

The application in this experiment measures temperature, humidity, and light, and sends this data to a base station every other second. The system



(a) X-MAC (b) TDMA

Figure 7.11: The dynamic design of Chi has a negligible impact on power consumption. The power consumption is measured for a data collection network using either X-MAC or TDMA. Note that the vertical scales are different.

uses the X-MAC protocol [1] as an energy saving MAC layer, and the data collection module in Rime to deliver the data to the sink. Route advertisements are sent once per minute. We measure the average consumed energy at each node in an indoor testbed of 15 Tmote Sky nodes during a period of 50 sensor measurements.

Figure 7.11(a) shows the average measured power consumption over five test runs with pre-compiled configuration and five corresponding runs with a configuration policy using Chi. The power consumption overhead of the dynamic configuration is on average 2.5%.

A TDMA-based data collection protocol

We build a small data collection network using a TDMA-based data collection protocol named CoReDac [21]. The leaf nodes in our network transmit a packet every 5 s. We measure the power consumption of a node in the middle of the tree with and without Chi. The result in Figure 7.11(b) shows a negligible overhead of 0.9% for the Chi-based system. As expected, slightly more CPU power is required to handle the variables stored in Chi.

7.7 Related Work

Although many sensornet protocols are adaptive, they mainly adapt by using intra-layer information. Examples include the scheduled channel polling MAC protocol [24] that changes its duty cycle using on the current traffic load and the MintRoute protocol [23] that changes its forwarding tables based on communication conditions. Independent self-adaptation at multiple layers can lead to sub-optimization where self-adaptation mechanisms at different layers counteract each other [11].

The multitude of non-standard, cross-layer designs have led to various efforts to generalize cross-layer interactions into configuration architectures. Lachenmann et al. present TinyXXL, a language and framework that supports cross-layer interactions [15]. The framework is similar to our work in that it provides a repository for storing system state and configuration. Like Chi, it supports cross-layer interaction and reconfiguration using a publish and subscribe mechanism. TinyXXL is a language extension of nesC, however, and requires recompilation when adding or removing parameters. Köpke et al. suggest using a blackboard for component-based interactions [14], but do not quantify the effects of using the blackboard. We use a similar technique for parameter storage, but focus on policy-based, cross-layer optimizations that retain the tiered networking design used in the Internet and in the Rime networking stack [6].

Several sensor network communication architectures provide mechanisms for inter-layer information sharing in the communication stack. SP [20] allows information to be shared between the link layer and the network layer. The modular network architecture by Cheng et al. [3] is a decomposition of sensornet protocols into common modules that can be shared at multiple layers. Chameleon [6] uses packet attributes to provide packet-based information sharing across layers while maintaining the separation of concerns as traditional layered architectures do. The drawback of these architectures is that they do not provide mechanisms for holistic system configuration.

Our work is also inspired by recent work on energy management architectures for sensor networks [13, 16]. Such architectures allow for applications to be configured to meet energy efficiency goals. The purpose of Chi, in contrast, is to provide a generalization of these principles that also extends to other objectives than energy-efficiency.

7.8 Conclusions

We present Chi, an architecture for full-system configuration and policy-based optimization in sensor networks. Unlike previous modular configuration architectures, Chi's dynamic properties make it possible to switch configuration policies and to add new parameters during run-time. Our experiments show that Chi improves the sensor network performance as much as specialized architectures, while maintaining a clear separation of concerns.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and by SSF. This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053.

Bibliography

[1] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 307–320, Boulder, Colorado, USA, 2006.

[2] N. Burri, P. von Rickenbach, and R. Wattenhofer. Dozer: ultra-low power data gathering in sensor networks. In *IPSN '07*, 2007.

[3] T. E. Cheng, R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A modular network layer for sensornets. In *Proceedings of OSDI*, August 2006.

[4] Daniel D. Corkill. Blackboard systems. *AI Expert*, 6(9):40–47, 1991.

[5] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based online energy estimation for sensor nodes. In *Proceedings of the Fourth Workshop on Embedded Networked Sensors (Emnets IV)*, pages 28–32, Cork, Ireland, June 2007.

[6] A. Dunkels, F. Österlind, and Z. He. An adaptive communication architecture for wireless sensor networks. In *Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems (SenSys 2007)*, Sydney, Australia, November 2007.

[7] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making Sensor Networks IPv6 Ready. In *Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008)*, pages 421–422, Raleigh, North Carolina, USA, November 2008.

