Embedded WiSeNts

IST-004400

Research Integration: Platform Survey

Critical evaluation of platforms commonly used in embedded wisents research

EW Identifier: EW-T21/D01-SICS-001-01
Date: 2006-06-01

Author(s) and company: Can Basaran (YTU), Sebnem Baydere (YTU), Giancarlo Bongiovanni (CINI), Adam Dunkels (SICS), M. Onur Ergin (YTU), Laura Marie Feeney (SICS, editor), Isa Hacioglu (YTU), Vlado Handziski (TUB), Andreas Köpke (TUB), Maria Lijding (UT), Gaia Maselli (CINI), Nirvana Meratnia (UT), Chiara Petrioli (CINI), Silvia Santini (ETHZ), Lodewijk van Hoesel (UT), Thiemo Voigt (SICS), Andrea Zanella (DEI)

Work package/task: WP 2 / T 1
Document status: final
Confidentiality: public
Keywords: sensor network, research platform, hardware, operating system, tools, development environment, testbed

Abstract: Critical survey of research platforms for sensor networks, with an emphasis on hands-on experience.

DOCUMENT HISTORY

Version	Date	Reason of change
1	2006-01-23	document created
2	2006-06-01	task completed

Table of Contents

Hardware	Operating systems	Simulators
ESB/2	TinyOS 1.x, 2.0	Tossim
Tmote Sky	Contiki	Glomosim
BTnode	BTnut	Matlab
μNode	AmbientRT	Avrora
EYES		Omnet++
		Ns2

Table 1: Contents

1 Executive Summary

This report presents an in-depth critical survey of a number of advanced research platforms for wireless sensor networks. The goal of our work has been to evaluate these platforms with respect to their practical capabilities as research platforms, focusing on ease of use issues. For this purpose, we have limited the selection of platforms to those with which Wisent partners have substantial hands-on experience.

The information presented here is intended to inform researchers' selection of a platform and to increase awareness of the variety of available research environments. By contributing to the development diverse communities using these platforms, this report contributes to increased integration within the research community.

The content includes presentation of five hardware platforms and four operating systems. In addition, the document presents a number of "service distributions": software packages providing portable functionality for the communication stack, sensor data processing, and network management. A survey of six simulation environments is also presented. Finally, descriptions of four testbed and development environments show how these platforms are being used in practice.

The report closes with some brief discussion of implications of the survey.

This information is provided for guidance only. Although the authors have tried to ensure that the material was accurate at time of publication, no responsibility can be taken for the accuracy of the contents, particularly with regards to features and prices of commercially available systems. It should be emphasized that no information in this document is intended as an endorsement or recommendation of any particular platform by any of the authors, their respective institutions, or by any sponsoring organization.

2 Introduction

This report presents a critical survey of a number of advanced research platforms – hardware and software – for wireless sensor networks. The report evaluates these platforms with respect to their practical functionality as research platforms, focusing on issues related to using these platforms for research activities in various contexts. [1]

The primary contribution of this document are its focus on practical usage issues and its insight into hands-on experience working with these platforms – content which the authors are uniquely qualified to provide. In this regard, this approach may be contrasted with work such as [Beu05], which explores the feature space defined by various platforms. Although this report provides extensive information about the features available on each platform, we focus on the less easily quantified, but absolutely essential, efficiency of the design-download-debug cycle.

Because of the focus on practical experience, a theme that recurs frequently is this document is 'ease of use': ease of programming, debugging, and deploying *robust* applications. Ease of use is essential factor to the eventual success of any research platform; otherwise its use will limited to trivial scenarios.

A second, related, factor that is crucial to the development of an effective research platform is the development of a user community . Meeting the needs of a diverse group of researchers ensures that the platform provides complete and easy-to-use functionality. Moreover, the understanding of both shared and diverging requirements informs the design of effective abstraction layers and API's. Such collective experience is the essential technical basis for meaningful and useful standards and system integration practice.

The report is not intended to present technical concepts or to assess the research contribution associated with the development of the various platforms. (Such information is found in other Wisent project documents [BO06, ZO06, PO06, SO06], as well as in the literature.)

Nor is the report intended to determine a "best" research platform. There is surprisingly large variation among the platforms examined here, suggesting that there is a particularly broad design space. It is not possible, given the extraordinary range of both application requirements and of potential deployment environments, to indicate a unique platform that can comply with the requirements of such a broad variety of researchers. Different research goals require different functionality from the underlying platform. For example, a researcher developing new MAC layer requires a platform that allows her to access and modify a platform's lowest layer communication functionality. By contrast, a researcher developing query management mechanisms may be better served by a platform that provides a clear high-layer communication API.

The authors hope that making information about various platforms available in an accessible format will assist researchers working in this area to appreciate the diversity of platforms available to them and to help them select the most appropriate platform for their purposes.

[1]The authors emphasize that this information is provided for guidance only. Although the authors have tried to ensure that the material was accurate at time of publication, no responsibility can be taken for the accuracy of the contents, particularly with regards to features and prices of commercially provided systems. It should be emphasized that no information in this document is intended as an endorsement or recommendation of any particular platform by any of the authors, their respective institutions, or by any funding organization.

2.1 Content and scope

The report provides both a catalog of features and, more importantly, the authors' collective experience in using these systems as research platforms. Each section is a combination of both tabulated data and discussion. It should be noted that due to the significant differences among the various platforms, a uniform side-by-side comparison is not possible. A limited summary comparison is provided, however.

The report covers five key areas:

- sensor hardware platforms
- operating systems
- service software distributions
- simulation and emulation environments
- testbeds

In general, each section contains a table summarizing the platform features (as specified by manufacturer), an (objective) structured discussion of the platform characteristics, and a (subjective) open discussion of the user experience with that platform.

Because of its emphasis on platforms with which the authors have hands-on experience, the report focuses on a relatively small number of platforms. The main hardware/OS platforms addressed in this report are:

- BTnodes (NutOS and TinyOS)
- ESB/2 nodes (Contiki and TinyOS)
- μnodes (AmbientRT and TinyOS)
- SmartTags (no OS)
- EYES node (TinyOS)
- TmoteSky (Contiki and TinyOS)

In addition to the underlying platforms, the report also presents some widely used software service distributions that operate at higher layers or are relatively independent of a particular platform are also presented. These include communication protocols, sensor querying services and management tools:

- LMAC MAC layer
- uIP TCP/IP stack (multiple platforms)
- query mechanisms such as TinyDB, Cougar, and Acquire
- publish/subscribe abstractions

- management mechanisms for reprogramming a network

The practical complexities of building sensor networks make simulation and emulation techniques are important research tools. Because such environments have only limited fidelity, it is important to understand the limitations of such tools.

Several simulation/emulation environments are presented, including:

- Tossim
- Glomosim
- Matlab
- Avrora
- Omnet++
- ns-2

3 WSN platforms

The first topic of investigation is the sensor hardware platform.

A brief overview table broadly comparing various platforms is presented first. It should be emphasized that this table presents only a broad comparison, as fields do not necessarily represent identical features. The description and discussion of each platform is more important. The structured description of each platform is followed by a subjective discussion of experiences using that platform. The selected platforms are:

- BTnodes (NutOS and TinyOS)
- ESB/2 nodes (Contiki and TinyOS)
- μnodes (AmbientRT and TinyOS)
- SmartTags (no OS)
- EYES node (TinyOS)
- TmoteSky (Contiki and TinyOS)

All of the above systems are intended as general purpose platforms to which a variety of sensor modules can be attached [2]. They can therefore be used as a basis for developing higher layer software and service distributions, as well as to build testbeds and specialized applications.

The basic features listed below were assessed as essential by the authors. The basic features under consideration are:

- Introduction/Overview
- MCU
- Radio Transciever(s)
- Storage
- Sensors
- Energy Storage
- External Interfaces
- Packaging
- Availability
- Support

This objective presentation of the features and functionality available in each H/W platform is followed by a discussion of the users' subjective experience with the platform.

[2] Some platforms, especially the ESB/2, have a number of simple on-board sensors.

company URL	name	price (kit)	price (node)	price (gateway)	order online
ScatterWeb GmbH www.scatterweb.com	ESB	∼970 € 4 ESB + gateway	∼89–149 €	∼129–199 €	yes
Infineon www.infineon.com	EyesIFX	∼300 € 5 EyesIFXv2.0	∼70 €	–	No
 www.art-of-technology.ch	BTNode	∼520€ 2 BTnode + s/w	∼165€	–	–
Ambient Systems BV www.ambient-systems.net	μNode	–	∼80€	–	–
MoteIV www.moteiv.com	TmoteSky	∼790$ 10 motes only	∼130$	∼150$	yes

Table 2: Cost and availability for hardware (not including sensors)

3.1 Overview and comparison

The cost data in table 2 is intended only as broad guidance and tends to suggest that there is a common price point roughly in the 100€/unit range, making an experimental set up relatively affordable. In particular, we note that the cost of time and effort establishing an experimental set up is a significant factor in the total cost, and emphasizes the importance of ease of use issues. (It is also worth commenting that, in real scenarios, the cost of some application-specific sensors may well outweigh the cost of the hardware platform itself.)

Table 2 indicates prices and availability for various hardware platforms. [3] In most cases, the hardware platform is provided without sensors. Some ESB hardware models include a variety of simple built-in sensors, which is particularly convenient for educational systems. Sensor kits, including with basic sensors are available for all of the various platforms and usually cost in the range ∼50€- ∼150€. Many manufacturers sell a "starter" kit. This usually includes several nodes, gateway nodes and/or sensors, various connectors and cables, software and documentation.

The overview comparison, table 3 below, provides a indication of the general breadth of hardware platforms (for a broader investigation of the design space, see e.g. [Beu05]). Unless there is a requirement for a particular hardware feature, however, the development and debugging environment is more likely to dominate the selection of a platform.

3.2 ESB/2

The ESB (Embedded Sensor Board) is an MSP430x149-based wireless sensor network prototype board that is equipped with a TR1001 radio transceiver and set of on-board sensors: a passive IR (PIR) sensor for motion detection, an IR receiver that can be used to control an ESB with a regular remote control, an IR transmitter, a temperature sensor, a tilt/vibration sensor, and a sound sensor. There are a number of external connectors, including an RS232 serial connector and a JTAG connector. The RS232 line can be used to communicate between an ESB and a PC and the JTAG connector is used for reprogramming the MSP430 flash ROM. Additionally, there are a number of processor lines that are connected to external

[3] Disclaimer: This information is provided for guidance only.

platform	processor	speed	memory	transceiver	kbps
ESB/2	MSP430	1 MHz	2 KB + 60 KB	TR 1001	115.2 kbps
TmoteSky	MSP430	1 MHz	10 KB + 48 KB	CC 2420 (IEEE 802.15.4)	250 kbps
BTnode	ATmega	8 MHz	4 KB + 128 KB	CC 1000 *and* Bluetooth	76.8kbps kbps
μNode	MSP430	1 MHz	10 KB + 48 KB	–	50 kbps
EYES	MSP 430	1 MHz	10 KB + 48 KB	TDA5250	64 kbps

Table 3: Overview hardware comparison (where applicable, value shown is the maximum of possibly several choices (e.g. processor speed).

connectors that can be used to extend the functionality of the ESB by, e.g., new sensors or communication chips.

The ESB was originally was developed at the Free University of Berlin but its continued development is carried out by the spin-off company Scatterweb [Sca]. Many of the original developers from Free University of Berlin are now working at Scatterweb.

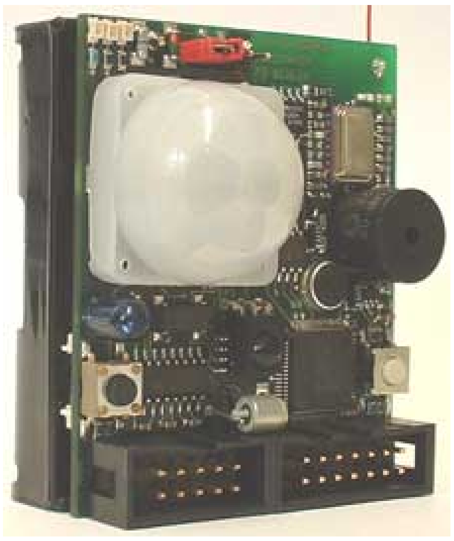


Figure 1: ESB/2 node

3.2.1 MCU

The ESB is equipped with a MSP430 microcontroller.

Processor: TI MSP430F149

speed	**features**
up to 1MHz	

Memory			
RAM	**ROM**	**cache**	**other**
2 KB	60 KB	n/a	32 KB EEPROM

Power modes	**sleep**	**MCU**	**sensor**	**all**
4 modes	8 μA	280 μA	–	12 mA

Power

input	**battery**	**capacity**	**external power**
3 V	3 AAA	XXX	XXX

charge time	**lifetime**	**other**
XXX	XXX	capacitor

Tranceiver(s): TR1001

PHY	**MAC**	**freq**	**antenna**(s)
ASK,OOK	n/a	868MHz	cable

power control	**range**	**bitrate**(s)
100 levels	up to 300m outdoor up to 100m indoor	up to 115.2 kbps XXX

power consumption			
sleep	**idle**	**xmit**	**rcv**
5 μA	XXX	7-8 mA	7-8 mA

Sensors

builtin	PIR, infrared, temp., tilt, mic
available	I/Os available
interfaces	RS232, JTAG

Table 4: At a glance: Platform ESB2

External interfaces

Programming	JTAG
Debugging	JTAG
Data	RS 232
ADC/DAC interfaces	n/a

Form factor

dimensions	packaging	environmental issues
60x50x40mm	standard	XXX

Support and community

commercial	manual	tutorials	other
yes	yes	no	mails answered quickly

users	web	mailing list	user forum
100 (est)	yes	yes	no

Table 4: At a glance: Platform ESB2 (con't)

Features The flash ROM can be rewritten either by using the JTAG connector, or by a program running on the MSP430.

Memory The on-board MSP430 microcontroller is equipped with 2k RAM and 60k flash ROM.

3.2.2 Radio Transceiver

The ESB is equipped with a TR1001 transceiver that is connected to on of the UARTs of the MSP430. Programs running on the MSP430 read and write individual bytes to the radio transceiver via the UARTs. Bytes written through the UART are immediately sent over the radio.

Because of the low-level interface of the transceiver, the entire radio protocol must be implemented in software running on the microcontroller. This has several implications. First, since every developer may implement their own radio protocol, different ESB implementations may not be able to communicate with each other despite them being equipped with the same radio transceiver. Second, the complexity of the code required to implement a low-level radio protocol is high which leads to code which may contain several bugs and that is difficult to debug.

The TR1001 can send data in one of two modes: Amplitude Shift Keying (ASK) and On-Off Keying (OOK). With OOK, a digital one is encoded as a high output transmit power and a digital zero is encoded as zero output transmit power. In contrast, in ASK mode a digital zero is encoded as a low output transmit power.

The range of the radio is determined by both external factors (the radio transmission environment) and internal factors (the speed of the UART and the encoding used by the radio protocol implemented in software). It is possible to do error-free transmissions that reach several hundred meters outdoors, at free-sight environments. The indoor range is significantly lower, and is typically shorter than a hundred meters.

The TR1001 measures the received signal strength and provides it on one of the output pins of the TR1001. On the ESB, this is connected to one of the A/D converters which makes it possible to obtain the current received signal strength using software.

3.2.3 Storage

The ESB has a 32 kilobyte large serial EEPROM that can be read from and written to by programs running on the microcontroller.

3.2.4 Memory

The on-board MSP430 microcontroller is equipped with 2k RAM and 60k flash ROM.

3.2.5 Sensors

PIR sensor The Passive IR (PIR) sensor is used to detect human movement in front of the sensor. It has a range of a few meters and a detection angle of 60 degrees.

When movement is detected, a digital signal is transmitted to the microcontroller. While it is possible to steer the output of the PIR sensor to an analog input of the microcontroller, the output from the PIR sensor is strictly digital and it is not possible to obtain any additional information regarding e.g. the intensity of the movement from the analog output. Instead, the number of digital indications of movement per time unit can be used to assess the amount of movement in front of the sensor.

IP receiver While the chief purpose of the IR receiver is to receive IR communication from either another ESB (through the use of the on-board IR transmitted) or from a standard remote control, the IR receiver can also be used to detect visible light. The information obtained from this can be used to infer if the ESB is located indoors, where the ambient light flickers with a frequency of 50 Hz, or outdoors, where the ambient sunlight does not flicker.

Tilt/vibration sensor The tilt/vibration sensor can detect a movement of the ESB. It transmits a digital signal to the microcontroller when vibrations are detected. The amplitude of the movement can be inferred by the number of signals received over a period of time.

Sound sensor The sound sensor is connected to an analog input of the microcontroller through which it is possible to sample nearby sounds. While the sound sensor is good enough to record speech, its main use is to detect other types of noise such as loud clicks or explosions.

Temperature sensor The temperature sensor is integrated with the MSP430 microcontroller.

Extensibility There are a number of microcontroller input lines routed to external connectors on the ESB/2. These can be used to add additional sensors or for adding new communication chips.

3.2.6 Energy Storage

Battery The ESB is equipped with a battery pack containing three AAA (1.5 V) batteries.

External Power-Supply Interfaces The ESB provides a connector for external power-supply that can be used to power the ESB with, e.g., an external mains-connected voltage generator, a solar panel, or a high-power capacitor [RSV$^+$05].

3.2.7 External Interfaces

Debug/Programming The ESB has an on-board JTAG connector that is used for reprogramming the MSP430 on-chip flash ROM and for debugging software running on the microcontroller. The JTAG connector typically is connected to the parallel port of a PC on which the reprogramming and debugging software is run.

Data Buses An RS232 connector is connected to one of the MSP430 UARTs. This is the primary way to communicate between an ESB an a PC; an RS232 serial link is connected to the RS232 interface of the ESB and a serial port of the PC.

3.2.8 Packaging

Dimensions The dimensions of the ESB, including the battery pack, is 60x50x40 mm.

Materials The ESB appears to be made out of standard electronics materials.

Environmental Safety Issues The status of any environmental issues is unknown.

3.2.9 Availability

Developer/Manufacturer The ESBs are available from Scatterweb [Sca].

Cost Each ESB device costs approximately 150 Euro.

3.2.10 Support

Documentation All connections between the microcontroller and the sensors are documented by Scatterweb. Documentation for the microcontroller, sensors, and the radio transceiver are available from their respective manufacturers.

Community A mailing-list for users of the Scatterweb software exists. The list has a few mails per month, but questions are promptly answered from Scatterweb developers.

3.2.11 Experiences with the ESB platform

General. There are several operating systems for the ESB. Scatterweb themselves have a while-loop OS from FU Berlin, and the ESB are one of the main platforms for SICS Contiki operating system. FU Berlin have also ported TinyOS to the ESB nodes.

Processing and Memory. The 2 kilobyte RAM of the ESB can often be very challenging and the developer has to keep track of the memory used by the applications to make sure that it does not over-utilize the ESB RAM.

Sensing/Actuating Capabilities. At SICS, we have used most of the sensors, in particular the vibration and PIR sensors are very useful for intrusion detection sensor network settings.

Communication. The radio has quite a long range; 200-300 meters outside in temperatures below zero and more than 50 meters inside. However, the packet loss/bit error rates with the simple TR 1001 radio are quite high.

Energy. Besides battery, the ESB nodes can be powered by solar cells or high-capacity capacitors. The batteries are three standard AAA batteries. While we in the beginning when we had only a few nodes used rechargable accumulators, we are now using batteries since it avoids the hassle of keeping track of the accumulator's status. Further, the batteries power the ESB nodes for a reasonably long time, usually several months for the ones used quite often (when the radio is turned off most of the time). High-capacity capacitors such as GoldCaps are useful for comparing the energy-efficiency of e.g. communication protocols, since they power the nodes for a short time only, about one hour with a non-negligible duty cycle.

Platform specific development experience. Since the ESB has an on-board JTAG connector, reprogramming the flash ROM is straightforward: the ESB is connected to the parallel port of a PC with a special cable and the flash is rewritten. When using the Contiki OS, it is also possible to reprogram any number of ESB boards remotely over the radio.

We have found that some the ESB boards can physically fall apart after long time handling. Usually the plastic protector lens of PIR detector falls off from the board after a while. That said, we also have experience with ESB boards quite robust. During one field trial where we deployed ESB boards throughout a military training area, several ESB boards were deployed in a room where they were exposed to the blasts of a number of training hand grenade explosions. The ESB boards, running Contiki, were able to detect the explosions using the PIR and vibration sensors and to transmit the information back to the base station placed in the basement of the building in which the military exercise took place. After the exercise all ESB boards were found to be intact.

Support. There is a mailing list for the ESB board where questions typically are answered within a day. Scatterweb provides several support and management tools.

3.3 Tmote Sky

The Tmote Sky is a general purpose WSN platform with a large market share both in academia and industry. It is the successor of the popular TelosA and TelosB research platforms from UC Berkeley. Tmote Sky is one of the few FCC Certified WSN platforms available on the market, making it particularly suitable for OEM integration. Thanks to the ultra-low-power microcontroller and the electrically buffered external components that can be switched-off when not in use, the platform achieves very low quiescent currents in the sleep mode, and very fast wake-up times.

3.3.1 MCU

The Tmote Sky is equipped with a MSP430F1611 microcontroller from the TI MSP430 family of ultra-low-power 16-bit microcontrollers. The MSP430 family comes with a AD converters, ports, uart, SPI, I2C buses. On the Tmote much of this can easily be connected to external devices.

Memory The on-board microcontroller offers 10k byte RAM and 48k byte flash programmable ROM. The board also comes with 1M byte of external flash memory connected to the SPI bus.

3.3.2 Radio Transceiver

The Tmote Sky is equipped with CC2420, an IEEE 802.15.4 compliant radio transceiver from Chipcon/Texas Instruments. The transceiver is connected to an internal inverted-F antenna giving it about 50 m of useful communication range when used indoors, and upwards of 125 m outdoors. The maximum effective RF output power of the platform is about -7 dBm.

3.3.3 External Storage

The Tmote Sky has a 1 MB serial flash, M25P80, that can be read and written by the MSP430. Tmote Sky provides hardware based write protection to parts of the flash that is disabled only when the module is powered via the USB interface (see below). This enables storing a protected factory image which can not be erased/modified during normal node operations. The Tmote Sky arrives with one such "Golden Image" in sector 15 of the flash containing support for network reprogramming using the TinyOS Deluge protocol.

3.3.4 Sensors

In addition to the internal temperature sensor of the MSP430 microcontroller, the Tmote Sky board has predefined positions for mounting a humidity/temperature sensor from Sensirion AG (models SHT11 and SHT15 are supported), as well as for light sensors like the Hamamatsu Corporation S1087 for sensing photosensitive solar radiation or the S1087-01 for total spectrum measurements.

3.3.5 Energy Storage

Battery The Tmote Sky is powered by two AA size (1.5 V) batteries affixed in a standard battery pack. The operating range of the platform when on battery power is from 2.1 to 3.6 V, with 2.7 V needed for

Processor: TI MSP430F149

speed	**features**
up to 1MHz	

Memory			
RAM	**ROM**	**cache**	**other**
10 KB	48 KB	n/a	1M ext. flash

Power modes				
	standby	**idle**	**MCU**	**all**
–	5.1 μA	54.5 μA	1.8-2.4 mA	–

Power

input	**battery**	**capacity**	**external power**
21.-3.6 V	2 AA	XXX	XXX

charge time	**lifetime**	**other**
XXX	XXX	USB power

Tranceiver(s): CC2420

PHY	**MAC**	**freq**	**antenna**(s)
IEEE 802.15.4		2.4 GHz	internal inv-F

power control	**range**	**bitrate**(s)
	$>$ 125m outdoor $\sim$ 50 indoor	250kbps 250kbps

power consumption			
sleep	**idle**	**xmit**	**rcv**
20 μA	365 μA	17.4 mA	19.7 mA

Sensors

builtin	
available	light, temperature/humidty
interfaces	USB serial, upto 6 ADC channels, up to 6 general i/o

Table 5: At a glance: Platform Tmote Sky

External interfaces

Programming	USB, JTAG
Debugging	JTAG
Data	10 pin and 6 pin expansion connector, I2C
ADC/DAC interfaces	up to 6 ADC channel

Form factor

dimensions	**packaging**	**environmental issues**
32x66x7mm	standard	n/a

Support and community

commercial	**manual**	**tutorials**	**other**
yes	yes	no	–

users	**web**	**mailing list**	**user forum**
largest community	yes	yes	Wiki-based

Table 5: At a glance: Platform Tmote Sky (con't)

internal and external flash reprogramming.

External Power-Supply Interfaces The Tmote Sky can also be powered via the on-board USB interface when plugged into the USB port of a host computer for programming or communication. The operating voltage when attached to the USB is 3 V.

3.3.6 External Interfaces

Data Buses The Tmote Sky uses a FT232BM usb-to-serial chip from FTDI as a primary communication channel with the host controller. The USB interface is connected to the USART1 module on the microcontroller. On the PC side, the USB connection appears as a regular serial port. In addition to the USB serial channel, the expansion connector on the Tmote Sky also exports the raw UART0 receive and transmit lines and one I2C and/or SPI bus.

Debug/Programming The same USB interface is also used to (re)program the microcontroller flash. Tmote Sky exports the microcontroller JTAG pins for in-circuit debugging and reprogramming via an additional interface. The microcontroller flash can also reprogram itself from software (with some OS support).

ADC Inputs The 10-pin extension connector provides access to four ADC channels. Additional two channels are exported via the 6-pin "exclusive" interface.

Digital I/O and Interrupts The extension connector provides four pins that can be used as general I/O (when not used as ADC/I2C functional pins). Two additional general IO pins are available on the "exclusive" interface. This interface also exports the two DAC channels of the microcontroller as well as the interrupt lines for the user interface elements == the reset and the user buttons.

3.3.7 Packaging

Dimensions Tmote Sky has the following nominal dimensions: width – 3.2 cm, length – 6,55 cm and height [4] – 0.66 cm.

3.3.8 Availability

Developer/Manufacturer The Tmote Sky boards are produced by MoteIV Corporation, San Franciso, CA. [Mot].

Cost The units can be obtained directly from the manufacturer via a convenient web ordering service. Purchased individually, a single Tmote Sky unit (without the on-board sensors) costs about $130. The humidity, temperature and light sensors cost additional $50. A 10-unit Tmote Sky Developer Kit can be obtained for $790, driving the single unit price down to $79.

3.3.9 Support

The main support for the Tmote Sky platform is provided via the manufacturer web site and via e-mail. Software related issues are also handled via the TinyOS mailing lists since many of the Tmote Sky users are also using TinyOS.

