Energy-efficient Management of Heterogeneous Wireless Sensor Networks

Markus Anwander, Gerald Wagenknecht, Torsten Braun

Abstract

Heterogeneous wireless sensor networks (WSN) consist of a number of several heterogeneous sensor nodes. Typically, different sensor node platforms are equipped with different types of radio modules and, therefore, cannot communicate with each other. To interconnect the different sensor sub-networks, we propose a wireless mesh network (WMN) to operate as a backbone. For each sensor sub-network, a sensor node gateway is connected to a mesh node.

We have chosen Contiki as an appropriate operating system to support dynamic and energy-efficient reconfiguration as well as code updates of different sensor node platforms. Contiki supports preemptive multithreading, inter-process communication and dynamic runtime linking of standard Executable Linkable Format (ELF) files. The lightweight and compact base system is highly portable to other sensor platforms. Contiki also offers a minimal TCP/IP stack for communication.

The management architecture consists of the following infrastructural elements: a management station, a number of management nodes and a high number of heterogeneous sensor nodes. The supported management tasks are: monitoring the WSN, configuration of the WSN, code updates and managing sensor data. The management station executes all management tasks. Management nodes are implemented as wireless mesh nodes.

Categories and Subject Descriptors C.2.1 [*COMPUTER-COMMUNICATION NETWORKS*]: Network Architecture and Design; C.2.2 [*COMPUTER-COMMUNICATION NETWORKS*]: Network Protocols; D.2.9 [*SOFTWARE ENGINEERING*]: Management; D.4.4 [*OPERATING SYSTEMS*]: Communications Management; K.6.3 [*MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS*]: Software Management

General Terms Algorithms, Design, Management

Keywords wireless sensor network, WSN management architecture, code distribution protocol, operating systems, distributed system

1. Introduction

A heterogeneous wireless sensor network (WSN) consists of a number of several sensor nodes. A wireless mesh network (WMN) operates as a backbone. With a management station the WSN can be maintenance. An appropriate WSN management architecture has been designed. As operating system Contiki runs on the sensor nodes. It supports preemptive multithreading, inter-process communication and dynamic runtime linking of standard Executable Linkable Format (ELF) files. Program modules can be updated and loaded at runtime. Two integrated communications stacks (Rime and uIP) support the communication between the sensor nodes and to the mesh nodes.

This report is structured as follows: the second Section introduces related work in the topics of Contiki and code updating mechanisms. The third Section gives an overview of the management architecture, which has been defined in [1]. The fourth Section covers management middleware on the mesh nodes and the sensor nodes. In Section 5 the sensor nodes platforms and the operating system Contiki is presented, especially the communication issues. Section 6 gives an outlook and points out the next tasks of the project.

2. Related Work

The technical report [1] defines a management and code distribution framework for wireless sensor networks. This includes the identification of appropriate sensor node platforms and an appropriate sensor operating system. Furthermore, it includes the definition and implementation of a management architecture for WSNs. Management tasks, which have to be supported are: monitoring the WSN, configuration of the WSN, performing code updates and managing sensor data.

A secure remote management and software distribution framework for wireless mesh networks (WMNs) [2] has been developed at the University of Bern. It is an application to reconfigure and to update mesh nodes in a WMN and will be used for our work. It is a fully distributed management solution without a central management node. Configuration errors and corrupt software updates can be handled by the system.

Contiki [3] is a dynamic operating system designed for portability. It is written in C and supports over 14 embedded systems and 5 CPUs. A very small TCP/IP stack [4] has been implemented. Protothreads [5], a novel thread-like construct on top of the event-driven kernel reduces the complexity of event-driven programs by removing flow-control state machines. Contiki also supports preemptive multi-threading, inter-process communication and dynamic run-time linking of standard ELF files. Program modules can be updated and loaded at run-time. Contiki and the network simulator COOJA [6] for Contiki are open source and run under BSD license.

Trickle [7] is an algorithm for propagating and maintaining code updates in WSNs borrowing techniques from the epidemic, gossiping, and wireless broadcast. Sensor nodes periodically broadcast a code summary to local neighbors but stay quiet if they have recently heard a summary identical to theirs. When a node hears an older summary than its own, it broadcasts an update. Instead of flooding a network with packets, the algorithm controls the transmission rate so that each node hears a small trickle of packets, just enough to stay up to date.