[8] L. Van Hoesel, T. Nieberg, J. Wu, and P. Havinga. Prolonging the Lifetime of Wireless Sensor Networks by Cross-Layer Interaction. *IEEE Wireless Communications*, 11(6):78–86, 2004.

[9] J. Hui and D. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proceedings of the 6th international Conference on Embedded Networked Sensor Systems*, Raleigh, North Carolina, USA, November 2008.

[10] R. Jurdak, P. Baldi, and C. Videira Lopes. Adaptive low power listening for wireless sensor networks. *IEEE Transactions on Mobile Computing*, 6(8):988–1004, 2007. ISSN: 1536-1233

[11] V. Kawadia and P. Kumar. A cautionary perspective on cross-layer design. *Wireless Communications, IEEE*, 12(1):3–11, 2005. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1404568`

[12] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. Culler, P. Levis, S. Shenker, and I. Stoica. Flush: A reliable bulk transport protocol for multihop wireless networks. In *Proceedings of ACM SenSys'07*, Sydney, Australia, November 2007.

[13] K. Klues, G. Xing, and C. Lu. Link layer support for unified radio power management in wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN '07)*, pages 460–469, Cambridge, Massachusetts, USA, 2007.

[14] A. Köpke, V. Handziski, J.-H. Hauer, and H. Karl. Structuring the information flow in component-based protocol implementations for wireless sensor nodes. In *Proc. of Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN)*, Berlin, Germany, January 2004.

[15] A. Lachenmann, P. Marrón, D. Minder, M. Gauger, O. Saukh, and K. Rothermel. TinyXXL: Language and runtime support for cross-layer interactions. In *Proceedings of the Third Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, pages 178–187, 2006.

[16] A. Lachenmann, P. Marrón, D. Minder, and K. Rothermel. Meeting lifetime goals with energy levels. In *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems*, 2007.

[17] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, April 2006. IEEE.

[18] R. Madan, S. Cui, S. Lall, and A. Goldsmith. Cross-layer design for lifetime maximization in interference-limited wireless sensor networks. In *IEEE INFOCOM*, March 2005.

[19] R. Musaloiu-E., C-J. M. Liang, and A. Terzis. Koala: Ultra-Low Power Data Retrieval in Wireless Sensor Networks. In *IPSN '08*, 2008.

[20] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. In *SenSys*, 2005.

[21] T. Voigt and F. Österlind. CoReDac: Collision-free command-response data collection. In *13th IEEE Conference on Emerging Technologies and Factory Automation*, Hamburg, Germany, September 2008.

[22] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proc. ACM MobiSys'04*, Boston, MA, USA, June 2004.

[23] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of ACM SenSys'03*, Los Angeles, California, USA, 2003.

[24] W. Ye, F. Silva, and J. Heidemann. Ultra-low duty cycle mac with scheduled channel polling. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 321–334, New York, NY, USA, 2006. ACM Press.

Recent licentiate theses from the Department of Information Technology

2011-003 Rebecka Janols: *Tailor the System or Tailor the User? How to Make Better Use of Electronic Patient Record Systems*

2011-002 Xin He: *Robust Preconditioning Methods for Algebraic Problems, Arising in Multi-Phase Flow Models*

2011-001 David Eklöv: *Efficient Methods for Application Performance Analysis*

2010-005 Mikael Laaksoharju: *Let Us Be Philosophers! Computerized Support for Ethical Decision Making*

2010-004 Kenneth Duru: *Perfectly Matched Layers for Second Order Wave Equations*

2010-003 Salman Zubair Toor: *Managing Applications and Data in Distributed Computing Infrastructures*

2010-002 Carl Nettelblad: *Using Markov Models and a Stochastic Lipschitz Condition for Genetic Analyses*

2010-001 Anna Nissen: *Absorbing Boundary Techniques for the Time-dependent Schrödinger Equation*

2009-005 Martin Kronbichler: *Numerical Methods for the Navier-Stokes Equations Applied to Turbulent Flow and to Multi-Phase Flow*

2009-004 Katharina Kormann: *Numerical Methods for Quantum Molecular Dynamics*

2009-003 Marta Lárusdóttir: *Listen to Your Users - The Effect of Usability Evaluation on Software Development Practice*

2009-002 Elina Eriksson: *Making Sense of Usability - Organizational Change and Sensemaking when Introducing User-Centred Systems Design in Public Authorities*

UPPSALA
UNIVERSITET

Department of Information Technology, Uppsala University, Sweden