Software Tmote Sky is fully compatible with TinyOS 1.x and TinyOS 2.0 and MoteIV originally distributed a cygwin based software installer providing full TinyOS development environment for Windows. Recently, they have developed a hardened TinyOS distribution called Boomerang, specifically optimized for the MoteIV product family.

More recently the Tmote has also become one of the standard platforms for the Contiki operating system.

Documentation When delivered to the customers, the Tmote Sky units are accompanied with a "Quick Start Guide" and a detailed "Data Sheet" documents. New versions of the documentation can be downloaded from the MoteIV web site.

Community The MoteIV web-site features a community-based Wiki section [5] which is the central repository of information for the Tmote Sky user community.

[4] without battery pack and SMA antenna

[5] `http://www.moteiv.com/community/Moteiv_Community`

3.3.10 Experiences with the TMote sky platform

General. Tmote sky nodes are, as the ESB2 nodes, equipped with a TI MSP 430 microcontroller, however with 10 KB RAM (five times larger than on the ESB nodes) which allows developers to pay less attention to RAM usage. The nodes themselves are robust and do not break easily.

Processing and Memory. As mentioned above, the Tmote sky nodes feature enough memory, more than many other platforms in particular RAM.

Sensing/Actuating Capabilities. In contrast to e.g. the ESB nodes, Tmote sky nodes feature only three sensors on their board, namely light, temperature and humidity. These sensors are not very exciting for demo purposes.

Communication. One advantage of the nodes is the use of USB for serial communication (USB is on the way to replace the traditional serial interfaces).

Having a standard compliant 802.15.4 radio is an attractive feature which is appreciated by industry. The radio is also very flexible and provides a number of features such as automatic CRCs, automatic ACKs, MAC address filtering, security features built on the AES chiper, etc. An additional advantage of the radio is its very low power consumption when in idle mode. However, the integrated on-board antenna does not seem very powerful.

Energy. The nodes can also be powered via USB which is very convenient when experimenting in labs since nodes do not behave strangely as they do when batteries are depleting.

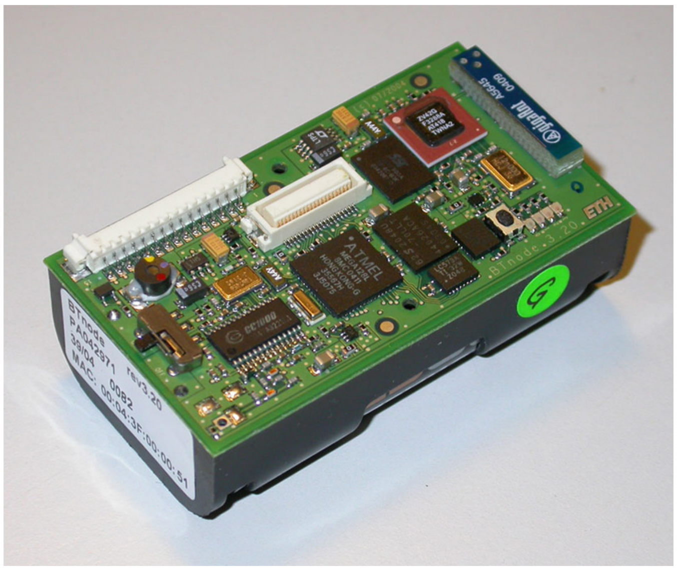


Figure 2: BTnode rev3

3.4 BTnode

3.4.1 Introduction

The BTnode [BTna] is an autonomous wireless communication and computing platform based on a Bluetooth radio and a microcontroller. It has been developed as a demonstration platform for research in mobile and ad-hoc connected networks and distributed sensor networks. The BTnode has been jointly developed at the ETH Zurich (Swiss Federal Institute of Technology in Zurich) by the Research Group for Distributed Systems, and the Computer Engineering and Networks Laboratory. Its development had been primarily supported by the NCCR-MICS [NCC] and the Smart-Its [Sma] research projects, the latter being a part of the European initiative *The Disappearing Computer*, and funded by both the Commission of the European Union and the Swiss Federal Office for Education and Science.

There have been three major hardware revisions of the BTnode hardware platform: BTnode rev 1, BTnode rev2 and BTnode rev3. The BTnode rev3 is now successfully used in severa lresearch projects spanning from rather simple applications with few nodes to large, interactive networking applications, some of them supported again by the NCCR-MICS [NCC] Swiss-founded project.

As a software system, the BTnode can run both the BTnut, and the TinyOS operating systems. Details on software support issues are provided in section 4.

Figure 3 provides an overview of the BTnode hardware architecture, whose main components are detailed in the subsections below and summarized in table 6[6].

[6]Table 6 shows the BTnode rev 3 power consumption as reported in [BTna]. All values are nominal values measured on a live system at $3.3V$.

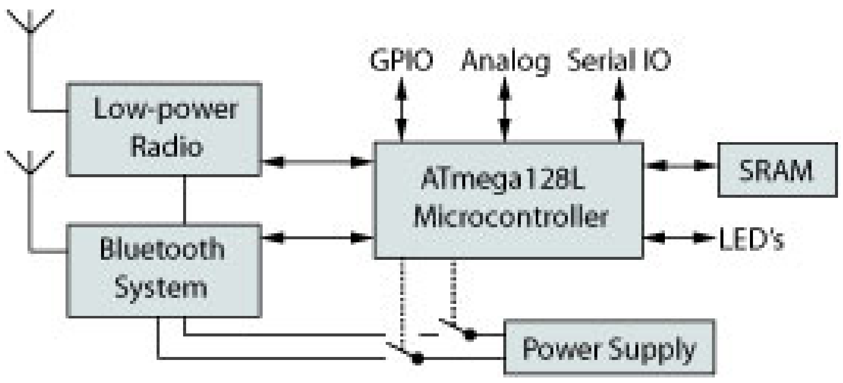


Figure 3: BTnode Hardware Architecture [BTna]

3.4.2 Microcontroller

The BTnode platform is equipped with an Atmel ATmega128L microcontroller. It features an 8-bit RISC core delivering up to 8 MIPS at a maximum of 8 MHz. The Atmel ATmega128L is provided with 128kbytes of in-system programmable Flash memory, 4 kbytes on-chip static RAM and 4 kbyte EEPROM. Since the Atmel ATmega128L can address up to 64 kbytes main memory, the 4 kbytes of internal SRAM are extended to 64 kbytes through an external, additional (60 kbytes) memory module.

The microcontroller features a 32 kHz real time clock and 7.3728 MHz system clock.

3.4.3 Radio Transceivers

The BTnode platform features two independent communication modules: a Bluetooth radio and a low-power radio. Both radios can be operated simultaneously or be powered off when not used.

- **Bluetooth radio** - The BTnode rev3 features a Zeevo ZV4002 Bluetooth system, supporting a Scatternet with up to 4 independent Piconets (each Piconets with 7 slaves). The Bluetooth module is connected to the Atmel ATmega128L through a UART interface.

- **Low-power radio** - The Chipcon CC1000 is the secondary BTnode on-chip radio. It operates at 868 MHz, but other operating frequencies within the ISM band 433-915 MHz can also be used. An integrated (PCB printed) monopole antenna is the default assembly option. Additionally, both an external wire and an external coaxial connector (MMCX type) can be used.

3.4.4 Storage

As mentioned earlier in this section, the BTnode on-chip memory consists of 128 kbytes of in-system programmable Flash memory, 4 kbytes EEPROM, and 4 kbytes SRAM. The on-chip SRAM is extended to 64 kbytes through an external 256 kbytes memory module. 180 kbytes out of the remaining 196 of the external SRAM are provided as three data cache memory banks, of 60 kbytes each. The remaining 16 kbytes are unused.

Basically, the flash ROM is used for storing programs and constant data, the RAM is used for allocating heap, global variables and stack, while the (power expensive) EEPROM is used to store persistent configuration data. Even though the Flash ROM has theoretically unlimited read capability, the number of write/erase cycles is limited. The flash ROM in the Atmel ATmega128L microcontroller is guaranteed to overcome at least 1000 of these cycles, in the reality this number is typically much higher. Since the content of the EEPROM is not erased when power supply is removed, and since write and read access are typically very slow, this memory is used for storing system configuration data, which must be read once at system startup and is typically seldom modified.

3.4.5 Sensors and Actuators

Sensing capabilities on the BTnode platform can also be easily obtained by connecting single sensors or a sensor board to the appropriate lines of the J1 extension or J2 debug connectors (see also section 3.4.7). Available sensor boards are the *ssmall* sensor board from Particle Computers and the *BTsense*.

ssmall Sensor Board. The *ssmall* sensor board is available for purchase from Particle Computer GmbH[7]. There are two versions of this sensor boards, the *Medium* and the *Full*. The *ssmall Medium* sensor board is equipped with the following sensors:

- TSL2500 daylight and IR light sensor, manufactured by TAOS (Texas Advanced Optoelectronic Solutions)
- TC74 temperature sensor (typical accuracy: $\pm 0.5^oC$)
- 2 LEDs (can be replaced by, e.g., vibration motor)
- MAX8261 OP capacitive microphone (high precision, high linearity)
- ADXL210 2-axis acceleration sensor, manufactured by Analog Devices (10g max, ± 40 mg resolution, responsiveness $< 1ms$)

It is also prepared for hosting pressure sensor, if needed. It does not feature its own processor and receives power supply either through the main board (the BTnode) or through a separate 1-3.3V power supply. It is 17x22 cm in size and can be connected to the BTnode through the USP programming board and the J1 connector.

The *ssmall Full* sensor board is identical to the *Medium* but features an additional

- 3 axis acceleration sensor (composition of 2 ADXL210)

BTsense. The BTsense is a 2x4 cm sensor board that can be easily affixed to the side of the BTnode. This board has been designed and developed at the Institute for Pervasive Computing of the ETH Zurich[8] and it has been successfully used in several student projects and teaching. It features:

[7]Particle Computer GmbH (www.particle-computer.net)

[8]By Matthias Ringwald and Jonas Wolf, with contribution by Benedikt Ostermaier and Marc Langheinrich

- TC74 temperature sensor (digital, I2C)
- TSL252R light sensor (analog)
- AMN1 passive infrared motion sensor, (digital, logic level)
- 7BB-12-9 piezo buzzer

Additional I2C digital sensors as well as one external analog sensor can be added to the board. This sensor board is currently used for teaching and research and not (yet) available for commercial purchase.

Actuators can also be easily connected to the BTnode platform through the extension connector J1. For debugging purpose, 4 LEDs are embedded on the BTnode circuit board.

3.4.6 Power Supply

The standard power supply are 2-cell AA batteries. The common range for these is $2.0 - 3.0$V DC when either primary (i.e., not rechargeable)or secondary (i.e., rechargeable) batteries are used. Alternatively $3.8 - 5.0$V can be supplied through the VDC_IN pin on the external connectors J1 (pins 17 and 18) and J2 (pin 15).

All BTnode electronics components are DC-powered at 3.3 V. To ensure stability of operation, a constant 3.3V voltage level is thus provided by adequate step-up/step-down converters. When powering the BTnode with batteries, the primary step-up (boost) converter can readjust every input voltage in the range $0.5-3.3$ V to a stable, 3.3V supply voltage. An analogous step-down (buck) converter, can regulate the $3.8-5.0$V voltage eventually supplied through the VDC_IN external pin.

To upload program code to the BTnode through a USB port, a dedicated programming board must be attached to the node. While connected to the USB programming board, the BTnode is powered by the USB connector, and does not need additional power supply.

A summary of the power requirements of the BTnode under various operational conditions is reported in table 6.

3.4.7 External Interfaces

Several external interfaces are available on the BTnode platform: UART/USART, SPI, I^2C, GPIO, ADC.

To physically connect to a BTnode, either the extension connector J1, or the debug connector J2 can be used. Both types are available at Farnell or Digikey in small quantities (Molex 1.25mm Wire-to-Board and Hirose DF17 Board-to-Board connectors)

3.4.8 Packaging

The BTnode has a very small form factor (6cm x 4 cm), which perfectly fits the platform battery case. The BTnode is completely lead-free. The BTnode is a very robust platform and mechanical destruction originates in most cases from users' improper handling. However, electrostatic discharge (ESD) and wrongful operation can harm BTnodes and make the hardware partially or completely unusable. Useful "Tipps & Tricks" for a safe BTnode handling are available from the project web site [BTna].

It is also worth to mention that since all electronics units are situated on one single side of the BTnode circuit board, the board itself is solidly affixed to the battery case.

3.4.9 Availability

The BTnode rev 3 platform, as well as related hardware and software products[9] are available for purchase through a contract (Swiss) manufacturer[10]. The pricing for a single BTnode sample is around $165e$, while a complete developer kit containing 2 BTnodes, the necessary hardware and software tools for in-system programming, and a CD with the BTnut software suite and related tools, has a pricing of about 520 €.

The *ssmall* sensor board is available for purchase from Particle Computer GmbH[11]. The price of the Medium board is about 80 €, while the Full can be purchased for 100 €. Samples of the BTsense board can be obtained by contacting the Institute for Pervasive Computing (Research Group for Distributed Systems) at ETH Zurich[12].

3.4.10 Support

The BTnode is a well-established prototyping platform and is used in more than 30 research projects. The BTnode website [BTna] is a rich source of useful information and technical documentation. The active mailing list additionally offers the possiblity to pose questions and problems and get assistance from expert BTnode users.

3.4.11 Experiences with the BTnode platform

General. The BTnode is a very robust hardware platform, that has been widely used in both research and teaching. As we will detail in section 4.4.6, once you purchased the hardware, getting your first application running on the Btnode will take at most few hours. The software development tools can be easily installed on top of the Windows, Linux or on Mac OS operating systems and detailed installation guides and tutorials are available from the BTnode website [BTna]. The BTnode can be programmed in standard C language, when the BTnut system software is installed on top of it, or in NesC, when the TinyOS operating system is used.

Due to its high flexibility, the BTnode can be used for accomplishing with a variety of different research tasks, varying from the implementation and testing of new MAC protocols [RR05a], to the development of network monitoring and debugging tools [RR05b, RYR06]. The BTnode has also been successfully and widely used as a prototyping platform for ubiquitous applications[KL01, Sma].

In September 2003 about 15 BTnode users and developers took part in a survey aimed at understanding what people mostly like/dislike about the platform. This study [BTnb], as well as subsequent experiences, proved the BTnode to be a suitable platform for supporting both wireless sensor networks research and application prototyping.

Processing and Memory. The processing capabilities and the available storage on the BTnode are reported to be adequate for most of the tasks undertaken.

[9]See section 4.4 for details about BTnode-related software products
[10]www.art-of-technology.ch
[11]Particle Computer GmbH (www.particle-computer.net)
[12]http://www.vs.inf.ethz.ch/

Sensing/Actuating Capabilities. The BTnode can be equipped with a variety of different sensors (as detailed in section 3.4.5) and is thus adequate for prototyping a variety of sensor-based, ubiquitous applications. A major drawback is the lack of on-chip sensors on the BTnode circuit board: attaching external sensors causes the hardware to be less robust and prone to faults. However, the high flexibility of the BTnode platform allows users to add a variety of different sensors, thus enabling a easy customization of the platform. When adding new sensors to the BTnode, however, the correspondent custom sensor drivers must be implemented by the user.

Communication. Many users particularly like the fact the BTnode is able to easily interact (through the Bluetooth radio) with other devices like mobile phones or PDAs. Moreover, the existence of two independent radio modules offers a mean for unobtrusive debugging and monitoring, since, e.g., the Bluetooth radio can be used as a monitoring backchannel while the Chipcon radio operates as the standard communication channel between nodes [RYR06].

It has recently been observed that having both the Chipcon and the Bluetooth radio active, the measured bit error rate for the Chipcon radio is higher than when the Bluetooth module is off. BTnode developers are currently working on this problem and it seems that shielding the Chipcon module would help in avoiding this kind of interferences between the two on-board radios.

The Bluetooth radio has been widely tested and used in a real testbed with more than 70 nodes and it reaches a communication range of about 20 to 30 meters in indoor environments.

Several users complain about the unpredictability of the gain of the on-board antenna for the Chipcon radio, which implies an unpredictability of the achievable communication range. Therefore, it is recommended to equip the BTnode with an external antenna, which would allow to achieve more than 100 meters (and up to 300 meters) communication range in outdoor environments. To add an external antenna, it is sufficient to solder a wire on a (therefor provided) connector on the BTnode board.

Energy. The Bluetooth module, even if often indicated as the most appreciated feature of the BTnode platform, also has the drawback of the (for wireless sensor networks requirements) relatively high power consumption. Recent studies have however shown, that using adequate power management policies allows for a significant reduction in terms of energy consumption of the Bluetooth radio [NT06].

Equally problematic is the relatively high power consumption occurring when the BTnode operates in idle mode. Therefore, BTnode developers recently developed a software driver for power management allowing to turn down the idle power consumption to about 0.5 mW. The driver, currently under evaluation, will be soon publicly available.

Support. For any kind of hardware and software problems, the BTnode users can always refer to the small but active BTnode community, or to the well-maintained BTnode "Tips and Tricks" website[13].

[13] http://www.btnode.ethz.ch/Documentation/TipsAndTricks

Processor: Atmel ATmega128L

Architecture			
8-bit RISC			

Speed			
8 MHz			

Memory			
RAM	**ROM**	**EEPROM**	**cache**
4 (+60) kbytes SRAM	128 kbytes Flash	4 kbytes	3x60 kbytes

Power Consumption

	BTnode rev3 Bluetooth	BTnode rev3 Low Power Radio
Battery Supply	2 AA cells	
Minimum Vin	0.85 V	
Battery Capacity	2900 mAh	
Regulated Supply	yes	
CPU sleep, Radio off	9.9 mW	9.9 mW
CPU on, Radio off	39.6 mW	39.6 mW

Tranceiver(s): CC1000

PHY	**MAC**	**Freq**	**Antenna**(s)
B-MAC		868 MHz	integrated monopole, external wire, and external coaxial connector
Range	**Bitrate**	**Other**	
> 100 m outdoor (ext antenna)	76.8 kbps	-	

Tranceiver(s): Bluetooth ZV4002

PHY	**MAC**	**Freq**	**Antenna**(s)
Bluetooth		2.4GHz	integrated monopole
Range	**Bitrate**	**Other**	
20-30 m indoor	up to 723 kbps	-	

Table 6: At a glance: BTnode rev 3

Power consumption (transceivers)

	BTnode rev3 Bluetooth	BTnode rev3 Low Power Radio
CPU on, Radio listen	92.4 mW	82.5 mW
CPU on, Radio RX/TX	105.6 mW	102.3 mW
CPU on, Bluetooth Inq	198 mW	–
CPU on, Radios both listening	135.3 mW	
Max. Power	198 mW	102.3 mW

Sensors

Builtin	**Available**	**Interfaces**
no	*ssmall* sensor boards, BTsense board	I2C, J1, J2 connectors

External interfaces

Programming	**Debugging**	**Data**	**ADC/DAC**
USB progamming board	J2 connector	UART, SPI, I2C GPIO	yes

Form factor

Dimensions	**Packaging**	**Environmental issues**
6cm x 4cm	standard	lead free

Support and community

Commercial	**Manual**	**Tutorials**	**Other**
yes	yes	yes	–

Users	**web**	**Mailing list**	**Users' Forum**
> 30 projects	yes	yes	no

Table 6: At a glance: BTnode rev3 (con't)

3.5 Ambient platforms: μNode and SmartTag

Ambient platforms can be used in a variety of situations and applications, such as environmental monitoring, farming and agriculture, context-aware personal assistants, home security, machine failure diagnosis, distributed access control systems, building automation, medical monitoring, transportation and logistics, and surveillance and monitoring for security.

The Ambient platforms are developed by Ambient Systems B.V. [Amba], a spin-off of the University of Twente. Ambient platform support mesh-, star-, and hybrid-networks, which is typically composed out of Ambient μNodes (60 x 32mm, see Figure 4), Ambient SmartTags (36 x 20 mm), and Ambient Gateways (not discussed).

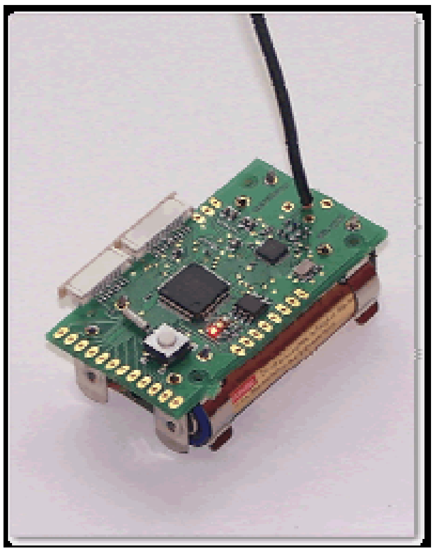


Figure 4: Ambient μNode

MCU The Ambient Systems μNode contains a TI MSP430 microcontroller, featuring 10kB of RAM, and 48kB of flash. This 16-bit RISC processor features extremely low active and sleep current consumption that permits the μNode to run long without the need to change the battery.

The microcontroller features a 32 kHz real time clock and 4.6 MHz system clock.

Radio Transceiver Communication takes place in the 868 MHz or 915 MHz band (frequency hopping can be used in both bands) at a rate of 50 kBaud. The output power is adjustable in the range between -10 dBm and 10dBm. The power consumption is 9mA at -10dBm for transmit, 12.5mA peak for receive, and 2.5μA at standby. The node has an integrated antenna with a 50m range indoors and 200m range outdoors.

The μNode is compatible with FCC standard CFR47 part 15, and ETSI EN 300 220-1.

Storage The Nodes have access to external memory for storing arbitrary data. Up to 4 Mbit of non-volatile memory is available to the user of the platform.

Processor: TI MSP430

speed	**memory**	**features**
4.6 MHz	58kB	Ultra low-power RISC

Memory			
RAM	**ROM**	**cache**	**other**
10 kB	48 kB	–	512 kB EEPROM

Power				
modes	**sleep**	**MCU**	**sensor**	**all**
4 modes	12 μA	< 2mA	-	

Power

input	**battery**	**capacity**	**external power**
2.7-3.6V	2x AA	-	-

charge time	**lifetime**	**other**
-	> 4 month	-

Tranceiver(s): XXX

PHY	**MAC**	**freq**	**antenna**(s)
-	-	868/915 MHz	whip

power control	**range**	**bitrate**(s)
4 levels	> 200m	50 kbit/s

power consumption			
sleep	**idle**	**xmit**	**rcv**
2.5 μA	2.5 μA	12-27 mA	8-12.5 mA

Sensors

builtin	On board switch
available	Temperature, Light, Humidity, etc.
interfaces	RS232, SPI, I^2C

Table 7: At a glance: Ambient Systems μNode

External interfaces

Programming:	JTAG, Over-the-air programming
Debugging:	JTAG, Graphical LCD, Status LEDs
Data:	RS232
ADC/DAC interfaces:	ADC (8 channels, reference voltage), DAC (2 channels), GPIO (8)

Form factor

form factor	dimensions	packaging	environmental issues
PCB, including battery clips	6 x 3 cm	-	-

Support and community

commercial	manual	tutorials	other
yes	yes	yes	-

users	web	mailing list	user forum
> 100 (est)	`www.ambient-systems.net`	-	-

Table 7: At a glance: Ambient μNode (con't)

Sensors The μNode v2.0 comprises an onboard temperature sensor and push button, but can be extended with several sensor boards, like light intensity, humidity, temperature and motion boards through its versatile digital and analog I/O interface. SPI and I^{2}C (bi-directional) interfaces are available.

The platform includes a 12-bit analog to digital converter and standard I/O to which the user can connect sensors and actuators. Several I/O lines can generate interrupts for efficient handling of sensor events. Additionally, three lines are available to generate reference voltages for analogue sensors.

Ambient Systems can design and implement specific interfaces needed for a specific project.

Actuators The Nodes incorporate three programmable status LEDs (RGB) and an ultra low-power graphical LCD of 102 x 80 pixels.

Others actuators can be added on request.

External Interfaces Node board provides pin heads for:

- JTAG (programming and debugging application processor)
- I/O lines
- Analog-Digital-Converter (ADC), reference voltages
- RS232

- I^2C
- SPI
- LCD connector

Energy Storage The μNode V2.0 is powered by two regular AA cells, however it can take any supply between 2.7V and 3.6V. Further, it has been successfully tested with solar cells as power source.

Packaging The dimensions of the μNode V2.0 is 60 x 32 mm. Batteries are connected to battery clips on the bottom side of the μNode. The μNode appears to be made out of standard electronics materials. The status of any environmental issues is unknown.

Availability The μNode is available from Ambient Systems B.V. [Amba]. Each μNode costs approximately EUR 80. Discounts apply for research centers and schools. The μNode are delivered with a CD containing documentation of the platform, software tools, an evaluation version of AmbientRT (can be used for all development work) and a description of its API.

Support is provided by the manufacturer Ambient Systems B.V.

3.5.1 SmartTags

The Ambient SmartTags (see Figure 5 are a "simpler" version of the μNode, with limited memory and sensing and actuating capability.

SmartTags have been designed for applications requiring very low-cost, ultra compact, and extremely low-power sensor nodes. The SmartTags seamlessly communicate with the mesh network established by Ambient μNodes to forward their data using a ultra-low power protocol. The SmartTags can not use multi-hop communication.

The main differences with the μNode is that SmartTags are only 2 x 3.6 cm (can be easily be fit in a key ring), run on a coin-cell battery, have a 16 MHz 8051 microcontroller, 32 Kbit EEPROM memory, the antenna range is 30m indoors and 100m outdoors, can only have one sensor, one button and one LED connected.

3.5.2 User experience with μNode and SmartTag

General At the University of Twente, we have roughly one year's experience with the μNodes and SmartTags of Ambient Systems B.V. The hardware is successfully used in various test applications, for example we experimented with WSNs registering temperature and humidity and tracking people in our building and the hardware is applied in experiments on the Great Barrier Reef, monitoring ocean temperature.

In general, we are satisfied with the hardware and complementary software (AmbientRT). It is reliable and allows rapid development of long-lived applications.

Processing and memory Sufficient for networking protocols and experiments. Sensor data can be stored locally on the nodes themselves in a 4MBit non-volatile memory.

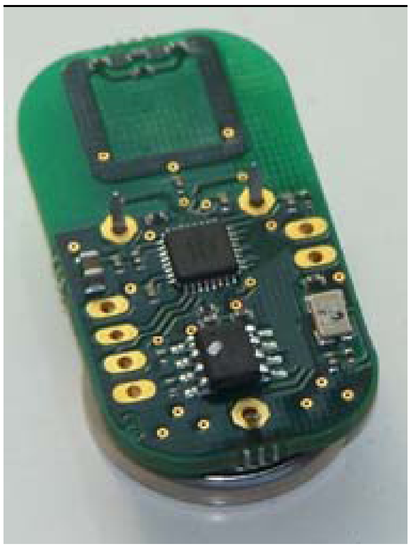


Figure 5: Ambient Smart Tag

Communication capabilities First of all the radio range. In open field experiments, the nodes have an outstanding communication range of 500m. Indoor we did measure a radio range of 100m. Both ranges in the second lowest transmit power.