Multi-hop Over-the-Air Programming (MOAP) [8] is a code distribution mechanism specifically targeted for Mica motes. It focuses on energy-efficient and reliable code distribution. The Ripple dissemination protocol transmits the data on a neighborhood-by-neighborhood basis. In fact, this is a recursive extension of a single hop mechanism to multi-hop. The source announces the new code to its one-hop neighbors, which may answer with a subscription. The code image is sent to the subscribers. If the whole image is received at the subscribers, some of them become the new sources and further announce the new code. A node detecting a lost segment requests a local retransmission from its source. A drawback of the scheme is that all nodes have the same entire image. This results in problems when using a sensor network with a heterogeneous hardware base. Furthermore, selective node updates are an open issue.

FlexCup [9] is a flexible code updating algorithm that minimizes the energy consumed on each sensor node for the installation of arbitrary code changes. It allows exchanging only the components of a program that have actually changed. Reijers and Langendoen [10] use a diff-like approach to compute a diff script that transforms

the installed code image into a new one. Jeong and Culler [11] presents an incremental network programming protocol, which uses the Rsync algorithm [12] to find variable-sized blocks that exist in both code images and then only transmits the differences. However, both of these approaches just compare the code image using very limited knowledge about the application structure, if at all.

3. Management Architecture

This section describes a typical scenario of a heterogeneous wireless sensor network. Based on the typical management tasks, the defined management architecture [1] is presented briefly. The management architecture contains the following structural elements: a management station, some mesh nodes as management nodes, sensor node gateways plugged into a mesh node, and the different sensor nodes.

3.1 Sensor Gateway Connection

A heterogeneous wireless sensor network consists of different types of sensor nodes, which might measure different data and perform different tasks. To operate such a (sub)network the following devices are required: one management station, several mesh nodes and a comparatively high number of heterogeneous sensor nodes. A possible scenario is shown in Figure 1.