We successfully implemented various medium access control and routing protocols on the platform. The radio driver included in AmbientRT provides clear functionality and simplifies implementation. The scheduling mechanism in AmbientRT eases critical timing of radio tasks.

The platform can also communicate via serial link (RS232), SPI or I^{2}C. Again, drivers are included.

sensing/actuating capabilities Besides a switch and a low-accuracy temperature sensor in the CPU, the μNodes do not have built-in sensors. However, the nodes have quite some I/O available. We successfully added analogue and digital (SPI, I^{2}C etc) sensors to the platform. The I/O pins can also be used for actuation.

Ambient Systems B.V. has a few sensorboards available for the μNode.

Energy consumption The power consumption of the platform matches documentation. Measurements reveal –not surprisingly– that the transceiver is the most energy consuming device of the platform. However the radio range is satisfactory at lowest transmit power level, resulting in peak currents of roughly 12mA.

The nodes in our tracking application lasted four months on a single set of batteries, while running LMAC and transmitting data messages (32 bytes) once per 10 seconds. However, nodes reporting temperature and humidity once per four minutes are still running on their first set of batteries. An important feature of the hardware is that the battery voltage can be monitored and the nodes can be programmed to signal almost empty batteries.

Platform specific development experience Development for the platform can be done with a GNU toolchain (commercial compilers and debuggers are also available). Installing is simple on both windows and linux machines. There is a vivid user community for developing on the MSP430 CPU's, the processor used on the μNode, willing to answer any question.

Developing and debugging is very convenient on the μNodes, certainly when AmbientRT is used. The devices have a connector for a graphical LCD, which can fit up to ten lines of status information. Drivers and fonts (single and double row) are included in AmbientRT.

Another development feature is the fact that the nodes are remotely programmable; convenient, certainly when large numbers of nodes have to be programmed at once. One μNode connected via serial link to PC allows us to put selected nodes to factory state and to upload new firmware to them. Of course the nodes are also programmable by a JTAG interface.

Discussion In conclusion, we experienced the μNode platform as a good platform to develop and test WSN applications. Processor and memory capabilities are sufficient and the radio range is outstanding. The platform by itself does not contain any sensors, but can easily be extended. The energy consumption of the platform is as is stated in documents. Our message intensive tracking application kept on working for four months on a set of batteries. Development, flashing and debugging are convenient, due to the combination of hardware and software (AmbientRT).

Processor				
type MPS430G1611	speed	memory 48 kB flash/ROM; 10 kB RAM	features 16 bit RISC architecture 3 DMA internal channels	
power modes	power (sleep) $< 0.1\,\mu A$	power (standby) $1.1\,\mu A\ \div 2\,\mu A$ $2.2\,V\ \div 3\,V$	power (active) $330\,\mu A\ \div 500\,\mu A$ $2.2\,V\ \div 3\,V$	
Memory				
	RAM 10 kB (on-chip)	ROM 48 kB flash	cache –	other 4MB serial EPROM (connected via SPI bus)
Tranceiver				
type TDA5250	PHY FSK	MAC CSMA supported	freq 868 ÷ 870 MHz	antenna(s) Internal+External (SMA-connector)
	power control High Low	range indoor $< 30\,m$ $< 10\,m$	range outdoor < 80 m < 30 m	bitrate(s) 64 kbps nominal 19.2 kbps ÷ 34.8 kbps (soft. limited)
	Voltage Range $2.1 \div 5.5\,V$	TX power $\leq 13\,dBm$	Abs.Current $12\,mA$	Abs. Current $9\,mA$
Sensors				
builtin:	*Temperature (LM61)* Range:$-30°C \div +100°C$; Accuracy: $\pm 2°C$ for room temperature; $\pm 3°C$ over the full range Output: $10\,mV/1°C + 600\,mV$ DC offset *Light (NSL19-M51)* *Radio Signal Strength Indicator (RSSI)* Range:$0x0000\ \div\ 0x0FFF$;			
interfaces:	16–pin connector available for additional sensors			
Power				
	input voltage $3\,V$	battery CR2477	lifetime $\sim 1\,h$ full activity	
	external power USB		other power $2 \times\ AAA$ $1.5V$ slim battery pack	
External interfaces				
Programming: Debugging: Data: ADC/DAC interfaces:	*USB, JTAG* *USB* *UART, USB* *12-bit*			

Aspect				
Form factor Oval	dimensions $3\,cm \times 7\,cm \times 2\,cm$	packaging none	environmental issues ???	
Support				
	commercial no	manual yes (39 pp)	tutorials no	other –
Community				
	number of users (est) 100	web site (s)	public mailing list	user forum

Table 7: At a glance: Platform EyesIFXv2.0 (cont'd)

3.6 EYES

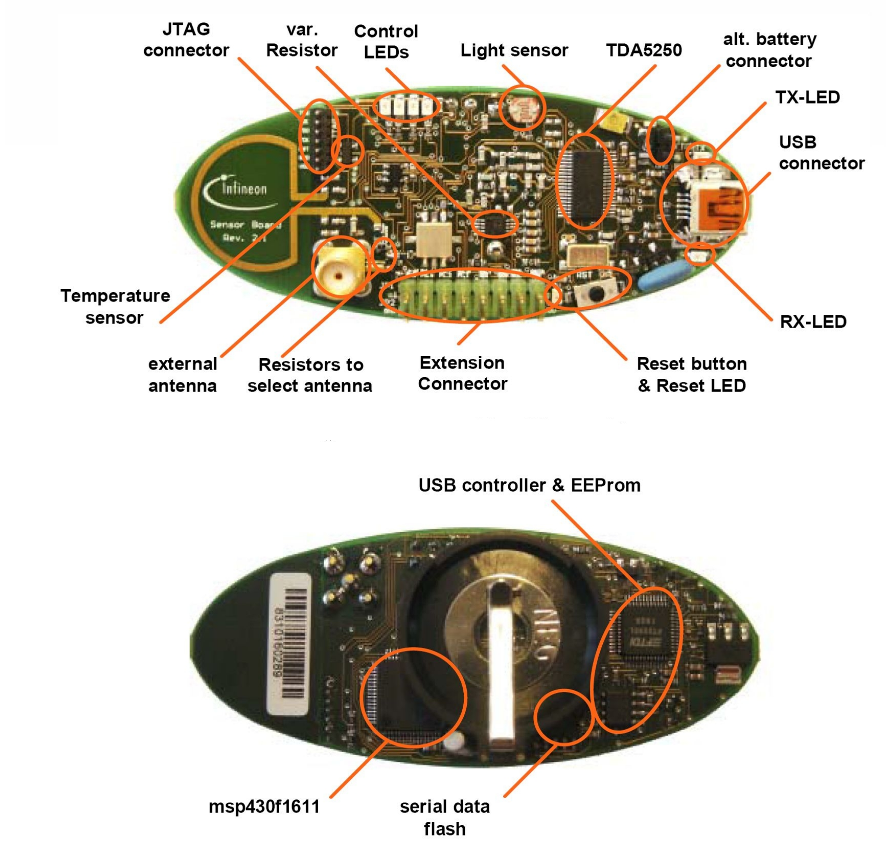


Figure 6: EyesIFXv2 v2.1

3.6.1 Overview

EyesIFXv2 is a sensor node developed by Infineon for the Energy-efficient self-organizing and collaborative wireless sensor networks project EYES.[14] Infineon has combined EYES baseband hardware with a number of optimized peripheral sets to create a series of chips aimed at specific automotive, industrial and consumer applications. Infineon has recently released a new version of the Eyes node, the EyesIFXv2.1. The components of this new board are almost the same of the older v2.0. Two leds have been added to show the radio activity. The two node versions have the same radio unit, so they can communicate without problems. Also the compiled binary images should work properly on both the platforms.

[14]The EYES project is a three year European research project (IST-2001-34734), on self-organizing and collaborative energy-efficient sensor networks.

3.6.2 MCU

As other sensor boards (T-Mote, ESB, etc), EYES nodes are equipped with an ultra-low power MSP430F1611 microcontroller by Texas Instruments, with 10 kB on–chip RAM and 48 kB flash/ROM. Additionally, there is a 4Mb Atmel serial EPROM connect via the SPI bus. Since this μC is used in other platforms, we have collected the details in Appendix 3.7.

3.6.3 Radio Transceiver

PHY The IC is a low power consumption single chip FSK/ASK Transceiver for half duplex low datarate communication in the $868--870$ MHz band. In the eyesIFX nodes it operates with FSK modulation, with a sensitivity < -109 dBm, enabling up to 64 kbps half duplex wireless connectivity (Manchester Encoded).

The platform is also equipped with an on–board stripline antenna and a SMA–connector for an external antenna. The external antenna is the pre–defined manufacturer choice, while onboard antenna can be selected by soldering a 0Ω resistor into the correct location on the chipboard. Therefore, switching between external and onboard antenna requires a (not trivial) manual intervention.

Capabilities The transceiver contains a highly efficient power amplifier, a low noise amplifier (LNA) with Automatic Gain Control (AGC), a double balanced mixer, a complex direct conversion stage, I/Q limiters with Radio Signal Strength Indicator (RSSI) generation, an FSK demodulator, a fully integrated Voltage Controlled Oscillator (VCO) and Phase-Locked Loop (PLL), synthesizer, a tuneable crystal oscillator, an onboard data filter, a data comparator (slicer), positive and negative peak detectors, a data rate detection circuit and a 2/3-wire bus interface. The RSSI is given as an analog voltage between 400 and 1300 mV. It is converted into a digital 12-bit value by the μC, which ranges from $0x0000$ to $0x0FFF$.

Energy consumption The transceiver are power with a supply voltage range in $2.1--5.5$ V. The typical current absorption is $I_s = 9$ mA in reception, $I_s = 12$ mA in transmit mode. The transmit power can be partially modulated by means of a potentiometer, up to a maximum of $+13$ dBm. Additionally there is a power down feature to save battery power.

3.6.4 Sensors

The sensor equipment of the EYES platform encompasses a temperature sensor and a light sensor. (An additional internal temperature sensor is also present, but it is not very accurate.)

The temperature sensor is the Model LM61 produced by National Semiconductor Range, which operates in the range $-30°C \div +100°C$ with an accuracy of $\pm 2°C$ for room temperatures $\pm 3°C$ over the full range. The output is given as an analogical voltage signal that is aproximately proportional to the measured temperature, with $10\,mV$ voltage gap for each Celsius degree, and $+600$ mV DC offset for measuring below zero temperatures. Light Sensor is the Model NSL19-M51 Light Dependant Resistor.

Besides the onboard sensors, additional external sensors can be connected by using the extender port provided by the platform.

3.6.5 Energy Storage

The nodes run on lithium batteries with a capacity of $1000\,mAh$. Alternatively, there is an external power connector to supply DC current as well as a USB port that can power the node.

3.6.6 External Interfaces

External data interfacing is possible through a USB or JTAG interface, which enables programming of the microcontroller and in-circuit debugging. To display status information, an array of 4 LEDs is available (in v2.1 there are 6 leds avaiable). In addition, there is an expansion port that allows the connection of secondary boards with additional analog/digital sensors, actuators, or measurement instruments (*e.g.*, logic analyzer). This connector provides access to the SPI 3-wire bus, additionally the RSSI analog level from TDA5250 can be measured and a external voltage reference for the A/D converter can be provided. Finally some bits from the MSP430 ports can be accessed for I/O operations.

3.6.7 Packaging

The eyesIFXv2.0 node appears as in Fig.15. It has an oval shape that measures approximately $3cm \times 7cm \times 2cm$. The packaging does not provide any form of protection against environmental hazards.

3.6.8 Availability

Infineon produces a starter kit containing 5 nodes with batteries, 1 USB cable, and 1 JTAG interface (see http://www.infineon.com/). Furthermore, the kit includes a CD with TinyOS operating system together with a basic network protocol stacks and development environment that can help beginners getting confidence with the technology.

The evaluation kit costs approximately $300\,EU$, while the single node shall come for 70 EU. Prices and order forms are available at the Infineon website.

3.6.9 Support

Some basic documentation regarding the EyesIFXv2.0 platform can be downloaded, free of charge, from Infineon website. Additional information can also be found in the website of the academical research groups that are using the platform for experimentation, such as Twente University (Netherlands), Technical University of Berlin (Germany), CINI (Italy), University of Padova (Italy).

3.6.10 Experiences with the EYES platform

In this section we report some additional information regarding the EyesIFXv2.0 platform that has been obtained by practicing with the board.

Communication capabilities Although the transceiver is potentially able to provide up to $64\,kbps$ data rate, Infineon limits the actual data rate at 19.2 kbps (software selected) which is often sufficient for sensor networks, in order to save energy. A bit rate of 34.8 kbps has also been successfully tested. The transceiver was designed for much lower bit rates, but the currently used ones make Manchester encoding superfluous – the start and stop bits of the UART protocol are completely sufficient. In principle, such a limitation might be easily removed by the software. Nevertheless, this action might have a negative impact on the reliability of the transmission and, hence, should be accomplished together with a modification of the external circuitry, as specified in the Infineon Data Sheet TDA5250, pages 4-28 to 4-30.

Depending on the coverage range desired, transmission power can be set (via software) to *low* resulting in a covered area of usually below $15\,\mathrm{m}$ in buildings or *high*, resulting in often more than $30\,\mathrm{m}$ range in buildings. An additional regulation is provided by means of a potentiometer that should allow fine grain emitting power regulation. However, from practical experience, it appears that the potentiometer settings are not so accurate as the datasheet reports. This is due to the fact that the potentiometer actually reduces the frequency spread of the FSK modulation – which has no impact up to a certain point. From there on, only the bit error rate raises. Furthermore, the low noise amplifier of the radio can be switched off. When switched off, the consumed energy for receiving drops to $5\,\mathrm{mA}$, and limits the range further. If the sending node transmits at low TX power, the resulting range is usually below $1\,\mathrm{m}$ – enabling sensor networks on a table top.

Sensing capabilities The internal temperature sensor resides in the micro controller and is rather inaccurate; therefore, an external one was added (LM61). To save energy, this external temperature sensor can be disabled by the micro controller.

The light sensor is powered with the reference voltage from the micro controller. This component can be switched off when not needed.

The RSSI is given as a 12-bit value, which ranges from $0x0000$ to $0x0FFF$. It might be worth noticing that when RSSI ADC conversion fails, then the value $0xFFFF$ is returned in the `strength` field of the received packet. Furthermore, from some measurements performed in the SIGNEt lab (at DEI), it appears that nodes need to be calibrated before trusting the RSSI readings for comparison purposes. Indeed, the RSSI value returned for a given received power may vary from node to node.

Energy The recommended current supply for the battery (CR2477) is 0.2 mA; however, in full power mode the eyesIFX node requires $\simeq 15$ mA. Therefore, during intensive transmission tests, an external power supply (USB, Battery Pack) has to be used to prevent packet loss.

Development The EYES platform v2.1 is fully compatible with Version 1.x of TinyOS operating system. It should also be supported by the upcoming TinyOS 2.0 release. The TinyOS installation occupies approximately 8 kB of memory, thus approx $1/6$ of the available memory capacity of the board. Applications are written in NesC, which is based on a simplified version of the standard C programming language. The EYES nodes are fully programmable, so that users can design and develop customized functional blocks, such as Medium Access Control, routing and so on. Nevertheless, a very essential protocol stack is provided by Infineon and TinyOS for beginners. The stack includes an implementation of Carrier Sense Multiple Access (CSMA) for Medium Access Control. Also, a testbed manager software is being developed at the SIGNET lab, at DEI. Version 2.x nodes can be programmed by USB, which provides 2 virtual COM ports, one used for data exchange and the second one for programming the node via bootstrap ability of the microcontroller.

3.7 Microcontroller: TI MSP430F1611

Because of its popularity in many sensor network applications, the following section provides additional information about the TI MSP430 microcontroller. This discussion focusses on the MSP430F1611; however, much of the discussion is applicable to other processors in the MSP430 family.

Features The MSP430's orthogonal architecture provides the flexibility of 16 fully addressable, single–cycle 16–bit CPU registers and the power of a RISC (Reduced Instruction Set Computer). The modern design of CPU offers versatility through simplicity using only 27 real and 24 emulated instructions and seven consistent–addressing modes. This results in a 16–bit low–power CPU that has more effective processing (up to 16 MIPS of performance available), smaller–sized, and more code–efficient than other 8-/16-bit microcontrollers.

The MSP430 clock system is designed specifically for battery–powered applications. Multiple oscillators are utilized to support event-driven burst activity. A low frequency Auxiliary Clock (ACLK) is driven directly from a common 32-kHz watch crystal-with no additional external components. The ACLK can be used for a background real-time clock self wake-up function. An integrated high-speed Digitally Controlled Oscillator (DCO) can source the master clock (MCLK) used by the CPU and sub-main clock (SMCLK) used by the high-speed peripherals. By design, the DCO is active and stable in 1μs. MSP430-based solutions efficiently use 16-bit RISC CPU high-performance in very short burst intervals. This results in very high-performance and ultra-low power consumption.

Memory The MSP430F1611 contains 48kB Flash Memory and 10kB on-chip RAM.

Energy Consumption The supply voltage range for EyesIFXv2.0 is $1.8\,V$–$3.6\,V$. Effectively utilizing peripherals allows the CPU to be turned off to save power or lets the CPU work on other activities to achieve the highest performance. All MSP430 peripherals are designed to require the least amount of software service. The analog to digital converters all have automatic input channel scanning, hardware start–of–conversion triggers and often DMA data–transfer mechanisms. These hardware features allow the CPU resources to focus more on differentiated application-specific features and less on basic data handling. This means that lower cost systems can be implemented using less software and lower power.

The platform supports three ultra–low power consumption operating modes, namely:
Active Mode, in which the platform is fully operative, that requires 330 μA current at $2.2\,V$ voltage;
Standby Mode, in which the transceiver is switched off, while the other functionalities of the board are maintained, that requires $1.1\,\mu$A current;
Off Mode, in which only RAM is retained, that requires 0.2μA current.

Figure 7: MSP430 Microcontroller

4 Operating systems

The other key element of the research platform is the operating system. Because of the extremely limited resources of the hardware platforms, it is difficult to virtualize system operation to create the kinds of system abstractions that are available in more resource rich systems. The concurrency model and abstractions provided by operating system therefore significantly impact the design and development process. As with the previous discussion of the hardware platform, ease of use issues - in this case, design, programming, and debugging - continue to be an important part of the discussion.

The description of each platform follows the outline below:

- Core functions
  - Basic features and design philosophy
  - Concurrency models
    - Multi-threading/tasking
- Basic OS Services
  - Hardware abstraction
  - Timers/Alarms
  - Memory management
  - Sensor primitives
  - Communication primitives
- Service support
  - what MAC, routing, localization or synchronization is provided by the HW/OS?
- Programming Environments
- Testing/Debugging Tools
- Support and community

As in the previous section, the objective presentation of the features and functionality available in each OS platform is followed by a discussion of the users' subjective experience with the platform. A outline for the discussion follows, though it is naturally somewhat less prescriptive than the outline for the presentation of objective material

Some operating systems have been developed for a particular hardware platform, others have been ported to a number of platforms. The following table shows currently feasible hardware/OS pairings.

	ESB/2	μnode	EYES	BTnode	Mica	TmoteSky
TinyOS 1x	Yes	Yes	Yes	Yes	Yes	Yes
TinyOS 2x			Yes		Yes	Yes
Contiki	Yes					Yes
NutOS				Yes		
AmbientRT		Yes				

Table 8: Hardware/OS compatibility

4.1 TinyOS 1.x

TinyOS [HSW+00] is one of the first execution environments specifically designed to meet the requirements of resource-constrained, event-driven and networked embedded systems. Originally developed by the University of California, Berkeley and Intel at the beginning of the decade, it has since become the most popular operating system for Wireless Sensor Networks (WSNs). Its open-source nature and large user-base make it the *de facto* standard for this class of devices.

The TinyOS 1.x family is the latest stable branch of the operating system and is used in this section to describe the basic design principles. This branch will soon be replaced by a complete new rewrite tentatively called TinyOS 2.0 that is presented in the next section.

4.1.1 Component-based modularization

Many of the features of TinyOS stem from the design goals of its implementation language called nesC [GLvB+03]. NesC is an extension of C, adapted to the special needs of the network embedded devices. One of the main characteristics of its programming model is the use of *component modularization*. In this model, the functionality of the traditional monolithic abstraction layers is broken-up in smaller, self-contained building blocks that interact with each other via clearly defined *interfaces*. This hiding of the implementation behind well-defined interfaces preserves the modularity of the solution and promotes reuse. At the same time, the component model supports richer interactions between the building blocks. The interaction is no longer in a strict up/down nature, but starts to resemble a graph. This enables finer extraction of common functionality and definition of complex relationships.

This model represents an especially good fit to the specific requirements in WSNs. Their event-driven nature and the constrained resources require a code organization that is very well covered by the component paradigm. The thin hardware wrappers, the communication primitives and the sensing tasks can all be naturally abstracted in the form of components. The applications are then composed by *wiring* together the necessary building blocks. This entails explicit specification of the components together with the involved interfaces and their role (provider/user) in the information exchange.

4.1.2 Concurrency model

Equally important to the type of code organization is the nature of the supported interactions between the components. The main concern here is how to best facilitate the asynchronous and event-driven type of exchange that occurs not only in the communication context, but also in the user space, as the applications in WSNs are tightly coupled with the environment and usually perform processing as a reaction to some sensed event.

Dealing with the dataflow-centric nature of the applications using limited hardware resources requires making smart trade-offs in the operating system design. TinyOS attacks the problem by offering two levels of concurrency: *tasks* and *events*.

Tasks are a mechanism for deferred computation that should be used whenever the timing requirements for the computation are not strict. This includes almost all application processing apart from the low-level, hardware related operations. Components can *post* a task, after which the execution immediately returns to the poster. The actual execution is delayed until the *task scheduler* executes the task later. Due to the high overhead of context-switching, the tasks run to completion and do not preempt each other. Consequently, they have to be kept short in order to guarantee low task execution latency and high system responsiveness.

Events also run to completion, but can preempt the execution of tasks and/or other events. A large part of the processing in TinyOS is triggered by receiving events representing hardware interrupts. Events are also used to signalize the completion of a *split-phase* operation as explained in the next sub-section.

TinyOS 1.x supports only single task scheduling policy – a FIFO. In contrast to the rest of the system (that is implemented in nesC), the scheduler is implemented as a pure C file. This makes changing the default scheduler more complicated than necessary. Also, the nesC *task* declarations and *post* commands only support parameter-less tasks, so it is hard to integrate scheduling policies that require additional parameters.

4.1.3 Split-phase operations

Due to the non-preemptive nature, TinyOS does not support blocking operations. This means that all long-latency operations have to be realized in a *split-phase* fashion, by separating the operation-request and the signaling of the completion. The client component requests the execution of an operation using *command* calls which execute a command handler in the server component. The server component signals the completion of the operation by calling a an *event* handler in the client component. In this way, each component involved in the interaction is responsible for implementing part of the split-phase operation.

The decision not to hide the split-phase nature of the long-latency operations has the benefit of forcing the programmer to be aware of their effect on the responsiveness of the system. At the same time, this feature turns out to be one of the biggest stumbling blocks for the novice TinyOS users that are used to the more traditional, linear execution model.

4.1.4 Static program analysis

The simple concurrency model of TinyOS allows high event-handling throughput while keeping the overhead significantly lower than in the traditional thread-based approaches. At the same time, the model is still vulnerable to the typical programing errors occurring in concurrent systems including *deadlocks* and *data-races*. Minimizing the introduction of such errors is particularly important in the domain of embedded-systems where there is no human-operator in the control-loop that can take mitigating actions. Thus, it is of paramount importance that such errors are detected and corrected prior to execution time.

NesC imposes several restrictions on the programmer that reduce the chance of introducing complex bugs in TinyOS code. It is a *static language* that does not support dynamic memory allocation or function pointers (even if the use of *malloc* or *free* in code do not generate compilation errors). Consequently, the call-graph of nesC programs is fully known at compile-time, allowing nesC to perform *whole-program analysis* for safety and performance optimization. This analysis can detect and report almost all potential data-races (any update of shared state from asynchronous code). Using this information, the programmer can remove the error by moving the critical sections into tasks or by preventing concurrent-access using *atomic* blocks.

4.1.5 Hardware abstraction

Abstracting the capabilities of the hardware is a basic OS responsibility. One of the early goals of TinyOS was to facilitate easy traversing of the SW/HW boundary by allowing some SW components to be replaced with real HW modules (that export the same interfaces) and *vice versa*. Since such a flexibility requires unrestricted component composition, the early releases of TinyOS 1.x were lacking a clear organization of the hardware abstraction components. In addition, the exported HW abstraction interfaces were strongly biased by the features of the Atmel AVR microcontrollers used on the mica family of nodes. This has hampered porting to new platform and was deemed one of the most serious shortcomings preventing wider adoption of the OS. The situation was significantly improved with the introduction of the *msp430* platform [HPHS04] in TinyOS 1.1.7 (June 2004), that abstracts the capabilities of the Texas Instruments MSP430 microcontroller family. This new platform, used as basis for the *telos* and *eyesIFX* TinyOS ports, introduced a novel abstraction model with gradual formation of platform-independent interfaces. This model was the forerunner of the Hardware Abstraction Architecture (HAA) used in TinyOS 2.0, leading to a much more streamlined porting process.

The core component categories involved in abstracting the supported HW platforms in TinyOS 1.x are:

MCU core Most of the differences between the MCU core architectures are hidden from the programmer simply by using a nesC/C programming language with a common compiler suite (the GNU C Compiler – GCC). The standard C library distributed with the compiler creates the necessary start-up code for initialization of global variables, the stack pointer and the interrupt vector table.

Pins TinyOS 1.x provides a thin abstraction over the external MCU pins using macros that enable setting and clearing of I/O pins as well as changing the direction and function.

Clocks and timers The top-level timer service in TinyOS 1.x, exported by the *TimerM* component, provides timed periodic and one-shot millisecond-resolution events, multiplexed from a single hardware timer compare register. With the introduction of the *msp430* platform, the timer service has been extended with platform-specific *Alarms* supporting millisecond, 32 kHz jiffy and microsecond resolutions.

Analog-to-digital converters and sensors The original ADC abstraction in TinyOS 1.x provided only commands for triggering a single or repeated conversion from one input channel. The interface did not support sequence conversion modes, setting individual reference voltage per-channel or changing the sample-and-hold time. This has been rectified with the new abstractions introduced by the *msp430* platform that reconfigure the hardware for each sampling command. The top-level OS service is provided by wrapper components that transform the platform-independent representation of the sampled sensors into platform-dependent settings for the specific ADC hardware module.

Data buses TinyOS provides abstraction for several standard data buses like SPI/USART, UART, I^2C, 1-Wire, etc. The abstraction is usually organized in two paths – one for data and second for control. The data interface provides adequate commands and events for sending and receiving bytes via the registers of the particular data-bus module. Using the control path, the clock source, prescaler and baud-rate can be changed, and the generation of interrupts during sending or receiving can be enabled or disabled.

External storage The secondary memory, in the form of EEPROM or flash chips, is interfaced to the MCU using one of the above data buses. At the lowest level, the external storage components provide execution of an "atomic access" to the chip, like sending a single command, read, write or erase instruction. At the top-level more complex abstractions are built starting from simple *block* or *log* reading/writing up to powerful filing abstractions like *Matchbox*.

Radios The component organization of the radio chip abstractions mainly depends on the particular data interface that the radio supports. The basic data unit (bit, byte or packet) is usually directly exposed by the lowest level components. The control of the radio is performed using a combination of hardware pins and control registers that are accessed using the data bus abstractions. In some of the supported radios (like the CC2420) the physical layer and parts of the MAC protocol are already defined by the hardware. For the bit and the byte radios (like the RFM, CC1000, etc.) the basic radio abstractions also contain software components implementing the needed packet serialization and medium-access logic.

4.1.6 Networking services

The *active messages* are the main communication abstraction in TinyOS [LMG+04]. They consist of a small identifier that is attached to each message, specifying the action that needs to be taken on the receive side when a given message has been received. In practice the active messages are used similar to the port concept in the TCP/IP stack. Each application has a subset of AM indexes reserved and its receive handlers are triggered whenever the networking stack receives such a message.

The lower parts of the networking stack are generally encapsulated in an abstraction called *GenericComm* that provides single hop unicast and broadcast service. The GenericComm exposes the *SendMsg* and *ReceiveMsg* interfaces for sending/receiving fixed size message buffers – *TOS_Msg*. The abstraction does not support send buffering. On the receive side it performs buffer exchange with the application so that a new message can be received while the user components are processing the current message.

The basic multihop communication pattern supported in TinyOS is the *reverse-tree* routing. A single node is selected to act as a *root* node. Each node in the network selects a *parent* node that is used for forwarding the messages back to the root and maintains its depth in the tree in the form of a *hop-count* to the root.

The tree-routing protocols can differ in the way that this structure is constructed and used as well as in the way it is maintained in the face of changing topology/radio environment. The first implementations have used an active approach in building the tree, issuing periodic floods from the root to maintain the topology. Lately, mechanisms have been introduced that rely on snooping the forwarded messages between the nodes and their parents. This information is used to create *candidate-parent* tables from which a new parent can be selected when the connection with the current one does not satisfy a predetermined *quality* metric. In the simplest case, the RSSI value is used to perform this evaluation. Because of the volatility of the quality metric, its direct use can lead to frequent changes of the routing structure. Consequently, more sophisticated techniques have been used to smooth out the metric, and filter out the unreliable neighbors by rejection of the detected asymmetric links.

On top of the forwarding, the tree-routing protocols usually provide sender buffering and message retransmission capability using the *QueuedSend* interface, for example.

4.1.7 Toolchain and PC-side tools

For basic code writing, the nesC package contains several plug-ins that provide improved support for editing nesC source files in the most popular text editors like vim, emacs, kwrite, etc. Recently, several academic institutions have started projects for building an integrated visual development environment for nesC/TinyOS using the Eclipse framework.

The TinyOS distribution also contains an extensive set of tools that help the users through the remaining phases of the development cycle like compilation, installation and debugging of applications. Most of them are integrated using a hierarchical system of make-scripts [SM00] that hide the complexity of the individual tools behind a convenient rule-based interface that is easily mastered even by the novice user. The make system leverages the concept of WSN *platforms* to pre-configure the component include-path, select the proper compiler, "uploading" tools and their parameters.

For example, the compilation and installation of a mica2 application with node address 2, including debug information, can be specified using:

```
make mica2 install.2 debug
```

Apart from the basic development tools, the distribution carries an extensive set of utilities that facilitate the writing of node-to-PC gateway applications. The serial abstraction in TinyOS 1.x contains a nesC implementation of a framed serial protocol similar to HDLC that can be used for bi-directional communication between a sensor node and a PC-host. This is complemented by the *SerialForwarder* utility on the PC-side that provides a convenient *sockets* abstraction over this packet stream. The writing of the PC-side applications is further simplified by the *mig* and *ncg* tools that can extract a packet-format specifications from nesC applications and automatically create Java boilerplate code for creating and manipulating such packets.

TinyOS also has its own simulation framework, *TOSSIM* presented in Section 6.1.

4.1.8 Licensing model, community support and documentation

In addition to the technical characteristics, the acceptance of an operating system critically depends on its licensing model, the quality and extensiveness of the documentation as well as the existence of lively user community that can provide peer-support. TinyOS 1.x has managed fairly well on all these aspects, as witnessed by the extensive list of over 500 academic and industrial institutions that use the OS.

One of the main reasons for the TinyOS success lies in the open nature of its development model, with the core of the operating system under OSI certified open-source licenses like the BSD License and the Intel Open Source License. The development effort is concentrated around a public cvs repository hosted on the *Source Forge* web platform, while the user community evolves around the project web-site at `www.tinyos.net` and the very active peer-support mailing list `tinyos-help`.

For novice users, the existence of an extended step-by-step *tutorial* [Tin03] has proved as very beneficial. Although a bit outdated (not uncommon for dynamic open-source projects like TinyOS), it illustrates well the basic TinyOS concepts and the common pitfalls. The new TinyOS programming manual [Lev06] provides a another useful alternative, while the nesC reference manual [GLCB06] is targeted at the advanced user that seeks deeper understanding of the language inner-workings.

4.2 TinyOS 2.0

The growth and the maturation of sensor systems during the last several years has led to much better understanding of the abstractions and boundaries a sensor OS should provide. While TinyOS 1.x has been able to adapt with this progress, it became obvious that some design requirements are now more important than others and that significant changes to the OS structure is needed to rectify the situation. This has led to the development of TinyOS 2.0 [LGH+05], a second-generation mote operating system that keeps many of the basic ideas of its predecessor while pushing the design in key areas like greater portability and improved robustness and reliability.

4.2.1 Portability

TinyOS 2.0 supports greater platform flexibility in several ways. First, it introduces a three-layer *Hardware Abstraction Architecture* (HAA) [HPH+05] that imposes composition rules on the components that abstract the hardware resources. The bottom layer is formed by the Hardware Presentation Layer (HPL) that provides access to basic resources such as registers, interrupts and pins via nesC interfaces. The middle layer is the Hardware Abstraction Layer (HAL) which has higher-level interfaces that provide useful abstractions of the full capabilities of the underlying hardware. The top layer is the Hardware Independent Layer (HIL) which presents abstractions that are hardware independent and therefore cross-platform. In addition to this "vertical" dimension, TinyOS 2.0 introduces the concept of *chips* to provide "horizontal" decomposition of the hardware abstraction code, leveraging the fact that mote hardware is usually built out of standard chips, with well defined interfaces. Reflecting these physical interfaces as platform-independent abstractions like buses allows reuse of subsystems corresponding to these chips across different platforms.

The second aspect of the improved portability of TinyOS 2.0 is the use of version 1.2 of the nesC language, which has new features to better support cross-platform networking. This version of nesC introduces the notion of *network type* at the language level: programs can declare structs and primitive types that follow a cross-platform (1-byte aligned, big-endian) layout and encoding. This allows services to specify cross-platform packet formats without resorting to macros or explicit marshaling/unmarshaling.

4.2.2 Robustness and reliability

TinyOS 2.0 improves system reliability and robustness by redefining some of the basic TinyOS abstractions and policies such as initialization, the task queue, resource arbitration and power management. For example, in TinyOS 1.x all components share fixed-size task queue and a given task can be posted multiple times. This causes a wide range of robustness problems, as if a component is unable to post a task due to the queue being full, it may cause the system to hung. In TinyOS 2.0, every task has its own reserved slot in the queue and can be posted *only once*. The new semantics lead to greatly simplified code (no need for task reposting on error) and more robust components. The same principle of compile-time allocation and binding is applied to all aspects of the system: components allocate all of the state they might possibly need; and all invariants are explicitly reflected by the components and their interfaces, rather then being checked at runtime. This design principle limits the flexibility, but makes many OS behaviors deterministic.

Apart from these new policies, TinyOS 2.0 also brings a completely redesigned timer system, new sensor stack, vitalized Active Messages communication layer, improved serial stack as well as new default dissemination and collection protocols.

4.2.3 Toolchain and PC-side tools

The new toolchain distributed with TinyOS 2.0 is characterized by cleaner separation between the generic nesC tools and the TinyOS-specific extensions. The make-based integration is maintained, and the Java development support is extended with plain C and Python variants. TinyOS 2.0 comes with a much more powerful simulation framework in the form of TOSSIM 2.0 with multi-resolution platform emulation, improved radio modeling and better nesC–Python integration.

4.2.4 Development model

The TinyOS 2.0 development effort uses the same technical infrastructure as in TinyOS 1.x, but with significant differences in the collaboration model that reflects the maturation of the project. The core development is currently centered around the TinyOS 2.0 Core Working Group (WG) that includes developers from Stanford, TU Berlin, UC Berkeley, UCLA, Crossbow, Moteiv, Arched Rock and Intel. The group interacts with the wider user community by issuing requests for comments on the TinyOS Enhancement Proposal (TEP) documents, that describe the design and structure of a given subsystem and document a reference implementation.

The success of the new model has led the members of the TinyOS community to start a few other smaller WG with more targeted agendas like the Networking WG, 8051 WG, 802.15.4 WG, etc. To better coordinate these efforts in the future, the community is in the process of forming a TinyOS Alliance that will provide strategic guidance to the project and push sensor networks towards ubiquity and deployment.

TinyOS 2.0 is currently available as a beta release from the TinyOS project website `www.tinyos.net` with a full-release slated for Q3 2006.

4.3 Contiki

The Contiki OS is an open source, highly portable, networked, multi-tasking operating system for memory-constrained systems [DGV04]. Contiki is developed at SICS, and is used in a number of project such as the RUNES project [Pro]. Swedish company Ewmitech is running Contiki in production networks on their Atmel-based hardware.

4.3.1 Basic features and design philosophy

Contiki is designed to be easily portable and runs on a variety of platforms including the ESB platform (see Section 3.2) and Tmote sky (see Section 3.3) where several low power modes are supported. The Contiki version for Tmote sky includes support for using the available 1 megabyte external flash memory.

Contiki includes advanced reprogramming support in form of loadable modules. This feature is useful for both program development since it shortens the development cycle and deployments. Replacing a module is of course more energy-efficient than having to replace the whole image.

Contiki includes the uIP TCP/IP stack, which includes web server, ftp server as well as a number of other example applications. While TCP/IP may not as energy-efficient as custom protocols tailored for wireless sensor networks, using TCP/IP provides interoperability with existing systems and makes it easy to integrate Contiki into existing IP network infrastructures.

4.3.2 Concurrency models

Contiki supports three concurrency models: *events*, *threads*, and *protothreads*. The Contiki kernel is event-based upon which application programs can use any of the three concurrency models, or a combination of them.

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

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

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

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

Protothreads Protothreads are a novel programming abstraction that provides a *conditional blocking wait statement*, PT_WAIT_UNTIL(), that is intended to simplify event-driven programming for memory-constrained embedded systems. The operation takes a conditional statement and blocks the protothread until the statement evaluates to true. If the conditional statement is true the first time the protothread reaches the PT_WAIT_UNTIL(), the protothread is not blocked but continues to execute without interruption. The PT_WAIT_UNTIL() condition is evaluated each time the protothread is invoked. The PT_WAIT_UNTIL() condition can be any conditional statement, including Boolean expressions.

A protothread is *stackless* meaning that a protothread does not have a history of function invocations. Instead, all protothreads in a system run on the same stack, which is rewound every time a protothread blocks.

A protothread runs inside a single function. A protothread is driven by repeated calls to the function in which the protothread runs. Because they are stackless, protothreads can only block at the top level of the function. This means that it is not possible for a function call to block inside the called function - only explicit PT_WAIT_UNTIL() statements can block. Nevertheless, by using hierarchical protothreads, it is possible to perform nested blocking.

The beginning and the end of a protothread are declared with PT_BEGIN and PT_END statements. The protothread code starts executing after the PT_BEGIN statement and ends at the PT_END statement. Protothread statements, such as the PT_WAIT_UNTIL() statement, must be placed between the PT_BEGIN and PT_END statements. A protothread can exit prematurely with a PT_EXIT statement. Statements outside of the PT_BEGIN and PT_END statements are not part of the protothread and the behavior of such statements are undefined by the protothreads mechanism.

Protothreads can be seen as a combination of events and threads. From threads, protothreads have inherited the blocking wait semantics. From events, protothreads have inherited the stacklessness. The main advantage of protothreads over traditional threads is that protothreads are very lightweight: a protothread does not require its own stack. Rather, all protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained systems, where a thread's stack might use a large part of the available memory. For example, a thread with a 200 byte stack running on an MS430F149 microcontroller uses almost 10% of the entire RAM. In contrast, the memory overhead of a protothread is as low as two bytes per protothread and no additional stack is needed.

The protothreads mechanism does not specify a specific method to invoke or schedule a protothread; this is defined by the system using protothreads. For example, protothread application programs running on top of the uIP TCP/IP stack are invoked every time a TCP/IP event occurs and when the application is periodically polled by the TCP/IP stack. Similarly, in the event-driven Contiki operating system a process' protothread is invoked whenever the process receives an event. The event may be a message from another process, a timer event, a notification of sensor input, or any other type of event that the system supports.

Protothreads can be seen as *blocking event handlers*. Normally, an event handler must explicitly return to the caller after processing the event. In contrast, by implementing an event handler as a protothread, the code in the event handler can use the PT_WAIT_UNTIL() statement to perform conditional blocking. The underlying event dispatching system does not need to know whether the event handler is a protothread or a regular event handler.

Multi-threading In Contiki, multi-threading is implemented as a library on top of the event-based kernel. The library is optionally linked with applications that explicitly require a multi-threaded model of operation.

The library is divided into two parts: a platform independent part that interfaces to the event kernel, and a platform specific part implementing the stack switching primitives. In practice, very little code needs to be rewritten when porting the platform specific part of the library. For reference, the implementation for the MSP430 consists of 25 lines of C code.

Unlike normal Contiki processes each thread requires a separate stack. The library provides the necessary stack management functions. Threads execute on their own stack until they explicitly yield.

Initial versions of the Contiki multi-threading library had optional preemptive multi-threading. However, as this feature was not used by application programs, the feature was removed for subsequent versions of Contiki.

4.3.3 Basic OS Services

Hardware abstraction The Contiki core code base provides abstractions for a number of hardware devices typically found on sensor network platforms: radio, LEDs, flash ROM, EEPROM, SPI, UART, RS232, watchdog timers, and external memory. Additionally, there are several abstraction modules that are under development as part of particular ports of Contiki to specific hardware platforms. As these abstraction modules mature they may be moved up to the Contiki core code base. Examples include hardware interrupt management and sound emitter hardware.

Additionally, there are a number of abstractions for sensor hardware that currently is under development as part of Contiki ports to the ESB and Telos Sky platforms. Examples include temperature sensors, passive IR sensors, radio signal strength sensors, vibration sensors, light sensors, sound sensors, and battery level sensors.

Timers Contiki provides two types of timers: timers with the ability to post events called *etimers* and timers without any event-posting ability. Event-posting timers are used by application programs who get an event when a timer has expired. Timers without event-posting ability are used in interrupt handlers from which events cannot be posted. Such timers also incur less overhead in terms of execution time.

Contiki timers are one-shot, meaning that a timer expires only once after being set. If a periodical expiration behavior is wanted, the application program must implement this by itself. The Contiki timers support this by providing a way to *reset* a timer so that its expiration time is set relative to the last expiration time. This ensures that periodical timers do not drift because of the time between the actual timer expiration and the time of the reset operation.

Memory management Contiki supports two forms of memory management: memory block allocation and managed memory allocation. With memory block allocation a program defines a number of fixed size blocks that the program should be able to allocate during run-time. The memory for the memory blocks is statically allocated at compile time which means that the application program can be sure that memory is always available. The memory for the memory blocks cannot be accessed by other applications. Memory is not automatically deallocated, but must be explicitly deallocated by the application program.

While memory block allocation is a simple yet very useful mechanism, there are situations in which the memory is not efficiently utilized. For instance, when the memory usage of an application program varies over time, it may be advantageous to let other parts of the system have access to the memory when the program does not itself use it. This is when the managed memory allocator is useful.

The managed memory allocator has a single pool of memory from which all application programs in a system can allocate memory. The allocated memory does not need to be of fixed size, but each allocation can be a different size. To avoid fragmentation, the managed memory allocator uses a compaction mechanism. Every time memory is deallocated, there is risk of fragmentation of the available memory. The managed memory allocator then compacts the allocated memory by moving the allocated memory so that any holes in the memory heap are removed. Since an application program cannot be sure that the allocated memory is at a fixed position, all access to managed memory goes through a special function, implemented as a C macro for performance reasons, that returns the actual memory position of the allocated memory.

Sensor primitives The *sensor* module in Contiki provides an abstract interface for the hardware sensors of an underlying platform. The interface has five functions that can be called from an application program: *activate*, *deactivate*, *value*, *configure*, and *status*. The Contiki system calls the *init* function of a sensor during system startup. Additionally, the interface has an *irq* function that a hardware device driver uses if the sensor uses a hardware interrupt.

An application program that activates a sensor gets sensor events every time the sensor changes value, if the sensor is configured to do so. Not all sensors have semantics that work well with such an interface, however. Examples include temperature sensors that an application program might wish to periodically query rather than to wait for a change in value. The sensor can then be configured to not send events, but to only report its value if the sensor's *value* function is called.

When sensors are deactivated, the sensor API has support for turning off the power to the sensors, thus conserving energy of the sensor board.

Communication primitives There are no standard communication primitives in Contiki as of yet because this area currently is under development. However, the current development seems to converge towards using the Contiki service mechanism to achieve independence of the actual protocols or mechanisms that implement a particular communication service. There currently are implementations of a number of routing protocols, including tree flooding, convergecast, and AODV.

The Contiki convergecast routing service illustrates how the Contiki service API is used to achieve a separation of a communication service and its implementation in terms of network protocols. Convergecast is a network service that transports data from the fringe of the network towards a central sink node. How the data messages finds their way from the network fringe towards the sink is defined by the network protocol used. To be able to use different kinds of convergecast protocols, the Contiki convergecast service does not define the actual protocol but only provides an abstract programming interface that applications can use. A convergecast protocol that implements the programming interface is then called upon to perform the actual transmission and routing of data messages.

The convergecast service defines a simple programming interface that supports three major functions: *send*, *listen* and *unlisten*. The *send* function sends out a message from a fringe node towards the sink. The *listen* function is used by a sink node to specify an interest for a particular type of data. Similarly, the *unlisten* function is used to deregister an interest for a data that has previously been registered with *listen*. In addition to the three basic functions, the service provides functions for configuring parameters of the service and for querying the status of the underlying routing protocol. To implement the convergecast service, a network protocol needs only implement the functions in the application interface.

4.3.4 Service support

MAC The ESB port of Contiki includes a simple MAC protocol for the TR1001 radio chip but there are no generic MAC protocols in the Contiki code base.

Routing There currently are implementations of a number of routing protocols in Contiki, including tree flooding, convergecast, and AODV. There is ongoing work on adding more routing mechanisms to Contiki.

Localization The Contiki code base does not include any localization protocols. However, there are localization mechanisms implemented on top of Contiki that have been developed in other projects and these implementations may be included in the Contiki code base.

Synchronization There are no synchronization mechanisms provided by Contiki, but there is ongoing work on implementing such services.

4.3.5 Programming Environments

Contiki is written in the C programming language and can be compiled with any C99-compliant C compiler. Contiki has been ported to a number of different C compilers and being compliant across C compilers is an expressed design decision.

4.3.6 Testing and Debugging Tools

Since the beginning, Contiki has always been focused on being able to develop code without having to work directly on the target system. This significantly reduces development time and makes software development, testing, and debugging easier.

Contiki supports three levels of off-target development: *host environment*, *simple network simulation*, and *full network simulation*. Under the host environment development environment, the entire Contiki OS and the program under development are compiled to run on the host on which the development is taking place. That is, the entire Contiki OS runs as a user process under e.g. Linux, FreeBSD, or MS Windows. The OS and its programs can then be debugged using standard host debugging tools such as GDB.

Since the host environment does not include any network simulation, development of software that requires network interaction is not immediately possible. The simple network simulation environment adds a rudimentary network simulator to the host environment which allows for simulation of networks of Contiki nodes. Each node runs the Contiki OS and its underlying programs as a separate host process under Linux, FreeBSD, or MS Windows. Network concurrency can thus be simulated: each node runs independently The nodes may communicate with each other using a simulated radio interface. The simulation of the radio medium is very simple: packets does interfere with each other if the sending nodes are placed too close to each other. It is also possible to add random packet drops. These two features allows the software developer to test software under network conditions that are slightly more realistic than a non-loss radio environment is.

The simple network simulation environment features a simple graphical user interface that shows all simulated nodes in a square grid where nodes appear as dots and radio traffic appear as circles where the radius of the circle is the range of the simulated radio transmission. The user can click in the grid to produce sensor input to the simulated nodes.

Finally, the full network simulation environment called COOJA [st06], which is still under active development, provides a deterministic and flexible environment for testing and developing software for Contiki. COOJA provides a significantly more advanced simulation of radio propagation effects and allows for multiple radio interfaces on the nodes.

When the software has been debugged in one of the simulation environments, the code can be tested on the target platform. Our experience is that this way of doing development is much easier than to do all development directly on the target platform. The initial development done under a simulation has a much shorter compile-test-debug cycle than direct on-target development.

4.3.7 Support

Documentation The Contiki source code is well-documented with in-line comments that are compiled into HTML and PDF documentation. The documentation can be either browsed with a web browser or printed out in book form. In book form, the documentation currently (April 2006) comprises over 200 pages.

The documentation consists both of description of individual functions, with their arguments, return values and side-effects, and of tutorial-style examples of how the Contiki modules are to be used.

Tutorials There are user tutorials under development, mostly within the Runes project [Pro].

Community There is currently no organized community around Contiki.

Licensing Contiki is distributed under a 3-clause BSD-style open source license that allows for unrestricted commercial and non-commercial use.

4.3.8 Experience

We have experienced that protothreads drastically simplify programming sensor nodes by offering a conditional blocking wait operation. Protothreads are also the basis of the protosocket interface. Protosockets provide an interface to uIP similar to the traditional BSD socket interface. The combination of protothreads and the protosocket interface make it possible to write network programs that are structured in a linear way, built around conditional blocking calls to the protosocket interface. Experience has shown that these abstractions simplifies the implementation of e.g. asynchronous protocols.

We have developed several applications on top of the Contiki operating system. Our experience is that using the process-model that Contiki provides (internally using protothreads) in combination with IP-based communication primitives greatly simplifies application development. For example, different protocols can easily be implemented as separate processes that open different IP ports. This makes the usage of the blocking wait statement even more simpler. The blocking wait statement for reception of a packet is only satisfied on reception of a packet to the right IP port. For example, adding a code update protocol involves mainly only a adding a new process that communicates using an unused port.

4.4 BTnut

BTnut is a lightweight, C-based operating system that provides the necessary software support to the BTnode hardware platform (presented in section 3.4). BTnut is built on top of Nut/OS, a simple, general purpose operating system for embedded devices[15] Nut/OS, licensed for both commercial and non-commercial applications under the BSD license.

The Nut/OS core, some BTnode-specific drivers and the Bluetooth stack for the on-board Bluetooth radio are the three pieces that constitute the BTnut system software. In the remainder of this section, we will provide a brief overview on the most significant characteristics of the BTnut system software.

Interested readers can refer for more information to the Nut/OS manuals and API specifications[16] as well as to the several works published by the BTnode/BTnut developers [BKM+04, BDH+04, KL01].

4.4.1 BTnut Core Functions

The BTnut core supports event-driven threads scheduling, offers primitives for basic memory management, synchronization, streaming I/O, and provides low-level device drivers.

The BTnut system software uses plain C-based programming and is based on standard operating system concepts, familiar to most developers. The operating system also offers a modular design, so that only needed parts are linked together and no unneeded parts are loaded into the limited program space of the target hardware platform (e.g., the BTnode).

Concurrency Model BTnut implements cooperative multi-threading. Since a single CPU can't run more than one thread at a time, BTnut provides scheduling and context switching services in order to allow a concurrent execution of several threads.

Threads in BTnut are labelled with a priority value (ranging from 0 to 255) that indicates how urgently a thread need to be executed. The thread with the lowest priority value is the most urgent and thus the one on the top of the priority ordered queue, in which all threads are put when waiting for execution. The most urgent thread is always run, unless it is waiting for an event to be generated. When a thread is running, it is not bound to a fixed time frame and can thus rely on not being stopped unexpectedly during execution, unless an interrupt signal is originated by the CPU. Thus, threads yield resources either when they have to wait for an event, or when they receive a CPU interrupt signal. To work properly, this non-preemptive, cooperative scheduling scheme requires individual threads to frequently and spontaneously (even if not waiting for an event) yield control of the CPU, in order to allow other threads to be executed too.

Threads in BTnut are all executed within the same address space and use the same hardware resources. This significantly reduces the overhead related to switching context whenever a thread yields resources to allow other threads to be executed. When a context switch occurs, the function `NutThreadSwitch()` stores the content of the 32 CPU registers on the stack and stores the correspondent stack pointer for retrieving the execution at a later point. Then, the stack pointer of the thread to be started is loaded and the CPU register are initalized with the correspondent values in this stack.

[15]Nut/OS is part of the Ethernut open source hardware and software platform for embedded, ethernet-enabled devices (`www.ethernut.de`). A TCP/IP protocol suite named Nut/Net and Nut/OS constitute the Ethernut software platform. Nut/OS has been adopted by the BTnode developers for building the core of the BTnut system software.