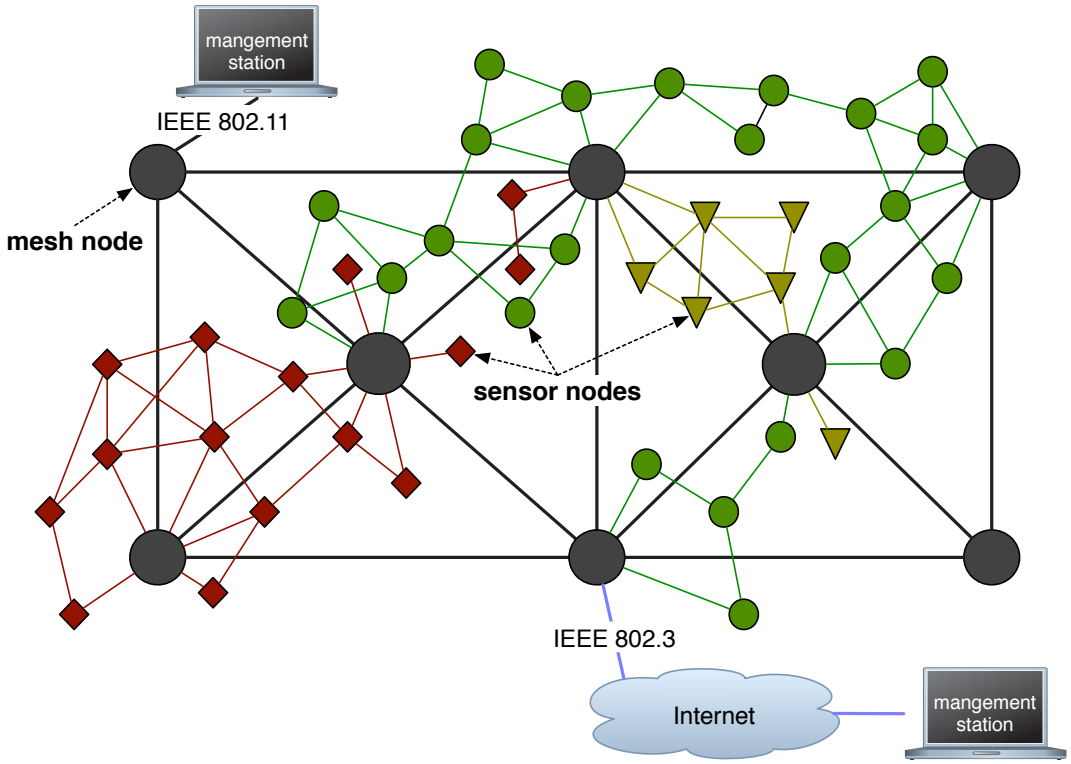

Figure 1. A possible management scenario

The sensor nodes might have different sensors for monitoring the environment. All sensor nodes of one type are able to communicate with each other and build a sensor subnet. Many existing sensor platforms have different radio modules and are thus not able to communicate with each other. The different subnets are interconnected by wireless mesh nodes. They provide interfaces for different sensor subnets and act as gateways. Besides the inter-subnet communication, the mesh nodes perform management tasks. Each mesh node is responsible for one or more subnets. Control of the sensor nodes is done via the management station. The management station is connected to the mesh nodes via Ethernet or via IEEE 802.11 WLANs.

From the management point of view there are several tasks required to manage a WSN and its sensor nodes. In general we can divide these into four areas:

- Monitoring the WSN and the sensor nodes,
- (Re)configuring the WSN and the sensor nodes,
- Updating the sensor nodes,
- Managing the sensor data.

The monitoring task requires that all sensor nodes in the several subnets are displayed at the management station with all necessary information. This includes sensor node hardware details (e.g. chip, transceiver), sensor node software details (e.g. operating system versions), and dynamic properties (e.g. battery). The node ID and other static information is sent when a sensor node joins the network. Additionally the management station may query sensor nodes. The (re)configuration task includes sensor node configuration and network configuration. Code distribution mechanisms perform the operating system or the application updates. Mechanisms to handle incomplete, inconsistent and failed updates have to be provided.

3.2 Management Station

The management station shown in Figure 2 contains of two parts: a laptop or remote workstation with a web browser as user interface and the management system for WMNs. The communication between both is done via HTTPS. The management system for WMNs contains a small Linux distribution including all required applications, especially a HTTP server supporting PHP and an authentication system. The HTTP server maintains several modules to handle the requests and transmit them to mesh nodes, sensor nodes or CFEngine [13]. The authentication system assures that only authorized users have access to the management functions. It is also possible to integrate the Authentication and Authorization Infrastructure (AAI) [14] for an advanced authentication system. Communication with a mesh node is done via HTTPS over TCP. For data transmission within the mesh network, CFEngine is used. The WSN monitor is responsible for monitoring the whole network. It shows the mesh nodes with their subordinate sensor nodes including all available information of the sensor nodes. The user may request information from a single sensor node. The WSN configurator is responsible for the configuration of the WSN. The code update manager distributes the uploaded image via CFEngine. It shows the available program versions and performs the updating process.

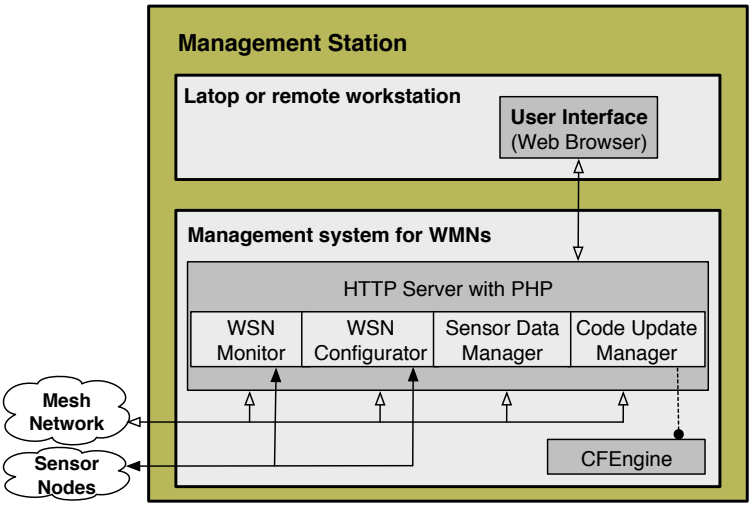

Figure 2. Management station architecture

3.3 Mesh Node

The WSN manager running on the mesh nodes consists of the following components: program version database, WSN information database, sensor database, WSN monitor module, WSN configurator module, and code update manager module (shown in Figure 3). The sensor database stores all measured data as tuples (node ID, sensor ID, value, timestamp). The WSN information database consists of all infra-structural data. Each entry contains ID, property ID, value, and timestamp. The properties are e.g. chip, transceiver, and battery status. The program version database stores the versions of all available programs containing program ID, version, target platform, timestamp, and link to the image. The CFEngine is responsible for distributing the databases within the mesh network. The WSN monitor module connects to the WSN information database and to the sensor data database for responding the requests from the management station. It writes sensor and node data into the databases. The WSN configurator module connects to the WSN information database to read and write data. It further queries the sensor node properties and sends commands. The code update manager module stores newly uploaded images in the program version database. It also updates the sensor nodes. It includes methods to reduce the distributed code (differential patch, compression).