[16]www.ethernut.de/en/documents/index.html

As already stated, threads keep executing as long as they are not forced to wait for some *event* to happen elsewhere, or they are interrupted by the CPU. We briefly summarize the two concepts of events and interrupts in BTnut:

- **Events** Events are a fundamental concept of the NutOS operating system. When a running thread needs to wait for an event to happen before continuing to execute, it yields its resources and lines-up in an event queue. Consequently, the next thread in the priority queue will be executed (if it's not waiting for an event to be posted). Waiting threads will continue with their execution as soon as the event they were waiting for will be posted to the event queue by another (running) thread. Threads synchronization is achieved through this interaction between running and waiting threads. When all threads are waiting for an event to happen, the idle thread (which has lowest priority) will be executed.
- **Interrupts** The cooperative, multi-threading nature of BTnut can be broken only by CPU interrupts. Interrupt routines are executed immediately after being called, even if the currently running thread is not willing to yield the CPU. The interrupted thread will continue with its execution as soon as the CPU finishes running the interrupt routine.

  If parts of the code of a thread need to be executed without being interrupted, interrupts may be (temporarily) disabled by clearing the interrupt enable flag.

4.4.2 Basic OS Services

Hardware Abstraction The BTnut system software relies on the hardware abstractions provided by the Nut/OS operating system. Nut/OS provides drivers to access physical hardware. The drivers' code is divided into a general part and a hardware dependant part, to simplify porting the code to different physical chips. Support is provided by Nut/OS for several standard data buses like UART/USART, SPI, I^2C, for the ADC converters, as well as for clock, timers and LEDs.

Timers BTnut provides both one-shot and periodic timers, as well as several primitives for timer handling. Among others, functions for initializing system timers, starting and stopping asynchronous timers, temporarily suspending the current thread, and the determination of the CPU speed are available. In previous versions of BTnut, timer events were processed in the timer interrupt routines, but are now processed during thread switches, thus reducing the number of hardware interrupts during execution.

To support the cooperative multi-threading scheme described above, threads in BTnut can yield CPU control by putting themselves into a "sleep"-state for an integral number of system clock ticks. A clock tick occurs every 1 ms by default, but may vary depending on the configuration and can be set up to 62.5 ms.

Memory management BTnut provides dynamic heap memory allocation. The so-called *free-list* provides a linked list of all unused blocks of memory which are dynamically allocated to host threads execution environments. The amount of stack space a thread can occupy during its execution must be specified at compile-time by the programmer. Typically, a maximum size for stack space must be estimated, but 512 or even 256 bytes of stack are enough for most applications threads. However, to reduce the risk of stack overflow, some bytes should be added to the estimated maximum size. The number of bytes available on the heap can be easily retrieved by calling the `NutHeapAvailable()` function. For the `main` thread BTnut allocates by default 768 bytes of stack, which is far more than most applications use. To allocate and release memory blocks, standard C calls to `malloc` and `free` can be used. The heap-manager allocates memory blocks keeping the free-list as unfragmented as possible, thus ending up with few large blocks of unused memory rather than with many small fragments.

Sensor primitives The Nut/OS operating systems does not provide support for sensing tasks. Sensing device drivers must thus be written as BTnut expansions or as part of an application. Device drivers for both the ssmall and BTsense sensor boards are available as part of the standard BTnut software distribution. Functions for initializing the sensor hardware during system startup and reading sensor data are provided.

Communication primitives There are no standard communication primitives in BTnut yet, but basic communication services are available for both the Bluetooth radio and the Chipcon radio the BTnode platform is equipped with.

The HCI, L2CAP and RFCOMM layers of the Bluetooth stack have been implemented for the BTnode/BTnut platform[17], and are fully functional. BTnut devices can connect and disconnect to and from a piconet, send and listen to messages, enabling the development of Bluetooth-based applications upon the BTnode/BTnut platform.

Furthermore, using the L2CAP connectionless data channel, a connectionless multi-hop layer has been recently implemented. Higher layer services can easily call the function `mhop_cl_send_pkt` to send connection-less multi-hop packets to a specific specific service on the target device. To send packets from hop to hop, the multi-hop layer encapsulates multi-hop data into L2CAP connection-less packets. Forwarding of a packet to its destination node is done by broadcasting or by consulting forwarding tables maintained by each node in the network: if a destination can't be found in the forwarding table, the packet is broadcasted to all directly connected devices, except to the directly connected device the packet was received from. On the other hand, routing (i.e. updating the forwarding tables on each node) is done by storing the source node's address of an incoming packet together with the connection handle the packet was received on in the forwarding table (source recording). Incoming packets having a target address that match to one of the entries in the forwarding table are then forwarded directly to the corresponding connection handle - instead of being broadcasted.

For the Chipcon Radio CC1000 a standard driver is available for sending and receiving packets. Additionally, more reliable custom drivers are currently under development.

Stream I/O BTnut provides the Nut/OS I/O library, which overrides the functions of the (standard) compiler's runtime library. The streaming I/O is interrupt driven and stream devices use input and output buffers to read/write data.

[17]We would like to point out that the Bluetooth stack for the BTnode has been written taking care of the peculiar constraints of this platform. For instance, the stack has been developed to have a small memory footprint and to be simple, because of limited available memory and processing power. The stack is however compliant with the Bluetooth standard specifications.

4.4.3 Service support

MAC The BTnode platform features two independent wireless communication modules, the Bluetooth module and the Chipcon CC1000 radio module. The Bluetooth MAC functions are supported by the hardware Bluettoth module featured by the BTnode platform. An implementation of the B-MAC [PHC04], a low power media access scheme for wireless sensor networks, is available for the Chipcon radio in the BTnut system software.

Routing Very recently, a connectionless multi-hop layer, has been implemented for the Bluetooth radio and is now part of the BTnut API. Nodes routes packets through the network updating packet forwarding tables.

A basic flooding algorithm for the Chipcon radio module is also currently being implemented by BTnut developers.

Localization Localization services are not (yet) provided by the BTnut system software.

Synchronization Time synchronization is supported in BTnut through packet time-stamping. Higher level time synchronization services are not (yet) provided by the BTnut system software.

4.4.4 Programming Environments

For basic software development, BTnut programmers will need an editor, a compiler[18], the BTnut standard libraries and an ISP (In-System Programming) software to upload compiled applications to the BTnode flash memory. A CD-ROM containing all tools, documentation and BTnut system software required for development and evaluation work on the BTnode platform is delivered when purchasing BTnode hardware. An ISO image of this CD is however also available as a free download from the BTnode website [BTna]. The software development tools can be easily installed on top of the Windows, Linux or on Mac OS operating systems. A windows installer as well as packages for an easy installation on several Linux distribution and on Mac OS are available.

The most recent releases of the BTnode system software can be downloaded at any time from sourceforge.net [19].

Testing/Debugging Tools When running under investigation, a BTnut library can be added to application code in order to trace events while the application is running. Events can be logged on the EEPROM for off-line inspection with a microseconds time resolution, s.t. also the behavior of interrupt routines can be observed.

The BTnode/BTnut platform also hosted the first implementation and the subsequent development of the Deployment-Support Network (DSN), a tool for supporting the development, debugging, monitoring, and testing of sensor networks algorithms and applications. For further information about the implementation of the DSN please refer to the BTnode Website [BTna] and the related publications [RR05b, BDMT05, RYR06].

[18]E.g., BTnut applications can be compiled using the free AVR-GCC compiler, a special build of the well-known GNU Compiler Collection.

[19]`http://sourceforge.net/project/showfiles.php?group_id=81773`

4.4.5 Support

The BTnut system software is a well-documented, open source project. The BTnut API comes along with some example applications that, together with the hardware specifications and a set of useful tutorials[20], allow a gentle learning curve into the BTnut software. A very active developers community and its mailing list[21] also provides a rich source of information.

4.4.6 Experiences with the BTnut System Software

General Once the BTnode hardware has been purchased, getting the first application running will typically take just few hours. The software development tools can be easily installed on top of the Windows, Linux or on Mac OS operating systems. A windows installer as well as packages for the installation on several Linux distribution and on Mac OS are delivered within the BTnut software distribution.

The installation is a three-step process, that first requires to install the tool-chain, then the BTnut system software, and finally to build and upload a sample application. Sample applications showing, among others, how to operate the radio modules and how to access sensor data are provided within the BTnut software system.

Memory Footprint The modular design of the BTnut system software allows to keep its memory footprint on the target hardware platform as small as possible. When compiling a BTnut program, only needed parts are linked together and thus no unneeded parts are loaded into the limited program space of the target hardware platform. For example, a program that causes the green led of the BTnode platform blink requires about 7 Kbytes, while a more complex application that reads sensor measurements, actuates a buzzer, writes data on the EEPROM and communicates through the Chipcon radio, requires about 50 Kbytes of memory space.

The use of multithreading is typically inefficient in terms of memory usage. However, the memory space provided by the Bnode platform, on which the BTnut system software has been extensively tested, proved to be ample enough for implementing typical sensor network applications.

Programming Experiences Developing applications with BTnut is easy for most educated programmers, since it requires knowledge of standard C, and allows the use of threads. Since no new programming language must be learnt, the first BTnut application can be written within few hours: the programmer only needs to get familiar with the BTnut libraries.

Programs can be uploaded to the CPU Flash memory using the `AVRDUDE` driver[22], a full featured FreeBSD Unix program for programming Atmel's AVR CPU's. The driver performs very well when used on most Linux systems, though it may give some troubles when used on Windows. Please refer to the Btnode Website[23] and to the mailing list for more details on this issue.

[20]The BTnut API, the BTnode hardware documentation, and beginners tutorials are available from the BTnode project web site [BTna]

[21]Mailing-list archive available at: `http://lists.ee.ethz.ch/btnode-development/`

[22]For more information please refer to `www.bsdhome.com/avrdude` or to the BTnode Website [BTna]. It is recommended to use the `AVRDUDE` driver and not the UISP tool for uploading programs to the BTnode platform.

[23]Especially the "Tips & Tricks" section: `www.btnode.ethz.ch/Documentation/TipsAndTricks`

4.5 AmbientRT

AmbientRT [HDJH04] is a Real-Time Operating System for embedded devices with very limited memory, processing, and energy resources, such as wireless sensor networks. The kernel size implemented on a MSP430 processor is 3800 bytes, and the kernel itself needs 32 bytes of RAM.

Some of powerful features of AmbientRT include, real-time scheduling, online reconfiguration, and support for a modular data driven architecture. Where other operating systems for tiny embedded applications offer configuration only during compile time, AmbientRT is a dynamic system that is able to adapt its functionality to create the most efficient configuration for every situation.

AmbientRT uses lightweight RT scheduling and dispatching based on pre-emptive Earliest Deadline First (EDF). Furthermore, with the module support in AmbientRT, applications can be defined as modules, compiled off-line and dynamically inserted, or removed, in binary format. Firmware can now be upgraded by replacing only certain parts, instead of the complete binary. This simplifies the upgrade process, and it limits the use of precious energy.

4.5.1 Real-time Scheduler

AmbientRT has an RT-Transactions EDFI scheduler [JMHS03]. EDFI is a lightweight preemptive real-time scheduling algorithm theoretically proven to be deadlock free. Mutual exclusion of shared resources is enforced by the scheduler itself. Through analysis of a given task-set a guarantee on meeting the real time constraints for each task can be given.

The real-time scheduler in AmbientRT uses dynamic priorities. This means that the priority of a task relative to the priorities of other tasks changes over time. The scheduler uses the absolute deadline as the priority of a task. The task with the earliest absolute deadline has the highest priority.

Figure 8 shows an example of two tasks being scheduled. Task B is running since r_B when an event causes task A to be selected for scheduling. The release time of A is r_A. Because the absolute deadline of task A (d_A) is earlier than the one of task B (d_B), task A is of higher priority and is assigned to the processor. After task A finishes, task B completes its operation.

The impact of using dynamic priorities can be seen in Figure 9. In this case the duration of task B is extended and the event that causes task A to be scheduled comes later. Although task A has a smaller deadline ($D_A < D_B$), it has a later absolute deadline ($d_A > d_B$), and is therefore of lower priority than task B. If static priorities should have been used here, task B would have been suspended by task A and wouldn't have completed on time. Using the absolute deadline as the priority in general, leads to a better utilization of the processor than when fixed priorities are used.

4.5.2 Concurrency Model

Context switching is a demanding mechanism in processing power, as well as in memory usage. The advantage of the AmbientRT kernel is that the tasks in the system all share a single stack. Because of this the context of a task doesn't have to be saved somewhere else but can be left on the stack itself. If a task is preempted the new context will be created just on top of the old one. Restoring a context only occurs when the running task exits and a preempted task continues. The task that will continue is always the task that has its context directly below the running task, and therefore restoring a context is nothing more than removing the context of the running task.

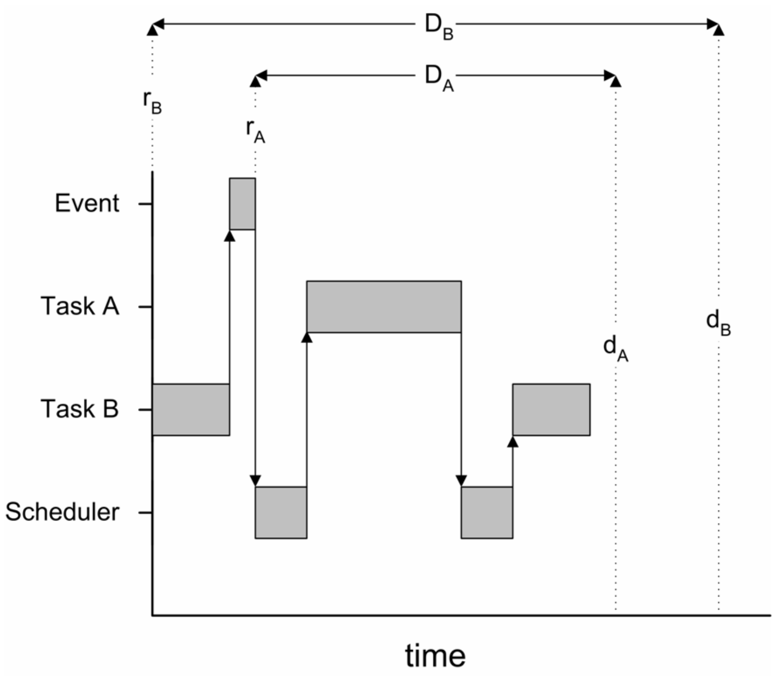


Figure 8: Scheduling two real-time tasks I

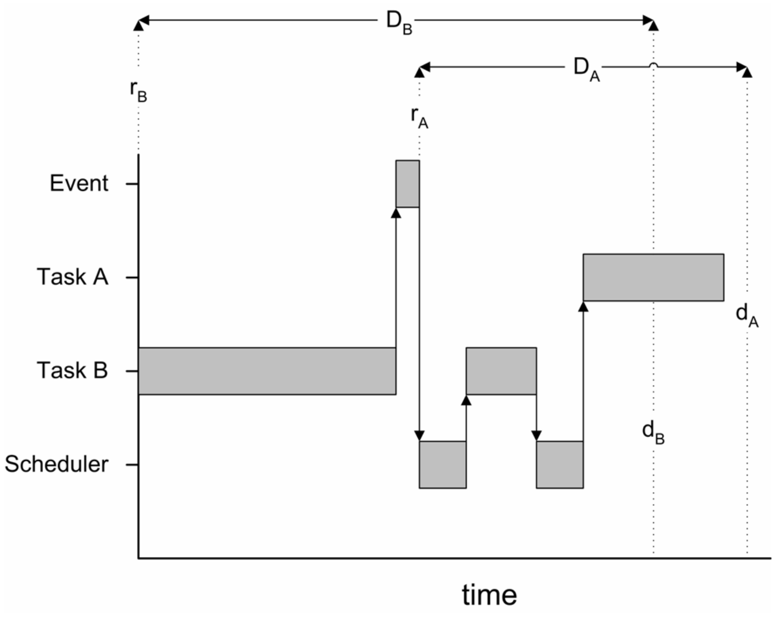


Figure 9: Scheduling two real-time tasks II

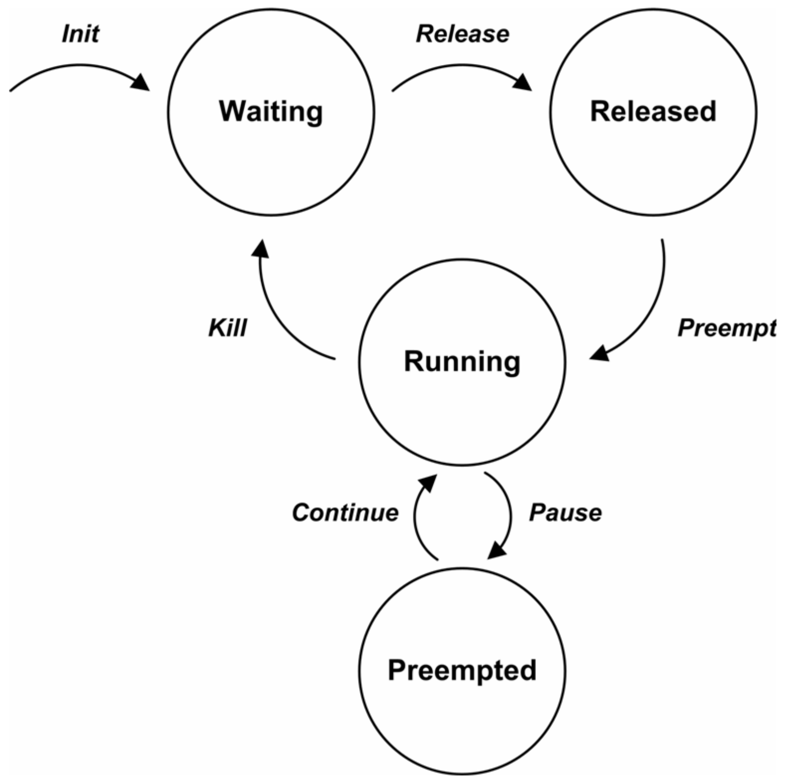


Figure 10: Scheduler task state transition diagram

In AmbientRT a task can be in several states. Figure 10 illustrates the state transition diagram of a task. Every task is created in the *waiting* state, in which the task will wait until an event causes it to be transferred to the *released* state. A task in the released state must run on the processor as soon as possible. The scheduler assigns the processor to the task in the released state with the highest priority. This task is then transferred to the *running* state. When a task is in the running state and a new task is transferred to the released state, the priority of the new task is compared with that of the running task. If the new task has a higher priority, then the running task will be preempted, or paused. The old task is transferred to the *preempted* state, and the new task will transfer to the running state. When a task in the running state finishes, it will be killed and moved back to the waiting state. The scheduler will then compare the highest priority preempted task to the highest priority released task. If the preempted task is of higher priority, it is moved back to the running state where it will continue its operation. If not, the released task will be moved to the running state. At all times, on a single processor platform, only one task can be in the running state.

4.5.3 Resource Synchronization

Resources are elements that can be used by different tasks. A resource can be a variable or a data structure, or a hardware device like a serial port or an LCD display. In order to preserve the integrity of a resource in a multitasking system, it must be prevented that two tasks sharing the same resource, have access to it at the same time. A *mutual exclusion* mechanism avoids this concurrent use of un-shareable resources.

Mutual exclusion in the AmbientRT kernel is obtained through the scheduler. It provides automatic synchronization of shared resources. The scheduler compares on initialization the resource usage lists of every task and generates per task, on basis of the relative deadlines, a threshold value. Comparing this threshold value of a task to the deadline of another indicates whether they share a resource. When the scheduler determines which task is allowed to run it will not only compare the dynamic priorities of the tasks, but also the threshold of the running task to the deadline of the candidate task. If they share a resource, the scheduler will first let the running task finish even if the candidate task has an earlier deadline. In this last situation the candidate task is *blocked* by the running task, and the added delay because of this is called the *blocking time*.

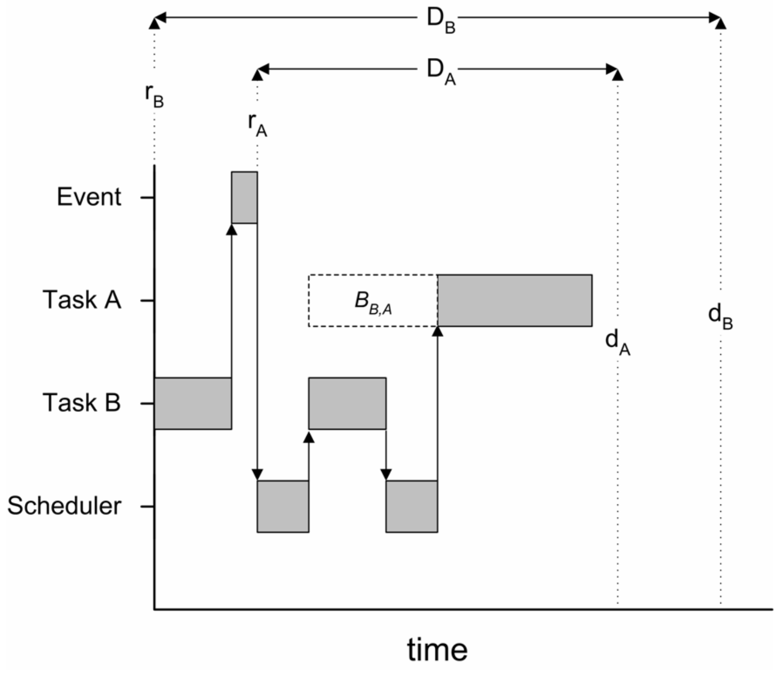


Figure 11: Scheduling two tasks that share a resource

This mechanism is illustrated by Figure 11. In this case task A and B share a resource. This time, when the event occurs, the scheduler let task B finish first. After task B is done, the scheduler runs task A. The blocking time is indicated by $B_{B,A}$.

4.5.4 Real-time guarantees

AmbientRT offers hard real-time scheduling. This means that the system is designed in such a way that for a given set of tasks a mathematical analysis is performed that calculates whether the set is feasible or not. If the set is feasible and is scheduled by AmbientRT it is guaranteed at all times that every task will finish before its deadline.

4.5.5 Data centric architecture

The kernel supports a data centric architecture. Such an architecture enables the application to dynamically reconfigure its functionality. The main differences of the data centric architecture to a static configured application is that the functional building blocks are centrally coordinated, and that these blocks are loosely coupled (meaning that a block has no hard coded connections to other blocks). The component that coordinates the blocks can make and break virtual connections between them like an old-fashioned style telephony patch station. Rearranging the connections will create different configurations, making the system functionally adaptable.

In the data centric architecture, functionality is divided in data producing and consuming components called Data Centric Entities (DCE). Data is a generalization of memory objects and events, where an event is for example the occurrence of a hardware interrupt. Each distinct memory object or event is called a Data Type (DT). The DCEs are used in a publish/subscribe system that allows them to react to DTs produced by others. New configurations can be achieved by altering the set of active DCEs and modifying their subscriptions.

The AmbientRT kernel enables the data centric architecture through its data manager. The data manager keeps track of data with the use of a DT table. Every entry in the table holds the contents of the DT and the list of entities that are subscribed to it. The data manager also regulates the (de)activation of DCEs. When a DCE publishes a collection of data types, the data manager will lookup all the subscribers and activate them.

4.5.6 Dynamic Loadable Modules

Modules are blocks of software that are not part of the operating system itself. The kernel can load and run modules dynamically. Because of the data centric architecture, these modules can be inserted in the configuration during runtime. In this way, AmbientRT is able to support reconfiguration based on functionality becoming available even after device deployment.

A Dynamic Loadable Module (DLM) is a task compiled separately from the kernel code. A DLM can be loaded and executed anywhere in program memory, which makes it the building block for creating new configurations online. With this module support in the operating system, modifications to an application can be done more efficient. Instead of updating the complete application only a subset of the modules making up the task-set has to be changed, resulting in less data traffic and thus less energy consumption. Another advantage is that it allows nodes in a network to be heterogenous in the software point of view, resulting in less occupied memory space and better dedicated operation possibilities.

A ready DLM is transferred to the target hardware through for example the radio or the serial port where on arrival it is stored in the secondary storage. For the communication a packet protocol is provided. This protocol divides the DLM into small packets which are uploaded individually so that the node can store it temporary in RAM before writing it to the secondary storage. When the node successfully received a complete packet, the sender will send the next packet. The protocol can be used for any type of binary that has to be uploaded to the secondary storage. A low complexity file system is used for creating, and keeping track of files representing the received binaries in the secondary storage.

4.5.7 Hardware Abstraction

AmbientRT abstracts the hardware for its client by creating a Hardware Abstraction Layer (HAL). The HAL is a collection of drivers that each provide an abstracted access method to specific hardware devices. For a shorter learning curve and portability, AmbientRT adopts the Portable Operating System Interface (POSIX) standard for accessing devices. With POSIX each device can be accessed through the filesystem using the normal file operations. This will give the client a standardized method of device access while not overflowing it with enormous amounts of different API functions.

4.5.8 Command Shell

The operating system comes with a shell task that can be used to execute commands on the device. When connecting the device to the serial port of a PC and running a terminal emulation program the shell provides a command line interface.

4.5.9 Licensing model, community support and documentation

At the moment two versions of AmbientRT is offered by Ambient System B.V. [Amba]. The FREE version is called AmbientRT Demo, which is fully operational, yet has some restrictions, such as limited number of tasks and heap space. AmbientRT full version has all features, and comes with sample device drivers with source code.

AmbientRT is based on a site license, which means that you can use the system for every product you develop without having to pay additional licenses. For non-commercial applications, universities can use the system for free.

5 Service distributions

In addition to sensor hardware and operating systems, there is some common functionality that is found in higher layers of the communication stack or in middleware elements.

This section of the report

- higher-layer communication protocols
- sensor querying mechanisms
- sensor reprogramming

5.1 Communication stacks

We present two communication stacks.

The first is LMAC MAC layer. The extent to which MAC layer functionality is portable naturally depends on the functionality that the underlying PHY layer is required to export.

The second is μIP. By contrast, like any TCP/IP stack, the μIP TCP/IP stack has been ported to many platforms and is RFC compatible. The direct use of TCP/IP in sensor networks allows a sensor network to be connected to the Internet without use of specialized gateways. However, an extremely memory efficient implementation is needed to fit the resource constraints of the sensor hardware.

5.1.1 LMAC protocol

The LMAC protocol is based upon scheduled access. Each node gets periodically a time interval in which it is allowed to control the wireless medium according its own requirements and needs. Outside this interval, nodes are notified when they are intended receivers. When a node is not needed for communication, it switches its transceiver to standby and is hence able to conserve energy.