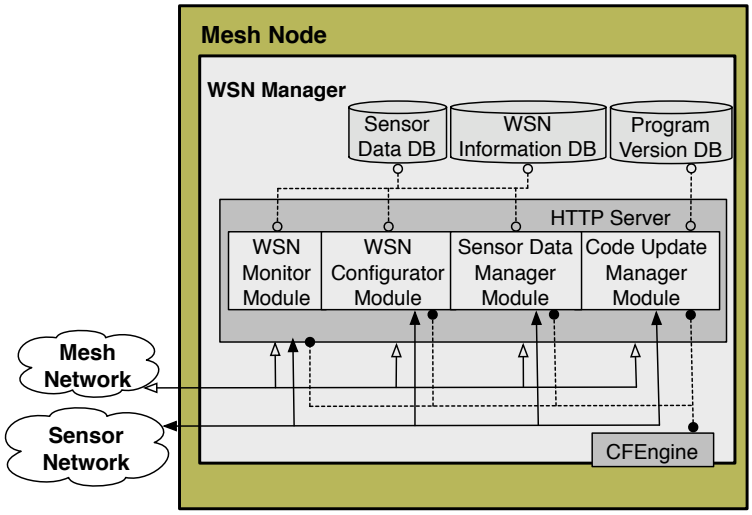

Figure 3. Mesh node architecture

3.4 Sensor Node

As shown in Figure 4, the sensor node manager handles the management tasks. It consists of sensor node monitor, sensor node configurator, sensor data sender, and code updater. The sensor node monitor sends the requested values to the mesh node. The sensor node configurator executes the configuration requests and notifies the mesh node. The code updater receives the image of the application or operating system (differential patch, compressed, or uncompressed) and performs the update. It confirms the success of the update to the mesh node.

4. (Sensor Node) Middleware

In [1] the management scenario, the management task, the management architecture, and the management protocols are described. The management middleware consists of a WSN manager running on the mesh node and a sensor node manager running on the sensor nodes. In this section we describe the middleware.

4.1 WSN Manager

The WSN manager consists of three databases and four program modules. The databases are: the program version database, the WSN information database, and the sensor data database. The program modules consist

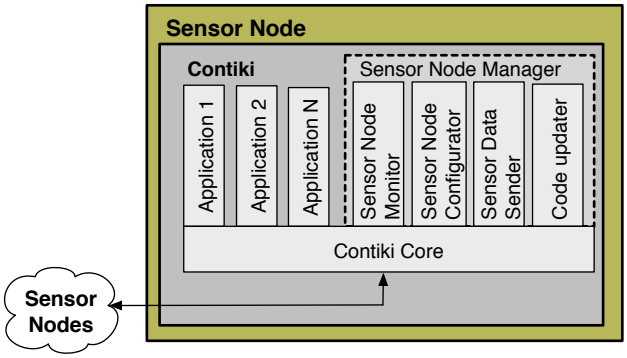

Figure 4. Sensor Node Manager

of the WSN monitor module, the WSN configurator module, the sensor data manager module, and the code update manager module. This is shown in Figure 3. The sensor data database stores all data measured by the sensors. The database is SQL-based and contains the following values: ID of the sensor node, ID of the sensor (property), value, and timestamp. To access the database the following functions are available:

- getSensorData(sn_id, prop_id, time interval) ← value[]: This function returns a list of all sensor data from a given sensor (prop_id) on a given sensor node (sn_id) within a time interval (time interval) .
- getCurrSensorData(sn_id, prop_id) ← value: This function gives the last value of a given sensor (prop_id) on a given sensor node.
- getAllSensorData()← value[]: This function dumps the database and gives a list of all values of all sensors on all sensor nodes.
- insertSensorData(sn_id, prop_id, value[]): This function inserts a value (or a list of values) from a given sensor (prop_id) on a given sensor node (sn_id) into the database.