Schedule-based MAC protocols have the advantage that nodes are never using their power consuming transceivers, while not needed and hence this type of medium access has good foresight in being energy-efficient. Since each node gets its own turn in using the medium, there will be little collision of messages which is in other types of MAC methods—such as carrier sense multiple access (CSMA)—one of the main reasons for energy waste.

Of course, any protocol requires some overhead to function properly, as is the case with schedule-based MAC protocols. Nevertheless, the philosophy of the designed MAC protocol is to keep it simple, to group transmissions and thus save energy consumption on physical layer overhead and to make it robust to function completely autonomously in a distributed environment.

Currently, the protocol has been implemented on sensor node prototypes, using a program memory footprint of 3.4 kBytes and 550 bytes RAM (including message queues and neighbor table).

When a node has some data to transmit, it waits until its time slot comes up, addresses a neighboring node (or multiple) and transmits the packet without causing collision or interference to other transmissions.

In order to be capable of receiving messages, other nodes always listen at the beginning of time slots of other nodes to nd out whether they are addressed either by node ID or by broadcast address. In the LMAC protocol, nodes can receive multiple data messages per frame, but are only allowed to transmit once per frame. A higher layer in the protocol stack should combine data fragments into one message for transmission whenever possible.

A time slot is further divided into two parts of unequal length: control message (CM) and data message (DM). Between the CM and DM is a small gap, which allows the MAC layer to process the just received CM.

The distributed algorithm LMAC uses is as follows. When a node joins the wireless network, it needs to find out which time slot to control, before it can start sending data and participate in networking. This procedure of finding a time slot can be implemented completely localized and distributed.

First step for a node is to determine which time slots in a frame are already in use, either by direct neighbors or by nodes that are outside transmission range, but would be troubled by interference of the node. We will call time slots not belonging to this set "free" time slots.

Remember that each node already present in the network, broadcasts a CM in its time slot. By just listening to an entire frame, the new node in the network is aware of all its first order neighbors—even by ID. Every node in the network continuously gathers this local time slot usage information and transmits it in its occupied slots field in the CM. This allows a new node in the network to obtain a two-hop view of the network, providing enough information to create a list of free time slots from which the node can choose any. For now we assume that a node chooses a random time slot for the list of free ones.

The above described algorithm is very simple to implement. When a node finds a neighbor transmitting in a certain time slot, it inserts a '1' in the occupied slots bit vector at the respective position for the time slot otherwise a zero is inserted at the position. To obtain a list of free time slots, a node simply needs to 'OR' all received occupied slots bit vectors that were transmitted in the frame. A '0' in the resulting vector means that the time slot is considered free in a two hop region and a '1' that a time slot is already taken by a first or second order neighbor. This ensures a spatial time slot re-usage after no less that three hops.

Synchronization Key issue in scheduled medium access is that it needs a common sense of timing in order to create a long-lived network. Without precise (local) synchronization, nodes have to use long guard intervals to ensure that receivers are ready when transmitters start transmitting, wasting valuable energy.

Timing experiments presented in [vHH06] show that the prototype wireless sensor nodes can maintain relative synchronization with little error. To establish multi-hop synchronization in the network, LMAC uses a hierarchical synchronization scheme (i.e. all timing is relative to the timing of a gateway node).

5.1.2 uIP TCP/IP

uIP is an implementation of the TCP/IP protocol stack intended for small 8-bit and 16-bit microcontrollers. uIP has a very small code memory footprint, on the order of a few kilobytes. RAM usage is on the order of a few hundred bytes. uIP is used by at least 100 companies worldwide for a multitude of different applications ranging from oil pipeline measuring equipment to satellite communication systems. uIP is also used in many research projects at academic institutions.

Despite its small code size uIP includes the base-line IP protocols: TCP, UDP, IP and ICMP. The implementations are RFC compliant and is therefor able to communicate with any other RFC compliant TCP/IP protocol implementation. The TCP implementation also includes support for several passively listening TCP (server) sockets and multiple simultaneous TCP connections. The maximum number of these is configurable at compile time. The TCP implementation supports flow control, fragment reassembly and retransmission time-out estimation. The small memory consumption is achieved by trading throughput for memory: for sending only one outstanding TCP segment per connection is supported.

The uIP code base also includes several example applications, including a web server, a web client, and an SMTP client. uIP is also part of the Contiki operating system (see Section 4.3). The uIP code is well documented and freely available under a three-clause BSD-style license. uIP is also the only TCP/IP stack that has been ported to TinyOS. The uIP port to TinyOS was done by HP Labs.

Traditional TCP/IP implementations have required far too much resources both in terms of code size and memory usage to be useful in small 8 or 16-bit systems. Code size of a few hundred kilobytes and RAM requirements of several hundreds of kilobytes have made it impossible to fit the full TCP/IP stack into systems with a few tens of kilobytes of RAM and room for less than 100 kilobytes of code.

Many other TCP/IP implementations for small systems assume that the embedded device always will communicate with a full-scale TCP/IP implementation running on a workstation-class machine. Under this assumption, it is possible to remove certain TCP/IP mechanisms that are very rarely used in such situations. Many of those mechanisms are essential, however, if the embedded device is to communicate with another equally limited device, e.g., when running distributed peer-to-peer services and protocols. uIP is designed to be RFC compliant in order to let the embedded devices to act as first-class network citizens. The uIP TCP/IP implementation that is not tailored for any specific application.

The uIP stack does not use explicit dynamic memory allocation. Instead, it uses a single global buffer for holding packets and has a fixed table for holding connection state. The global packet buffer is large enough to contain one packet of maximum size. When a packet arrives from the network, the device driver places it in the global buffer and calls the TCP/IP stack. If the packet contains data, the TCP/IP stack will notify the corresponding application. Because the data in the buffer will be overwritten by the next incoming packet, the application will either have to act immediately on the data or copy the data into a secondary buffer for later processing. The packet buffer will not be overwritten by new packets before the application has processed the data. Packets that arrive when the application is processing the data must be queued, either by the network device or by the device driver. Most single-chip Ethernet controllers have on-chip buffers that are large enough to contain at least 4 maximum sized Ethernet frames. Devices that are handled by the processor, such as RS-232 ports, can copy incoming bytes to a separate buffer during application processing. If the buffers are full, the incoming packet is dropped. This will cause performance degradation, but only when multiple connections are running in parallel. This is because uIP advertises a very small receiver window, which means that only a single TCP segment will be in the network per connection.

In uIP, the same global packet buffer that is used for incoming packets is also used for the TCP/IP headers of outgoing data. If the application sends dynamic data, it may use the parts of the global packet buffer that are not used for headers as a temporary storage buffer. To send the data, the application passes a pointer to the data as well as the length of the data to the stack. The TCP/IP headers are written into the global buffer and once the headers have been produced, the device driver sends the headers and the application data out on the network. The data is not queued for retransmissions. Instead, the application will have to reproduce the data if a retransmission is necessary.

The total amount of memory usage for uIP depends heavily on the applications of the particular device in which the implementations are to be run. The memory configuration determines both the amount of traffic the system should be able to handle and the maximum amount of simultaneous connections. A device that will be sending large e-mails while at the same time running a web server with highly dynamic web pages and multiple simultaneous clients, will require more RAM than a simple Telnet server. It is possible to run the uIP implementation with as little as 120 bytes of RAM, but such a configuration will provide extremely low throughput and will only allow a small number of simultaneous connections. Nevertheless, there exists at least one example of a device using this configuration (a pico-satellite system where uIP is used for transmitting data from space to earth and back).

The uIP Application Program Interface (API) is different from most TCP/IP stack APIs. The most commonly used API for TCP/IP is the BSD socket API which is used in most Unix systems and has heavily influenced the Microsoft Windows WinSock API. Because the socket API uses stop-and-wait semantics, it requires support from an underlying multitasking operating system. Since the overhead of task management, context switching and allocation of stack space for the tasks might be too high in the intended uIP target architectures, the BSD socket interface is not suitable for memory-constrained systems.

Instead, uIP uses an event driven interface where the application is invoked in response to certain events. An application running on top of uIP is implemented as a C function that is called by uIP in response to certain events. uIP calls the application when data is received, when data has been successfully delivered to the other end of the connection, when a new connection has been set up, or when data has to be retransmitted. The application is also periodically polled for new data. The application program provides only one callback function; it is up to the application to deal with mapping different network services to different ports and connections. Because the application is able to act on incoming data and connection requests as soon as the TCP/IP stack receives the packet, low response times can be achieved even in low-end systems.

uIP is different from other TCP/IP stacks in that it requires help from the application when doing retransmissions. Other TCP/IP stacks buffer the transmitted data in memory until the data is known to be successfully delivered to the remote end of the connection. If the data needs to be retransmitted, the stack takes care of the retransmission without notifying the application. With this approach, the data has to be buffered in memory while waiting for an acknowledgment even if the application might be able to quickly regenerate the data if a retransmission has to be made.

In order to reduce memory usage, uIP utilizes the fact that the application may be able to regenerate sent data and lets the application take part in retransmissions. uIP does not keep track of packet contents after they have been sent by the device driver, and uIP requires that the application takes an active part in performing the retransmission. When uIP decides that a segment should be retransmitted, it calls the application with a flag set indicating that a retransmission is required. The application checks the retransmission flag and produces the same data that was previously sent. From the application's standpoint, performing a retransmission is not different from how the data originally was sent. Therefore the application can be written in such a way that the same code is used both for sending data and retransmitting data. Also, it is important to note that even though the actual retransmission operation is carried out by the application, it is the responsibility of the stack to know when the retransmission should be made. Thus the complexity of the application does not necessarily increase because it takes an active part in doing retransmissions.

Finally, an alternative socket-like API, called *protosockets*, is under development for uIP. The API is based on protothreads and allows applications to be written in a sequential style. Experience has shown that this simplifies implementation of application layer protocols on top of uIP.

5.2 Sensor Querying Tools

The most commonly defined middleware for wireless sensor networks is query management software. These mechanisms allow an application (or user) to define a query or sequence of queries over the sensor data being sampled by the network. This section describes experience with a publish/subscribe system, as well as providing an overview of some other commonly used mechanisms.

5.2.1 Publish/Subscribe Abstraction

The publish/subscribe interaction scheme is a realization of the data-centric networking paradigm: it decouples the identities of the producers and consumers of data. A subscriber (consumer) specifies its interest in certain data in form of a subscription, for example expressed as a combination of the constraints `temperature < 30 AND humidity > 20`. Such a subscription is not explicitly addressed to a certain publisher (producer) but implicitly addresses all publishers that are capable of publishing data that matches the constraints in the subscription. To limit the scope of a subscription the subscriber may add position, location or radius attributes but it will not include the identity (network address) of the destination. It is the task of a publish/subscribe middleware to deliver all matching notifications from publishers to the respective subscribers; for example, a notification `temperature = 25 AND humidity = 22` would match the above subscription and should be signalled to the registered subscriber(s). There exist several implementation options for the publish/subscribe scheme: centralized vs. decentralized, topic-based vs. type-based vs. content-based publish/subscribe and different variants of the naming scheme used to express subscriptions and notifications [EFGK03].

Sensor nodes are usually deployed redundantly (e.g. in order to account for unreliable wireless links or limited energy supply) and a wireless sensor network application is often not interested in the identity of a particular node, but in the physical properties that can be observed in the environment. Therefore abstracting from node identities by means of identity decoupling as in publish/subscribe is a valuable service of a wireless sensor network middleware. The asynchronous nature of the publish/subscribe scheme - subscriptions and notifications are issued in a non-blocking way - is an additional advantage over traditional middleware concepts like remote procedure calls (RPC) or shared memory systems as it allows easy integration in the event-based sensor node architecture.

The TKN group at TU Berlin has developed a content-based publish/subscribe middleware for wireless sensor networks as a component-based framework for TinyOS.[24]

The top-level architecture of the framework is depicted on Figure 12.

A subscription is expressed as a logical conjunction of constraints over attributes; typically sensors will be represented by attributes and constraints are basic comparison operators (e.g. $=$, $<$, $>$, $\leq$, $\geq$) which compare two attribute values.

The publish/subscribe system is implemented in a decentralized fashion, subscriptions are disseminated to all network nodes and the matching of notifications and subscriptions is performed on the individual publisher nodes in order to save network resources. Exploiting the component-based nature of TinyOS, the middleware can be combined with several routing protocols for the dissemination of subscription and collection of notifications as long as they provide the standard TinyOS interfaces *Send* and *Receive*, respectively (otherwise a layer of wrapper components needs to be implemented).

[24] The middleware was developed as part of the final demonstrator of the European research project EYES (IST-2001-34734)

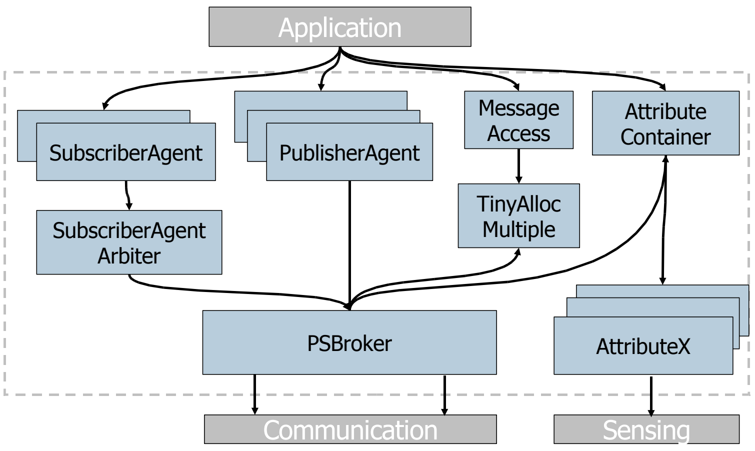


Figure 12: Top-level component architecture of the TKN publish/subscribe framework

Metadata, such as sampling rate or duration of a subscription, are expressed as attributes and the naming scheme allows to easily extend the middleware by new (metadata) attributes and respective operations. The number of constraints and attribute-value pairs in a subscription and notification is limited only by the size of the SDU of the respective routing protocol. Attributes and operations are specified in XML, which facilitates extensions and the decoupling of the implementation, for example using a graphical user interface on a PDA connected to the sensor network via a gateway node. All middleware code is open source and available via CVS from Sourceforge.[25]

5.2.2 TinyDB

TinyDB runs on top of TinyOS. It sees the nodes as storage points of a distributed table called SENSORS. As such, sorting and similar operations are not allowed on this dynamic table. It introduces a materialization concept very similar to view concept in SQL, i.e. a window is created for a given query such that bounded operations are possible on this window.

TinyDB has a built-in lifetime prediction and optimization algorithm. This is used for lifetime optimization for lifetime based queries. So the user can force a lifetime on the network and TinyDB decides on the sampling rate and other (not forced) parameter to satisfy this request.

TinyDB also has support for actuator queries, where, on a given event, the node executing query performs an action rather than sending query results. This property is useful for changing state when a given event occurs.

[25]http://tinyos.cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/contrib/eyes

5.2.3 Acquire

Acquire introduces a different query mechanism than other three middleware examined here. A query is an active entity, which can be a complex query containing several sub-queries. So there is no clear distinction between query dissemination and response gathering stages. As the query disseminates through the network, it is partially resolved and responded by nodes so that it becomes smaller and smaller. As the last node contributes and closes the query, completed response follows the reverse path or the shortest path to the sink. In order to realize this mechanism, nodes continuously (periodically) share sensing data with each other.

They claim that this approach is efficient. However, it also makes Acquire more suitable for one-shot queries and makes it unsuitable for event based queries.

Acquire seems to be an academic test-bed for this new approach.

5.2.4 Cougar

Cougar is a coqmplete query layer for sensor networks. Its main focus is on query optimization algorithms, thus Cougar is more suitable for long running queries. As with the TinyDB, query syntax is very similar to SQL. During the query dissemination phase, a network tree is constructed. The main objective of the project is to reduce communication needs, so in network processing is highly exploited in Cougar. Each node sends its response to its parent node where multiple result packets are combined (compiled-aggregated) into one simple response then sent to one level above (in the hierarchy) up to the sink.

Cougar has a limited usage for event-based queries.

5.2.5 SQS

SQS runs on TinyOS. It is a complete wsn-querying system and constructs a hierarchy tree for query dissemination and response gathering in order to optimize energy consumption. This framework supports multiple queries within one packet. As such, results of such queries are compiled in one single packet (up to four distinct, complex queries supported within one packet). Current implementation supports only one query compilation within the system.

	TinyDB	Acquire	Cougar	Sqs
Data Aggregation	yes	yes	yes	yes
Complex Queries	yes	yes	yes	yes
Continuous Queries	yes	no	yes	yes
One-Shot Queries	yes	yes	yes	yes
Event Based Queries	yes	no	no	yes
Lifetime-Based Queries	yes	no	no	no
Monitoring Queries	yes	yes	yes	yes
Network Queries	yes	no	yes	yes
Node Status Queries	no	no	yes	yes
Nested Queries	no	no	yes	no
Multiple Queries	yes	yes	yes	yes
Multiple Queries in One Packet	no	no	no	yes
Aggregation of Multiple Query Results	no	no	yes	yes
Actuation Queries	yes	no	no	no
Offline Delivery	yes	no	no	yes
Query Optimization	yes	yes	yes	no
Routing Trees	yes	1	yes	yes

Data aggregation: Can a packet contain results from multiple nodes.
Complex queries: support for AND, OR, WHERE operations.
Continuous queries: Support for resident queries.
One-Shot queries: Support for one time queries.
Event based queries: Support for respond-on-condition queries.
Lifetime based queries: Support for queries forcing a life-time on nodes by optimizing system performance.
Monitoring queries: =Continuous queries.
Network queries: Support for querying network structure and health (parent id...).
Node status queries: Support for node status and health queries (battery condition...).

Nested queries: Query within query support (SELECT ... WHERE ID=(SELECT...)).
Multiple queries: Support for concurrent multiple queries.
Multiple queries in one packet: Can one query packet contain more than one query?
Aggregation of multiple query results: Results of multiple queries in one packet?
Offline Delivery: Take one sample every second and send results after 10 readings kind of query support.
Query optimization : Any effort made to optimize the query processing.
Routing Trees: Whether a hierarchical tree is constructed or not (Acquire introduces a different approach, it processes the query as it is distributed and after the last node contributes the result is sent back by reversing the data path).

Figure 13: Comparison of sensor querying tools

5.3 Reprogramming

From experience with wireless sensor networks, it has become apparent that 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.

Manual reprogramming of individual nodes over an interface such as JTAG is impractical in particular if the number of nodes is high. In some cases manual reprogramming is even impossible, for example, when nodes are deployed at remote or inaccessible places. During the development cycle, individually reprogramming nodes leads to inefficient use of time in testing and debugging. In an testbed or deployment environment, especially one consisting of a larger number of nodes, it may be impossible to retrieve all of the nodes, much less individually reprogram them.

5.3.1 Reprogramming Scenarios

There are several reasons why users want to reprogram their sensor networks:

- Software development: most of often a single algorithm or module is updated frequently.
- In sensor network testbeds: WSN applications are installed and tested.
- Correction of software bugs: needed at several levels in deployed application, e.g. both OS and application
- Application reconfiguration, but reconfiguration (sometimes also called retasking [MKL+04]) does not necessarily include code updates [MKL+04].
- Dynamic applications: the whole application is replaced during the lifetime of a network

5.3.2 Code Execution Models and Reprogramming

The choice of the execution model directly impacts the data format and size of the data that needs to be transported to a node. Current execution models include:

- Script languages
- Virtual machines
- Native code that allows several ways of reprogramming:
  - Full image swap: often used approach, e.g. in TinyOS
  - Diff-based approaches:
  - Loadable modules: prelinked, dynamically linked, position independent code

There is an inherent trade-off between virtual machine code and native code: virtual machine code (Mate, Java VM) can be much more compact, thus less bytes need to be transferred. However, code execution is more energy-expensive. Thus, the VM approach is useful when it is known that the network needs to be reprogrammed frequently.

Native code is the most straightforward approach to execute code on sensor networks and a full image swap the common way of reprogramming. This approach is the standard way of reprogramming in TinyOS but requires the replacement of the whole image also for small bug fixes. There exist several variant of diff-based approaches that are able to reduce the size of the code to be transferred for e.g. bug fixes.

Being able to replace modules dynamically during runtime is a more energy-efficient approach taken by e.g. Contiki, Impala and SOS. Impala's updates are coarse-grained since cross-references between different modules are not possible. SOS uses position independent code, i.e. code that only uses relative references. However, position independent code requires compiler support and requires architectures that feature relative addressing modes, currently not available for e.g. gcc and MSP430. The Contiki OS features a linker that is able to perform dynamic linking, relocation and loading of object code files. Contiki also supports the ELF format, the default object file format produced by the GNU compiler collection.

While most of the approaches mentioned above can be reprogrammed over the air, the Mantis OS does not yet support dynamic reprogramming over the radio.

While remote reprogramming is supported in some testbeds such as the the MoteLab at Harvard, there is not much experience with reprogramming in real sensor network deployments.

6 Simulation/Emulation environments

The practical complexities of building sensor networks make simulation and emulation techniques are important research tools. Because such environments have only limited fidelity, it is important to understand the limitations of such tools, this topic is discussed in the final subsection.

Several simulation/emulation environments are presented, including:

- Tossim
- Glomosim
- Matlab
- Avrora
- Omnet++ (and related packages)
- ns-2

This section continues the model of the earlier section. The main features of each environment are presented, followed by a discussion of advantages and disadvantages. A comparison table is also presented.

	Property	**Glomosim**	**Tossim**	**Omnet**	**Avrora**	**NS2**	**Matlab**
1	Simulator/Emulator	Sim	Sim	Sim	Emu	Sim	Sim
2	CPU level simulation	No	No	No	Yes	No	No
3	Scalability	10000s	100s	100s	-	100s	100s
4	MAC simulation support	Yes	Yes	No	-	Yes	Yes
5	Different MAC Protocols	Yes	Yes	No	-	Yes	No
6	Transport Protocol	Yes	No	Yes	-	Yes	No
7	Different Routing Protocols	Yes	No	No	-	Yes	No
8	Multi-tier	Yes	No	Yes	No	Yes	?
9	Mobility	Yes	No	Yes	No	Yes	No
10	WSN specific	No	Yes	No	Yes	No	No
11	Event Generation	Yes	Yes	Yes	Yes	Yes	No
12	Battery Model	No	No	No	Yes	Yes	No
13	Lossy Transmission	Yes	Yes	Yes	-	Yes	No
14	Channel Model	Yes	No	No	Yes	Yes	Yes
15	Localization Simulation	?	?	?	-	???	Yes
16	SMP support	Yes	?	?	Yes	No	No
17	Realtime	No	No	No	Yes	Yes	No
18	Sensor model	No	Yes	No	Yes	No	No
19	Debug tool	?	?	Yes	Yes	No	Yes
20	Language	C	NesC	C++	Java	C++/Tcl	M-code, C, Java
21	GUI Support	Yes	Yes	Yes	Yes	Yes	Yes
22	Commercially Availability	No	No	No	No	No	Yes
23	Ease of use	Medium	Hard	Easy	Medium	Hard	Hard
24	Community Size	Large	Large	Large	Large	Large	Large
25	Ease of getting help	Easy	Easy	Medium	Hard	Medium	Medium

Table 9: Data Sheet of Simulation Platforms

6.1 Tossim

Tossim is the most basic simulator for wireless sensor networks, since it directly runs the actual tinyOS code. By exploiting the sensor network domain and TinyOS's design, TOSSIM can capture network behavior at a high fidelity while scaling to thousands of nodes [LLWC03]. TOSSIM takes advantage of TinyOS's structure and whole system compilation to generate discrete-event simulations directly from TinyOS component graphs. It runs the same code that runs on sensor network hardware.

6.1.1 Advantages

- Runs the actual tinyOS code.
- Has models for many tinyOS and hardware specifications like radio and ADC models.

- Same simulation code will run on the actual motes.
- Provides lossy network link model.

6.1.2 Disadvantages

- Difficult to write codes if only simulation is aimed.
- Not all desired models have simulation support yet (e.g. battery model)
- MAC layer is not well simulated yet.

Some experiences on Tossim gained by the implementations done at YTU exist. Basically, Tossim has been used in order to port the simulation code onto the Mica2s easily. Tossim is using the actual NesC code and creating a C code from it and we could easily produce a topology file and set the boot up times of motes easily. However, some of the experiences were restrictive. Here, the most significant experiences from YTU are stated.

During the simulation implementation of a network querying tool, it is observed that the following drawbacks: In Tossim, a network signal is either a one or zero. All signals are of equal strength, and collision is modeled as a logical or; there is no cancellation. This means that distance does not affect signal strength; if mote B is very close to mote A, it cannot cut through the signal from far-away mote C. This makes interference in Tossim generally worse than expected real world behavior.

Almost in all of these implementations, energy consumption was critical. Both in the querying tool study and some code propagation studies done at YTU, lack of energy model in Tossim has been an issue. After a simulation is run, a user can apply an energy or power model to these transitions, calculating overall energy consumption. Because Tossim does not model CPU execution time, it cannot easily provide accurate information for calculating CPU energy consumption.

Tossim uses ADC and RFM models to simulate different communication scenarios and different ADC readings that can be achieved. Lossy model models node connectivity with graph edges and loss rates for that link. A customized version of lossy model has been used in the tests. As a result, simulator's bit level communication stack has been replaced with packet level communication stack by replacing the bindings of packet transmission component wiring a file to a newly developed packet transmission module.

Another important issue with Tossim is, MAC layer simulation. It simulates the MAC behavior of Mica motes only. When using it, lots of collisions occur when queries are tested in Tossim (although in such cases which, the maximum hop count was only three). For example, in a tree constructing application, a child nodes packet was lost, since its parent tried both sending its own packet and forwarding child nodes packet. This problem was temporarily solved by by-passing the MAC layer.

Also, some inconsistency between TinyOS and Tossim about data simulation have been noticed. Although in real motes data losses are in packet-level, Tossim support only bitlevel simulation of losses.

Lastly, it should be added that Tossim runs quite slow in large scales. Especially, when simulating the propagation of the entire OS code, it happened to be a big problem. But, when comparing to other WSN simulators, it is still a bit faster.

6.2 Glomosim

Glomosim is a scalable wireless network simulation library, built over the PARSEC simulation environment. Each wireless communication protocol in a protocol stack is defined as a library module, which is developed by using PARSEC, a C-based parallel simulation language. This modularity allows Glomosim users to easily implement new protocols and add them to its library set. In order to gain the maximum performance from the simulator it was implemented on shared memory and distributed memory computers. Glomosim is best to used for Adhoc and mobile network simulations.

6.2.1 Advantages

- Scales in order of ten thousands
- High optimization of simulation
- Modularity of protocol implementation
- Easy to use and develop network models
- Parallel and distributed implemetation

6.2.2 Disadvantages

- Glomosim Api prevent users to use global variables.
- Maximum number of Network partition is fixed to 12.
- Scalability decreased to order of hundreds when mobility is simulated

In the work done for a service discovery project, at Yeditepe University, a simple cross-layer protocol is designed for use of ad hoc network applications. This protocol has necessary algorithms to announce, discover and bind to network services. While providing service access for hosts, the protocol also offers solutions to host addressing, multihop packet routing, session and buffer management, and service naming.

Using the proposed protocol, applications may use non-interactive services available on the ad hoc network. The protocol algorithms are designed as simple as possible to help building an easy to implement network stack and only assume a basic link connectivity from underlying data link layer protocol. Applications using this protocol do not need any other protocols to have transport service or service discovery service.

For the algorithms of the protocol to work in a service-aware manner, a service definition model is designed using extensible markup language (XML). This attribute-value pair holding design is used throughout the algorithms of the protocol to refer specific instances of services offered by the hosts of the ad hoc network. Hosts of the ad hoc network are also represented as attribute-value pair holding XML instances.

Having host and service instances used in algorithms, an easy to enhance protocol is designed. Features like battery awareness or QoS provisioning may be implemented to be embedded into the proposed protocol.

In this work, dynamic access to named non-interactive services in ad hoc networks is studied and a simple cross layer protocol is designed for service discovery and routing. The algorithms of the proposed protocol are implemented in a wireless network simulation software, GloMoSim, for the purpose of algorithm verification and performance evaluation. Some representative applications and scenarios designed out of these applications using the simulation software extensions for the new protocol are also implemented. The results from these experiments have shown that a service aware slim protocol stack implementation is possible for non-interactive service access in mobile ad hoc networks.

6.3 Matlab

The sensor network research community has very recently started using MATLAB as a simulation environment for wireless sensor networks.

MATLAB, short for MATrix LABoratory, is a matrix-based numerical computing environment and a programming language. It was invented in the late 1970's but commercialized only in 1984 (after having being rewritten in C) and is nowadays widely adopted as a tool for data analysis and plotting. Since it is an interpreted language, MATLAB is slower than C or Java when executing programs. MATLAB is also a scripting language, and favors therefore rapid development (at costs of efficiency of execution) and can easily communicate with program components written in other languages (like e.g., Java). Being an interpreted scripting language, is maybe the characteristic that more than others makes MATLAB a promising way to simulate and test algorithms for sensor networks: the user can directly interact with the sensor networks by calling commands on the MATLAB command line. This interaction through the command-line environment is not possible when executing a Java sensor network application: in this case, once the application has been started it can only be controlled through a GUI.

MATLAB can be easily used in conjunction with TinyOS which is the de facto standard operating system for current wireless sensor networks research. Messages from and to the network can be exchanged between the TinyOS Java Tools and the MATLAB environment, allowing for an on-line sensor data analysis and plotting. For instance, MATLAB can be connected to a a physical sensor node (connected through a serial port), to a serialForwarder (The serialForwarder is a TinyOS Java Tool that allows to read packet data from a serial port and forward it over a TCP/IP connection), or to an instance of TOSSIM, the TinyOS "native" simulator.

Scripts and documentation for properly setting the MATLAB environment to work with TinyOS, as well as tutorials and example applications, are available for download at the TinyOS project website (www.tinyos.net).

6.3.1 Advantages

- Modularity of the simulation process
- Compatibility with TinyOS
- Direct interaction with sensor networks
- Importing functions written in C and Java
- Easy to use and develop physical layer models

6.3.2 Disdvantages

- For higher layers of OSI structure, lack of simulations for WSN algorithms
- Slower than C and Java codes
- Scalability

The use of MATLAB for wireless sensor networks simulation and testing is also particularly interesting for researchers working on topics related with the lower layers of the protocol stack, such as physical and data link layers. In this case, MATLAB can be used to model the stochastic behavior of the wireless radio channel and the signaling among the nodes.

One of the most challenging areas of the wireless sensor network design is the positioning of the nodes. Once they are disseminated randomly, they must locate themselves in order to make their readings valuable. Recent localization studies for wireless sensor networks mostly try to estimate the distance between the communicating nodes. In order to get the distance information, the Received Signal Strength Indicator (RSSI) or Time-of-Arrival (TOA) methods are preferred. In both methods, the accuracy of the estimation is affected by the environmental conditions, such as multipath fading, noise, etc. On the other hand, MATLAB toolboxes provide users with modeling those phenomena. Generally speaking, the more precise stochastic process chosen for modeling the radio communication, the more accurate localization information. MATLAB can be used especially for the simulation of the localization algorithms designed for wireless sensor networks due to its accurate random process modeling ability. To sum up, MATLAB can be used for the simulation of the algorithms implemented in physical and data link layers. It gives an idea about the possible results of the real life implementation of proposed algorithms. Therefore, especially localization algorithms that rely on ranging and distance information and radio communication of the wireless sensor networks can be simulated in MATLAB.

In the Computer Architecture, Design and Test for Embedded Systems group (CADTES), the University of Twente, MATLAB is used for the design of a simulator for the localization studies in wireless sensor networks. Named as the LocSim, the simulator has been developed to build a common ground for various algorithms. The main interest of the simulator is the performance of localization protocols rather than the performances of the underlying networking layers.

The simulator is a library that can run with the Simulink of MATLAB where functions are represented as blocks and users can simulate various events by wiring those blocks. All MATLAB functions and Simulink blocks are available for various analysis such as portability across multiple platforms is assured; the tool itself is a standart, etc.

The main focus of the simulator is the precision achieved by the localization algorithms. Furthermore, delays, traffic amount or influence of errors can be measured as well. Also the influence of communication protocol stuck is provided.

Users of the simulator can easily collect data of the same simulation for a variety of parameters. Besides, functions programmed using C can be implemented. Each algorithm is represented as a one block. It is trivial to change, add or remove steps of algorithms, analyze how each stage influences the final results, etc.

6.4 Avrora

Avrora can be used for CPU level sensor-network monitoring. In that sense it is comparable to ATEMU. Although it comes with an insufficient energy model, implementing an accurate one is quite straightforward. If you are to monitor the behavior of a program in the CPU, like memory accesses, register usage, instruction level behavior, this tool is the right tool to chose.

6.4.1 Advantages

- Avrora is a cycle accurate simulation tool for sensor nodes. It simulates microcontrollers ATMega128, ATMega32, ATMega16 and supports mica2 and telos platforms.
- Since Avrora is basically a CPU emulator, it directly runs CPU images, enabling the developer to work on programs written in any language of choice; this is the main advantage of this tool.
- Avrora is written in java. It has a built-in plug-in support at CPU level. This means that programmer can interfere with the execution of the program, can put counters, energy monitors, samplers inside the CPU. Researchers do not need to hack Avrora code, all they need to do is to extend the monitor class and implement necessary interfaces. Direct monitor support includes probes for register and memory accesses and events fired before and after executions of instructions, which also apply to sensor device accesses and send, receive interfaces. Almost anything related to CPU monitoring made easy with this support. When monitors written in java put into the monitors directory, it becomes a part of the environment automatically.

6.4.2 Disadvantages

- This tool can also be used for sensor-network simulation and it supports topology definition. As usual, it can be used with serial-forwarder and other sensor-network analysis and simulation tools. However, since it is a CPU simulator, using it for a simple network simulation may slow down the research process.
- Avrora comes with a poor energy model; the tool itself does not provide a detailed and accurate CPU level energy consumption. However, as mentioned above, constructing an accurate energy model for the purpose at hand is quite simple and such development process won't take a significant time.
- Avrora is almost 50% slower than TOSSIM.

Avrora proved to be a great and easy to use simulation environment for the specific purpose of a research in [YTU]. It was used to obtain data on hit/miss rates of some benchmarks for different cache architectures. It was also used to gather most frequently written bytes to SRAM and registers [1] and also total number of SRAM and register reads/writes.

Since Avrora is a CPU-level simulator and it is a java application, although runs much faster than expected, it does not provide a great simulation speed; this is true especially if you want to record some figures for each instruction executed. So Avrora was used to extract read/write listing for each benchmark into a file and computed figures for cache architectures using a small C program.

Apart from this, it has a great plug-in support (they are called monitors) so that researchers didn't need to hack Avrora code. Coding two monitors (one for register, one for SRAM access information) was all that was done. This took about 100 lines of code. However, there is one exception to this: at one point the simulator code was modified because no other proper way to insert watches on registers (for job [1]) were able to be found.

Good things about this environment can be listed as easy to implement monitors, its speed, ease of use, good support for developers (mailing list and update frequency), its sensor network simulation mode (quite a feature for a CPU level simulator) and easy to understand java code.

When it comes to side-effects of Avrora environment: one might say that it is not well documented for the moment, such that when you want to start coding you have to spend some time wandering around java files and it still lacks some features (the simulator code was modified a bit). However, perhaps one of the weakest features of Avrora is its energy model. For some purposes, an accurate energy model was needed that takes into account energy consumption of SRAM reads/writes that makes a distinction between instructions writing to registers and reading from them (and that also differentiates number of registers being used). For the said research, calculations on generated data required for very accurate energy assumptions.

All in all, Avrora helps to save a lot of time for benchmarking processes.

6.5 OMNeT++ simulator

OMNeT++ is a general purpose discrete event simulator written in C++ that is slightly biased to simulate communication networks. It provides extensive visualization of the network protocol and parallel simulation on many hosts.

Its main advantage is the separation of concerns. The protocol code is implemented in C++ modules. The composition of these modules into complete protocol stacks is done using a simple composition language, which is also used to describe the composition of the network.

In simulations, the researcher wants to simulate a protocol in several variants and under several conditions. OMNet++ supports this using a configuration file, where all these variables are kept. These latter two aspects – specification of the network and the definition of variables – are mixed in e.g. ns-2, making it difficult to find the right variable to change. The initialization file format of OMNeT++ supports by default a batch execution: variables can be grouped into runs that are executed sequentially.

Using the extensive documentation and tutorials, the researcher is able to write his own models fast and in a modular fashion. OMNeT++ is also able to run the simulation fast, by providing two ways of parallel execution. If the simulation is small enough to run on a single computer, several replications can be run on different computers using different seeds – this way, statistically significant results are obtained faster, compared to a sequential execution of the simulation using different seeds. This feature requires the tool Akaroa. If the simulation is too large to fit on a single computer, it can be split in several segments that are distributed on different computers using the MPI library. When the simulation is over, OMNeT++ provides tools to analyze and plot the results.

OMNeT++ is supported for Windows and Linux. The parallel execution features usually require Linux.

The assessment of simulation speed is difficult. Many simulations are Input/Output limited, because they have to write the results into files that are post processed to arrive at simulation results. To avoid I/O operations, one can keep summary statistics in memory and write them to a file once the simulation finishes. This approach requires extensive statistical knowledge to remove the transients and ensure statistical independence of the observed variables. This knowledge is implemented in Akaroa.

The next big bottleneck after the I/O operations are the schedulers. The scheduler of OMNeT++ is heap-based with O(log N) performance, whereas ns-2 uses a calendar scheduler with O(1) performance. However, ns-2 performs comparable to OMNeT++, because the Tcl-binding requires extensive translations of the simulation time from and to strings.

6.5.1 Advantages

- modular protocol development
- clean architecture for instrumenting protocol state and studying cross-layer optimization *Mobility framework*
- consistent separation of variables and network configuration
- scalability, with support for parallel and distributed simulation
- good documentation
- visualization and statistics tools

6.5.2 Disadvantages

- fewer built-in models than e.g. ns-2
- requires knowledge of C++ and STL
- relatively small, but active, community

By design, the OMNeT++ environment provides relatively few built-in modules. Simulation support for wireless and sensor networks is provided by a number of contributed modules, several of which are described below.

6.5.3 Mobility Framework

The evaluation of the performance of a networking protocol is often done using discrete event simulation. The simulation allows the exact reproduction of runs and conditions under which a protocol is tested. However, the reproduction of the environment under which the protocol will be applied is necessarily limited. The physical environment (radio wave propagation, collisions) is an important part of such a performance evaluation, but is also fairly general and can be reused for many simulations. This observation lead to the implementation of the mobility framework [DSH$^+$03], a framework for the discrete event simulator OMNet++ that takes care of the physical environment.

Besides modelling the physical behavior efficiently for large scale simulations (10000 simulated nodes running a CSMA protocol fit into less than 200 MByte of RAM), the mobility framework introduces a blackboard for anonymous communication between modules. It introduced the ability to separate the performance evaluation code – which needs access to protocol internal state variables – and the protocol code itself. The state variables of the simulated protocol are published on the black board and received by performance evaluation modules that take care of the appropriate post processing. This separation allows the re-use of protocols without the need to remove the performance evaluation code first. Furthermore, the blackboard allows cross-layer optimizations without sacrificing the modularity of a layered protocol stack.

To sum up, the mobility framework adds

- radio wave propagation models
- random placement of nodes
- automatic update of links between nodes
- mobility, by default a mobility model is used where nodes move with constant speed.
- blackboard with publish/subscribe interface that allows anonymous exchange of information between different layers of the protocol
- basic MAC protocols like Aloha, CSMA and IEEE 802.11b
- basic network layers like flooding
- possibility to integrate UDP and TCP
- applications with different send behavior

Despite the limited number of implemented protocols, the framework is widely used in the community, especially the blackboard has proved to be a valuable tool. Many simulators for sensor networks implement a comparable approach. The framework is available on `http://mobility-fw.sourceforge.net`. The major drawback is that the user needs a good understanding of C++ and the STL.

6.5.4 NesCT

NesCT is a compiler written by Omer Sinan Kaya that translates TinyOS code to OMNeT++ code such that it can be used with the Mobility Framework. It is available from `http://nesct.sourceforge.net`.

6.5.5 WSN simulation template

The simulation template for wireless mobile sensor networks proposed in [DH03] concentrates on similar aspects like the Mobility Framework. Being inspired by an early version of framework, it models the physical layer, manages connections and makes use of the blackboard. Its main focus is to ease the development of protocol stacks by providing the developer with template code for each layer in the stack. The template code is generated only for layers that the developer specifies and which he has to fill in afterwards.

Further implemented features include:

- Mobility (Random Way Point algorithm by default)

  Each node is responsible for defining its own trajectory and announcing it to the simulator;

- The user can specify if unidirectional or bidirectional links have to be used. Each node can specify and update its transmission range independently;
- Some basic energy management
- Nodes failing identification

  The nodes have different kinds of failing probabilities (individual failures, failures that affect regions of the map, etc.) Maps for area failures can be specified and used. Other maps can easily used for obstacles, fading, etc.

It makes use of C-Macros that make programming easy, but on the other hand can cause portability issues.

6.6 Network Simulator 2 (ns2)

ns2 is a discrete event simulator targeted for network research. It was produced by the VINT project [ns202] and it is the standard *de facto* in the research community to perform wired and wireless network simulations.

- It is written in C++ and its core is composed by few classes that can be binded with OTcl library, an object oriented Tcl-based scripting language. The coupling between C++ and OTcl makes possible to set on the fly many parameters (i.e. transmission power of antennas, MAC attributes, nodes' position in simulated scenario...) and to perform multiple simulations without recompiling the ns2 code: this simplify very much batch simulations.
- It supports many wireless and wired protocols: routing (AODV, DSR, OLSR, DirectedDiffusion, DYMO...), 802.11, 802.11e, 802.15.4, Bluetooth (BlueHoc), GPSR (Greedy Perimeter Stateless Routing), satellite communications, tcp/ip, traffic generators (persistent and pipelined http connections, video streaming of mpeg4...), topology generators, scheduling and queue management algorithms (WFQ, CSFQ, JoBS,...). It has also a set of mobility models (CANU Mobility Simulation Environment, mobility generator tool). You can find a complete reference at [ns2].
- Simulator is mainly composed by a *Scheduler* of *Events* and from *Handler* (i.e. timers or network protocol layers). Networks *Packets* are a particular class of *Events* and communications between entities can be performed by method invocation, Packets passing and Event notification to Handlers.
- It comes with a simple Tcl/Tk tool to perform offline analysis of simulation's traces: Network Animator (NAM). This software shows the topology layout, animate packets delivery and helps to inspect data packets. Unlikely it is is not very useful for wireless simulations since there isn't a good representation for broadcast packets (they are shown as circular propagation waves).
- Its structure is monolithic: there is no support for dynamic libraries or any kind of modularization of code.

The use of ns2 by mean of OTcl scripts is very simple to understand from a beginner: there is a good tutorial and many examples (directory *ns-tutorial*) that drive a user from the creation of a simple scenario with a point to point wired connection between two devices to a very complex wireless network with hundreds of different devices and a complete network stack as well as traffic generators (i.e 802.11 as Mac layer, DRS as routing layer, and FTP as traffic agent between two devices).

While OTcl approach and the use of well tested network solutions make easy to perform batch simulations, the learning curve of an user that wants to implement in ns2 its own solutions (i.e a MAC layer, a routing protocol, a physical propagation model...) is steep. Since documentation is very scarce and often code is not commented, its reuse is often a difficult task. Further, ns2 directory structure and files placement is not so obvious to understand (a standardized approach is missing): for example a developer that wants to add a new network protocol, needs to declare it in *common/packet.h*, change a couple of lines in *trace/cmu-trace.cc* to manage tracing of its packets and declare it in OTcl in *tcl/lib/ns-packet.tcl*.

The use C++ facilities as *Standard Template Library* are not so much encouraged because some bugs in old version of gcc compiler (solved in newer versions).

ns2 has a good energy model: it is possible to set energy consumption of radio when idle, sleep, when it is transmitting or receiving. In last version (2.29) is also possible to se the consumption and duration of radio switching between idle and sleep.

One of the bigger problems in developing solutions for ns2 in wireless scenario is the lack of a good and integrated solution for debugging. Tracing of packets is not adequate when number of nodes grows up and network is wireless (a broadcast can produce several reception). Often a developer is constrained to use *printf* and parse its own dump.

Online documentation on ns2 and mailing lists however cannot yet replace user's experience: many situations which have to be analized when implementing protocols in ns2 require experience to be solved. Here we present an example taken from a real implementation of a MAC protocol layer in ns2 to give an idea of this issue.

When a packet is sent, the class that represent the physical medium delivers it to each device in the network that verify the connectivity constraint: the received power of message is higher than receiving threshold. All destination devices will receive the whole packet as a single event.

In real word, the reception of a packet takes time; for this reason the destination entity needs to bufferize the received packet and start a timer to simulate the process of reception of the packet.

This approach is not so obvious to manage since many situations can happen:

- other packets can arrive in the middle of a packet reception (generating collisions),
- another packet is received such as its received power is greater than old packet's one (so previous packet should be discarded and maybe more powerful packet captured),
- the device can listen only a little part of header and turn radio off if packet is not destinated to it (and do not stay in receiving mode for the whole packet duration).

This example shows how sometime a simple event like packet reception must be modified to reach a real behaviour (we need a timer, a member variable as buffer and some methods to manage the real reception...). All those issues are often difficult to manage for a beginner.

The fine-grain control of events in ns2 help to improve behaviour of simulation but produces the side effect of scalability's reduction: simulations cannot be performed with more than a thousand of devices (or even less if traffic rate of network is high).

A prominent technique used to make ns2 faster and scalable is *staging* [KW03]. This tecnique is summarized by the words: **function caching and reuse** to reduce redundancy of simulator's computation [sns].

Unlikely this enhanced version of ns2 is no more supported and its code is based on version 2.1b9a of ns2 (at the moment of writing of this document current version of ns2 is 2.29).

6.7 On the accuracy of simluation environments

The deployment and the debugging of wireless sensor networks in the real world can be rather hard, in particular if large networks are considered (typically thousands of nodes). There are several popular simulators [Cur] in the wireless sensor network area. They all provide advanced simulation environments. In any case, a simulation is helpful if it can produce a behavior which is as close as possible to the real one. The correct modelisation of components such as collision detection module, radio propagation or MAC protocols is a major issue. Developers focus is typically in the definition of all the aspects of their algorithms and/or protocols, but the interaction with the other layers is often disregarded. The proper setting of simulation parameters and the modelisation of the environment, i.e. mobility schemes, power ranges, connectivity, must be carefully done. Incorrect initial conditions, for example, may lead to unexpected results.

In [CSS02] the authors present a set of measures collected during the simulation of the flooding algorithm in MANETS on different simulators (OPNET, NS-2, GloMoSim). This very simple broadcast algorithm, which simply consists in forwarding to the neighboring nodes every message received for the first time, is a basic building block for several wireless network protocols. The authors have carefully implemented flooding in each simulator paying special attention in setting the same parameters and considering the same scenarios. Surprisingly enough, they have collected very different results, barely comparable.

In [GKW$^+$02] the authors confirmed the “complexity” of flooding also in the WSN environment. They present an empirical study involving over 150 nodes operating at various transmission power settings. The instrumentation in the experiments permits to separate the effects at the various layers of the protocol stack, in order to better understand the behavior of each component. The study confirms again that flooding, can exhibit surprising complexity.

We think that a further development of these studies, in order to better understand which is the reliability of WSN simulators, is an important task. In particular WSN have the nice property that testbeds can be easily deployed (if compared to MANETS). This allow us to compare the results of simulations and testbeds in order to get enriched feedbacks on the quality of nowadays WSN simulators.

7 Testbeds

In order for any platform to be useful, it is necessary to be able to develop and debug software and run tests easily and efficiently. These requirements apply to both the "day-to-day" coding-testing-debugging cycle and the deployment of larger and more complicated testbeds.

This section describes some of the testbeds developed and used in connection with the research platforms described above.

The discussions section presenting used experience address the following two questions:

- debugging: How do we write and debug code on our platform on a day-to-day basis?
- deployment: How do we deploy and work with a larger network?

7.1 DEI Testbed

Table 10: Testbed at the Department of Information Engineering (DEI), University of Padova (Italy)

	day-to-day testbed
name	SIGNET
size	48 EyesIFXv2 nodes
area	$100\,m^2$
topology	grid
location	indoor (room)
topic	localization - MAC - routing
other	nodes are powered and programmed via USB cables and hubs

7.1.1 Description

The testbed is made of 48 *eyesIFX v2.0* nodes. They are hung $60\,\mathrm{cm}$ from the ceiling of a $100\,m^2$–wide room. Nodes are arranged on a grid and powered via USB cables, which are connected to autonomously powered USB hubs. USB hubs are, in turn, connected to a control station (desktop computer), which is used to program the nodes and run debugging functionalities. To this end, we have developed a management software that permits simultaneously programming of bunch of nodes, from one up to the entire network. The software is written in Java and can be easily adapted to any other network, given that nodes are connected to the controlling station via USB. The software is available free of charge, upon request, and it comes with a basic instruction manual.

7.1.2 Discussion

The management software we have developed provides basic functionalities for exchanging data with the nodes. The added value of this software is that it permits to interact with multiple nodes simultaneously and to have a graphical representation of the entire network. Hence, programmer can easily write new plug-ins and link them in the software.

A collection of general–purpose plug–ins is included into the standard distribution of the software provided with the EyesIFXv2 development kit. The plug-in called *Programmer*, is useful when compiling an installing *TinyOS* applications. In a network where all nodes are connected via USB bus to the pc, the main problem is how to know the serial port addresses of each node. Our application and the *Programmer* plug–in allow the user to install applications into the desired nodes without having to look for the right port coordinates, which are automatically detected when the application is launched.

Some debugging functionalities are provided by *ComListener* and *SerialDump* modules. *ComListener* tool listens to ¡the serial ports and keeps log all data received, thus permitting to keep trace of the code execution. *SerialDump* is used to send data to the serial port over the USB bus every time a breakpoint is reached during the execution of the program. The plugin collects this information and prints them in a user–friendly window, possibly with different colors and formats.

Up to now, the testbed has been used for experimenting localization, MAC and routing algorithms. Setting the nodes in Low transmission power mode and low low-noise-amplifier (LNA) gain mode, as well as acting on the digital potentiometer, it is possible to reduce the coverage area in order to have multi–hop connections also in the limited area where the testbed is deployed. Notice that, while the coverage range is determined by both LNA gain and transmission power, the interference level depends on the emitted power only. Therefore, we can experiment different network behavior by varying the setting of transmission power and LNA gain, even if the coverage range remains unchanged.

The first experiments we run on the testbed have revealed the need for calibrating the RSSI circuitry of the nodes. Indeed, we observed non–trivial differences between the RSSI readings of multiple nodes operating in the same conditions. Such calibration errors have a dramatic impact on the performance of localization algorithms that are based on RSSI–ranging.

Our testbed is a useful tool for testing localization algorithms. Reducing the coverage (sicuro?) area with lower transmission power and changing antennas, it could be a good platform also to test routing algorithms in a real multi-hop network.

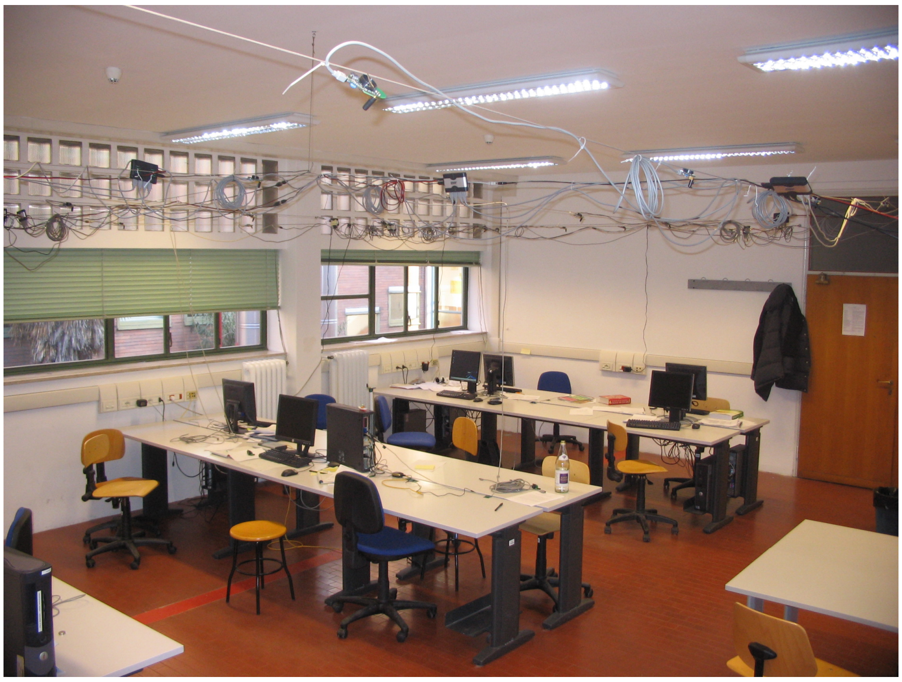
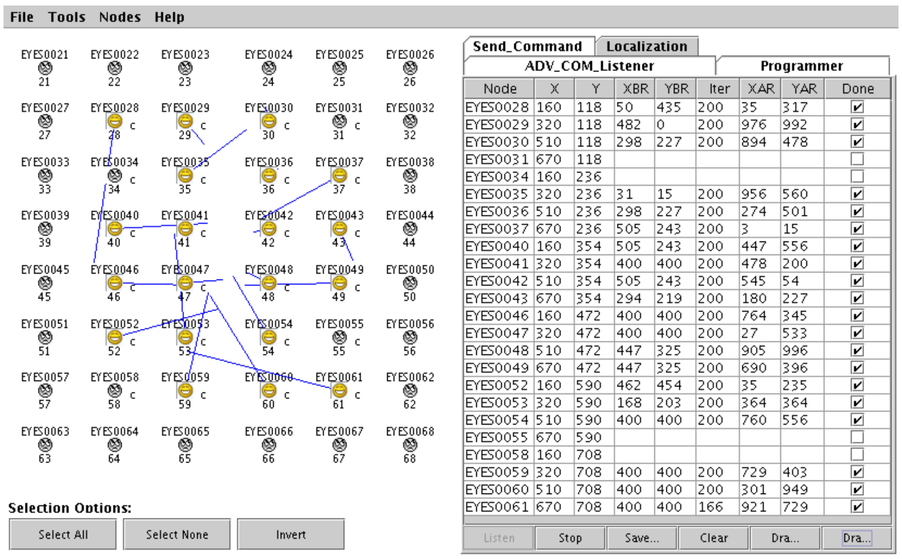


Figure 14: The indoor testbed at DEI.


File Tools Nodes Help
EYES0021 EYES0022 EYES0023 EYES0024 EYES0025 EYES0026
EYES0027 EYES0028 EYES0029 EYES0030 EYES0031 EYES0032
EYES0033 EYES0034 EYES0035 EYES0036 EYES0037 EYES0038
EYES0039 EYES0040 EYES0041 EYES0042 EYES0043 EYES0044
EYES0045 EYES0046 EYES0047 EYES0048 EYES0049 EYES0050
EYES0051 EYES0052 EYES0053 EYES0054 EYES0055 EYES0056
EYES0057 EYES0058 EYES0059 EYES0060 EYES0061 EYES0062
EYES0063 EYES0064 EYES0065 EYES0066 EYES0067 EYES0068
Selection Options:
Select All
Select None
Invert
Send_Command
Localization
ADV_COM_Listener
Programmer
Listen
Stop
Save...
Clear
Dra...
Dra...

Node	X	Y	XBR	YBR	Iter	XAR	YAR	Done
EYES0028	160	118	50	435	200	35	317	✓
EYES0029	320	118	482	0	200	976	992	✓
EYES0030	510	118	298	227	200	894	478	✓
EYES0031	670	118						☐
EYES0034	160	236						☐
EYES0035	320	236	31	15	200	956	560	✓
EYES0036	510	236	298	227	200	274	501	✓
EYES0037	670	236	505	243	200	3	15	✓
EYES0040	160	354	505	243	200	447	556	✓
EYES0041	320	354	400	400	200	478	200	✓
EYES0042	510	354	505	243	200	545	54	✓
EYES0043	670	354	294	219	200	180	227	✓
EYES0046	160	472	400	400	200	764	345	✓
EYES0047	320	472	400	400	200	27	533	✓
EYES0048	510	472	447	325	200	905	996	✓
EYES0049	670	472	447	325	200	690	396	✓
EYES0052	160	590	462	454	200	35	235	✓
EYES0053	320	590	168	203	200	364	364	✓
EYES0054	510	590	400	400	200	760	556	✓
EYES0055	670	590						☐
EYES0058	160	708						☐
EYES0059	320	708	400	400	200	729	403	✓
EYES0060	510	708	400	400	200	301	949	✓
EYES0061	670	708	400	400	166	921	729	✓

Figure 15: The management Software interface.

7.2 TWIST Testbed Architecture

TWIST [HKWW06] is a scalable and flexible testbed architecture for indoor experimentation with wireless sensor networks. The design of TWIST is based on an analysis of typical and desirable use-cases. It provides basic services like node configuration, network-wide programming, out-of-band extraction of debug data and gathering of application data, while introducing several novel features:

- Support for experiments with heterogeneous node platforms.
- Active node power supply control, enabling easy transition between USB-powered and battery-powered experiments, dynamic selection of topologies as well as controlled injection of node failures into the system.
- Hosting of both flat and hierarchical WSN configuration, due to a layer of "super nodes" that on one hand form a part of the testbed infrastructure but can also play a role as elements of the sensor network.

The self-configuration capability, the use of hardware with standardized interfaces and open-source software makes the TWIST architecture scalable, affordable, and easily replicable. Figure 16 depicts the hardware architecture of TWIST.

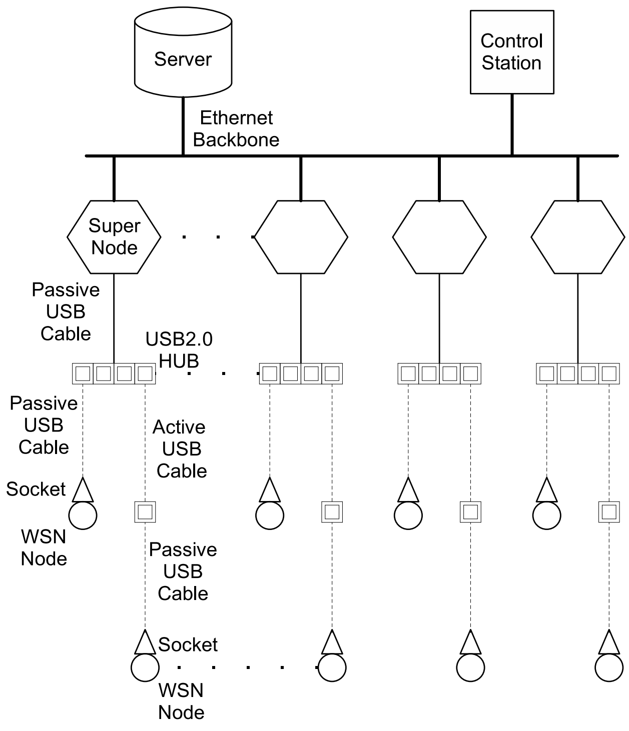


Figure 16: Hardware architecture of the TWIST testbed

7.2.1 TWIST Instance at the TKN Building

The local instance of TWIST spans three floors of the Telecommunication Networks Group office building at TU Berlin. Currently, there are about 100 fixed locations for nodes with known positions and additional 100 free slots on the USB hubs. The instance uses 37 NSLU2, 53 USB hubs and about 600 m of USB cables. The NSLU2 communicate over wired Ethernet, but alternatively they can use a USB-to-WLAN adapter (attached to the free USB port of the NSLU2) to establish a wireless backbone network.

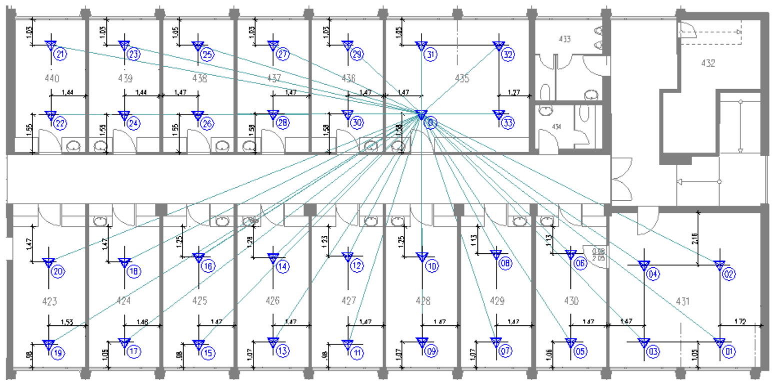


Figure 17: Euclidean distance between a base station and nodes on the $4^{(th)}$ floor

The placement of the nodes is kept as regular as possible, with at least two sockets in each room [26]. This provides at least two fixed and known node locations per room, giving insight into the spatial distribution of the measured values like light and temperature and allowing for cross-calibration of sensors of the same type. The node placement is shown in Figure 17. This figure also shows the exact location of the sockets, measured after the installation. The sockets are placed 1 m and 4 m from the window, and 1.5 m from each wall. For the large rooms a different arrangement is used, with four sockets per room, in order to keep the inter-node distances comparable. Any sparser, more random placement of the nodes can be emulated using the power supply control feature of the testbed.

The USB cabling is neatly routed using cable channels mounted along the walls and on the ceiling. To connect a node to a socket, it is simply connected it to the USB cable and affixed to the cable channel as shown in Figure 18. This also means that the nodes are mounted about 4 cm from the ceiling.

[26]The testbed is currently being expanded to double its capacity, resulting in four sockets in each room

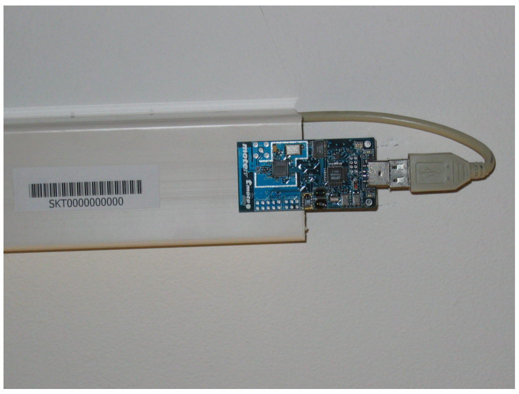


Figure 18: Tmote Sky node connected to a TWIST socket

Table 11: TWIST Testbed Instance at the Telecommunication Networks Group of TU Berlin

name	TWIST
size	100 multiplatform sockets: eyesIFXv2.1, TelosA, Tmote Sky, Tmote Invent
area	1200 m^2
topology	(see Figure 17)
location	indoor (room)
topic	general application
other	binary node power supply control and distributed testbed processing

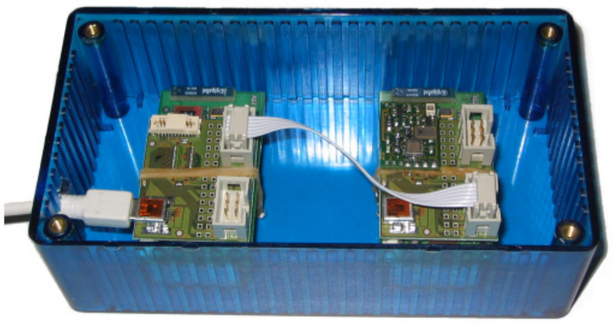


Figure 19: DNS node demonstrator [HW05]. The DSN node is situated on the left side, the target node with a mounted sensor board on the right side.

7.3 ETHZ Testbed

A permanent setup of 19 BTnodes has been deployed during summer 2005 at the Computer Engineering and Networks Laboratory at the ETH Zurich. This testbed is intended as a demonstration platform for implementing and validating the Deployment-Support Network (DNS), a new methodology for the deployment, debug and monitoring of wireless sensor networks. The main idea consists in physically attaching DSN nodes via a programming and debugging cable to target sensor networks devices like sensor nodes, as shown in figure 19. Currently, the DSN supports only BTnodes as target devices, but any other Atmel AVR target, like i.e. Berkeley/Crossbow Motes could also be easily added. The DSN nodes can autonomously interconnect through Bluetooth ad-hoc networking and thus form an autonomous, wireless support network. This support network provides connectivity from one or more host controllers (e.g., a desktop or laptop pc) to the target nodes, allowing remote programming, debugging, monitoring and control.

Figure 20 shows the placement of the 19 BTnodes that form the ETHZ testbed. The network is deployed on the G floor of the ETZ building at ETH Zurich, throughout the rooms and corridors of the Computer Engineering and Networks Laboratory. Currently, all nodes are powered over USB to ensure long-term network availability. However, the network can be easily extended with battery-operated nodes.

Interested readers can retrieve more information about the current test implementation of the Deployment Support Network (called *JAWS*), through the BTnode project web site [BTna]. A simple graphical user interface (written in java) is available for visualizing network topology and to access and control the DNS nodes[27].

[27]Available on-line at: `http://www.btnode.ethz.ch/Projects/Jaws`

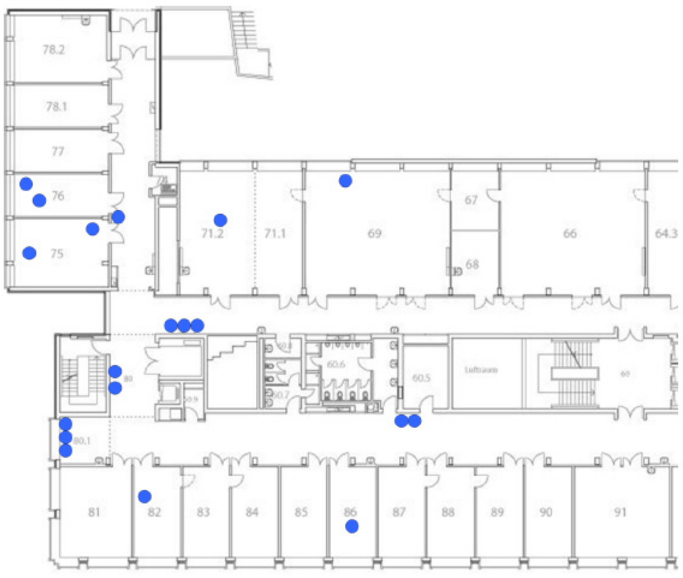


Figure 20: Placement of the 19 BTnodes that form the ETHZ testbed [HW05]. The network is deployed throughout the G floor of the ETZ building at ETH zurich.

7.4 YTU Testbed

7.4.1 Description

In YTU, a number of test-beds were constructed to support ongoing research projects. Most of the test-beds were one-hop or two-hop, simple topologies of either mica2 or t-mote sky nodes. A one-hop test-bed was constructed with 2 mica2 nodes for testing SQS query middleware within in a system designed for detecting human traffic in and out of a room. In another recent localization research, a two-hop test-bed consisting of five t-mote sky nodes was constructed. Apart from those mentioned, numerous simple one-hop test-beds were utilized for testing research code on a real platform.

7.5 SICS Testbed

7.5.1 Description

SICS currently does not have a permanent testbed, but rather a quite large number of nodes (around 60) of different types, namely TelosSky, ESB2 and ECR-like nodes from Scatterweb. Some of the nodes are tailored for specific real-world applications we are developing and hence will no longer be at SICS after deployment of the corresponding network.

Software development is done mostly under the host simulation environment (see Section 4.3.6) and the COOJA Contiki OS Java Simulator, a simulator that enables simulation of networks of Contiki nodes that is currently under development. The COOJA system is both flexible and extensible in that many levels of the system can be changed or replaced: sensor node platforms, operating system software, radio transceivers, and radio transmission models. The simulator is implemented in Java, making the simulator easy to extend for users, but allows sensor node software to be written in C by using the Java Native Interface. Furthermore, the sensor node software can be run as compiled native code for the platform on which the simulator is run. We are also developing a sensor node emulator for ESB/2 hardware that can be plugged into the simulator enabling emulation of actual ESB hardware.

Download of code on sensor networks is done using the serial interface or over the radio of the number of nodes is large. Contiki supports modular run-time reprogramming of selected parts of the system over the radio. Debugging on the ESBs can be performed using the three LEDs on the board, or the serial interface. It is also possible to debug a single node using the on-board JTAG-connector, but we have only very limited experience with this.

7.5.2 Discussion

While a testbed would be useful for certain scenarios and some of our application development, we have not yet invested the time to build such a network but instead are building special-purpose networks for specific experiments. We have also decided to work on the new COOJA Contiki simulator to replace the non-deterministic network simulator we previously used.

8 Conclusions

In this report, we have presented a critical survey of several commonly used research platforms for wireless sensor networks. The report presents both a catalog of feature data and, more importantly, a discussion of expert users' experience with the platforms.

In addition to providing a useful collection of information, we have also used this work to identify research gaps and issues related to the research platform. As a result, the report highlights the importance of the development environment and the effectiveness of the development (program-test-debug) and deployment tools.

An important way to improve the ease-of-use of a platform is to increase the number of users and the variety of purposes for which a platform is being used. Moreover, the collective experience of a diverse user community provides the necessary technical basis for standardization and system integration.

Providing information that makes researchers aware of the diversity of available platforms and helps them select the platform that is appropriate for their needs – as in this report – contributes to these goals.

References

[Amba] Ambient System B.v. http://www.ambient-systems.net/ambient/index.htm.

[Ambb] Ambient system products. http://www.ambient-systems.net/ambient/products-system.htm.

[BDH+04] J. Beutel, M. Dyer, M. Hinz, L. Meier, and M. Ringwald. Next-generation prototyping of sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04:)*, pages 291–292, New York, NY, USA, 2004. ACM Press.

[BDMT05] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable Topology Control for Deployment-Support Networks. In *Fourth International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, pages 359–363, April 2005.

[Beu05] Jan Beutel. Real-world sensor networks: Experiences in design and deployment. Technical report, Summer School on Wireless Sensor Networks and Smart Objects, Schloss Dagstuhl, 2005.

[BKM+04] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. Prototyping Wireless Sensor Network Applications with BTnodes. In *1st European Workshop on Wireless Sensor Networks (EWSN)*, LNCS, pages 323–338, Berlin, Germany, January 2004. Springer-Verlag.

[BMR90] S. Baruah, A. Mok, and L. Rosier. Preemptive scheduling hard-realtime sporadic tasks on one processor. In *Proceedings of the Real-Time Systems Symposium*, pages 182–190, December 1990.

[BO06] Sebnem Baydere and Others. Applications and application scenarios. Technical Report EW-T311-YTU-001-04, Embedded WiSeNts, 2006.

[BTna] BTnode: A Distributed Environment for Prototyping Ad Hoc Networks. (Project Website: www.btnode.ethz.ch).

[BTnb] BTnode rev3: Users Survey. (Available at: www.btnode.ethz.ch/pub/uploads/Documentation/20030905_btnode_v3survey.pdf).

[CSS02] David Cavin, Yoav Sasson, and Andr&#233; Schiper. On the accuracy of manet simulators. In *POMC '02: Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 38–43, New York, NY, USA, 2002. ACM Press.

[Cur] David Curren. A survey of simulation in sensor networks. www.cs.binghamton.edu/ kang/cs580s/david.pdf.

[DGV04] A. Dunkels, B. Grnvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[DH03] S. Dulman and P. Havinga. A simulation template for wireless sensor networks. In *Proceedings of the 2003 IEEE International Symposium on Autonomous Decentralized Systems (ISADS'03)*, Pisa, Italy, April 2003.

[DSH+03] W. Drytkiewicz, S. Sroka, V. Handziski, A. Köpke, and H. Karl. A mobility framework for omnet++. In *3rd International OMNeT++ Workshop, at Budapest University of Technology and Economics, Department of Telecommunications Budapest, Hungary*, January 2003.

[DSV05] A. Dunkels, O. Schmidt, and T. Voigt. Using protothreads for sensor node programming. In *Proc. of the Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, Stockholm, Sweden, June 2005.

[Dun03] 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.

[DvHHJ03] S. Dulman, L. van Hoesel, P. Havinga, and P. Jansen. Data centric architecture for wireless sensor networks. In *Proceedings of the ProRISC Workshop*, November 2003.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[GKW+02] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. Complex behavior at scale: An experimental study of low-power wireless sensor networks, 2002.

[GLCB06] David Gay, Philip Levis, David Culler, and Eric Brewer. nesC 1.2 Language Reference Manual. Online, 2006. http://nescc.cvs.sourceforge.net/*checkout*/nescc/nesc/doc/ref.pdf.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[HC02] Jason L. Hill and David E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

[HDJH04] T. Hofmeijer, S. Dulman, P. G. Jansen, and P. J. M. Havinga. Ambientrt - real time system software support for data centric sensor networks. In *2nd Int. Conf. on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Melbourne, Australia*, pages 61–66, Los Alamitos, California, December 2004. IEEE Computer Society Press.

[HKWW06] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. Twist: A scalable and reconfigurable testbed for wireless indoor experiments with sensor network. In *Proc. of the 2nd Intl. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality, (RealMAN 2006)*, Florence, Italy, May 2006.

[HPH$^+$05] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, and D. Culler. Flexible hardware abstraction for wireless sensor networks. In *Proceedings of Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, Istanbul, Turkey, February 2005.

[HPHS04] V. Handziski, J. Polastre, J. H. Hauer, and C. Sharp. Flexible hardware abstraction of the ti msp430 microcontroller in tinyos. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 277–278, Baltimore, MD, USA, November 2004. ACM Press.

[HSW$^+$00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[HW05] D. Hobi and L. Winterhalter. Large-scale Bluetooth Sensor-Network Demonstrator. Master's thesis, ETH Zurich, October 2005.

[JMHS03] P. G. Jansen, S. J. Mullender, P. J. M. Havinga, and J. Scholten. Lightweight EDF scheduling with deadline inheritance. Technical report TR-CTIT-03-23, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, May 2003. `http://www.ub.utwente.nl/webdocs/ctit/1/000000c6.pdf`.

[KHHK04a] 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. Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN)*, Technical Report TKN-04-001 of Technical University Berlin, Telecommunication Networks Group, pages 41–45, Berlin, Germany, January 2004.

[KHHK04b] A. Köpke, V. Handziski, J.-H. Hauer, and H. Karl. Structuring the information flow in component-based protocol implementations for wireless sensor nodes. In *Work-in-Progress Session of the First European Workshop on Wireless Sensor Networks (EWSN'04)*, January 2004.

[KL01] Oliver Kasten and Marc Langheinrich. First Experiences with Bluetooth in the Smart-Its Distributed Sensor Network. Workshop on Ubiqitous Computing and Communication, Conference on Parallel Architectures and Compilation Techniques (PACT) 2001, October 2001. Workshop proceedings available at http://research.ac.upc.edu/pact01/pucc.htm.

[KW03] Emin Gn Sirer Kevin Walsh. Staged simulation for improving the scale and performance of wireless network simulations. *In Procedings of the Winter Simulation Conference, New Orleans, LA*, December 2003.

[Lev06] Phil Levis. Programming TinyOS. Online, 2006. `http://csl.stanford.edu/~pal/pubs/tinyos-programming-1-0.pdf`.

[LGH+05] P. Levis, D. Gay, V. Handziski, J.-H.Hauer, B.Greenstein, M.Turon, J.Hui, K.Klues, C.Sharp, R.Szewczyk, J.Polastre, P.Buonadonna, L.Nachman, G.Tolle, D.Culler, and A.Wolisz. T2: A second generation os for embedded sensor networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin, November 2005.

[LH05] Koen Langendoen and Gertjan Halkes. Energy-efficient medium access control. In Richard Zurawski, editor, *Embedded Systems Handbook*, chapter 34. CRC Press, 2005.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Simulating large wireless sensor networks of tinyos motes. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[LMG+04] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, San Francisco, California, USA, mar 2004.

[MKL+04] 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.

[Mot] Moteiv. Web page.

[NCC] NCCR-MICS : National Center of Competence in Research - Mobile Information and Communication Systems. (Project Website: `www.mics.org`).

[ns2] Contributed code. http://nsnam.isi.edu/nsnam/index.php/Contributed_Code.

[ns202] The vint project. 2002.

[NT06] L. Negri and L. Thiele. Power-delay tradeoffs in bluetooth scatternets. In *Proceeding of the 3rd European Workshop on Wireless Sensor Networks (EWSN 2006)*, Zurich, Switzerland, February 2006.

[Par] Particle Computer GmbH. Company Website: www.particle-computer.net/.

[PHC04] J. Polastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pages 95–107, New York, NY, USA, 2004. ACM Press.

[PO06] Marcelo Pias and Others. Vertical system functions. Technical Report EW-T313-UCAM-001-06, Embedded WiSeNts Project, 2006.

[Pro] The RUNES Project. Web page. http://www.ist-runes.org/.

[RR05a] M. Ringwald and K. Roemer. BitMAC: A Deterministic, Collision-Free, and Robust MAC Protocol for Sensor Networks. In Erdal Cayirci, Sebnem Baydere, and Paul Havinga, editors, *Proceedings of the Second IEEE European Workshop on Wireless Sensor Networks and Applications (EWSN 2005)*, Istanbul, Turkey, February 2005.

[RR05b] Matthias Ringwald and Kay Römer. Monitoring and Debugging of Deployed Sensor Networks. 2. GI/ITG KuVS Fachgespräch Systemsoftware für Pervasive Computing, Arbeitsberichte des Instituts für Informatik, vol. 38/5, October 2005.

[RSV$^+$05] Hartmut Ritter, Jochen Schiller, Thiemo Voigt, Adam Dunkels, and Juan Alonso. Experimental Evaluation of Lifetime Bounds for Wireless Sensor Networks. In *Proceedings of the Second European Workshop on Sensor Networks (EWSN2005)*, Istanbul, Turkey, January 2005.

[RYR06] M. Ringwald, M. Yucel, and K. Römer. Demo Abstract: Interactive In-Field Inspection of WSNs. In *Adjunct Proceedings of the 3rd European Workshop on Wireless Sensor Networks (EWSN 2006)*, Zurich, Switzerland, February 2006.

[Sca] Scatterweb. Web page. http://www.scatterweb.com/.

[SM00] Richard M. Stallman and Roland McGrath. *GNU Make: A Program for Directing Recompilation, for Version 3.79*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, 2000.

[Sma] Smart-Its Project. (Project Website: `www.smart-its.org`).

[sns] http://www.cs.cornell.edu/people/egs/sns/.

[SO06] Silvia Santini and Others. System architectures and progamming models. Technical Report EW-T314-ETHZ-001, Embedded WiSeNts Project, 2006.

[Tin03] TinyOS 1.x Tutorial. Online, 2003. `http://www.tinyos.net/tinyos-1.x/doc/tutorial/`.

[vDL03] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 171–180, New York, NY, USA, 2003. ACM Press.

[vHH06] Lodewijk van Hoesel and Paul Havinga. Design aspects of an energy-efficient, lightweight medium access control protocol for wireless sensor networks. *INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS*, pages 1–21, 2006.

[YHE02] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1567–1576, June 2002.

[ZO06] Andrea Zanella and Others. Paradigms for algorithms and interations. Technical Report EW-D312-DEI-001-03, Embedded WiSeNts Proejct, 2006.

[st06] Fredrik sterlind. A Sensor Network Simulator for the Contiki OS. Technical Report T2006-05, SICS – Swedish Institute of Computer Science, February 2006.