The WSN information database stores all data about the sensor nodes and the WSN. This database is also SQL-based and contains the following values: ID of the sensor node, ID of the property, value, and timestamp. The properties are: node.type, node.role, chip.name, transceiver.name, battery.name, battery.capacity, battery.curr value, memory.name, memory.size, memory.free space, os.name, os.version, os.last update, routing table, neighbors.ID[], ip address, hosting mesh node.ID, and many more. To access the database the following functions are available:

- getAllMeshNodes()← mn id[]: This function returns a list of all available mesh nodes (mn id).
- getAllSensorNodes()← (sn_id, mn id)[]: This function returns a list of all available sensor nodes (sn_id) including the responsible mesh node (mn id).
- getAllOwnSensorNodes(mn id) ← sn_id[]: This function returns al list of all available sensor nodes (sn_id[]) to a given mesh node (mn id).
- getSensorNodeProperties(sn_id) ← prop_id[]: This function returns a list of all available properties (prop_id[]) on a given sensor node (sn_id).
- getPropertyValue(sn_id, prop_id) ← value: This function returns the value of a given property (prop_id) on a given sensor node (sn_id).
- insertPropertyValue(sn_id, prop_id): This function inserts (or updates) a value of a given property (prop_id) on a given sensor node (sn_id) in the database.
- insertProperty(sn_id, prop_id, [value]): this function inserts a new property (prop_id), [optional with the value of the property], on a given sensor node (sn_id) into the database.

- insertNewSensorNode(sn_id): this function inserts a new sensor node (sn_id) into the database.

The program version database stores all versions of all programs, which can be installed on the sensor nodes. It is SQL-based and contains the following values: ID of the program, version, target, platform, timestamp of the upload, and link to the image. To access the database the following functions are available:

- getAllProgVersions()← [(pn id, name, version)]: This function returns a list of all available programs (pn id, name) including their version information (version).
- insertNewProgVersion(name, version, image): This function inserts a new program into the database, including all related information. It copies the image into a special directory.
- getImage(pn id) ←: This function returns the image of a given program (pn id). The CFEngine is responsible for distributing data and information within the mesh network. For this, the following functions are available (Figure 3):
- propagateData(data): This function copies the given data to the CFEngine.
- newDataAvailable(): This Function notifies, if new data is available.

The WSN monitor module has several tasks. It connects to the WSN information database and to the sensor data database in order to answer to the requests from the management station. Furthermore it writes data coming from the sensor nodes into both databases. The module writes data to the CFEngine, to propagate it within the mesh network. The WSN configurator module is responsible for all configuration tasks. It connects to the WSN information database to read and write data. Furthermore, it creates TCP/IP packets for the sensor nodes, to query sensor node properties and send commands. The module writes data to the CFEngine, to propagate it within the mesh network. The sensor data manager module is responsible for aggregating and storing the sensor data. It writes data into the sensor data database and the WSN information database and copies this data to the CFEngine to propagate them in the mesh network. The code update manager module is responsible for storing newly uploaded images (and related information) in the program version database. Furthermore, it answers to requests from the management station about available programs by sending a complete list of all program versions stored in the program version database. It also executes the command for updating the sensor nodes. Therefore, it loads the new and the old program version of the selected sensor node from the according database. Furthermore, it calculates the differential patch between both versions to reduce the amount of transmitted data. Alternatively, it can just compress the new image. Finally it sends the packet with the differential patch (or the compressed or uncompressed image) to the sensor node gateway. To send queries to the sensor nodes via the sensor node gateway the following functions are available:

- getAllSensors(sn_id[]): This function generates a packet, which includes the request for all available sensors (prop_id) and sends it to the given sensor nodes (sn_id[]).
- getAllProps(sn_id[]): This function generates a packet, which includes the request for all available properties (prop_id) and sends it to the given sensor nodes (sn_id[]).
- getSensorValue(sn_id[], prop_id[]): This function generates a packet, which includes a request for a sensor value of a given sensor (prop_id) and sends it to the given sensor nodes (sn_id[]).
- getPropValue(sn_id[], prop_id[]): This function generates a packet, which includes a request for a property value of a given sensor (prop_id) and sends it to the given sensor nodes (sn_id[]).
- getNeighbors(sn_id[]): This function generates a packet, which includes a request for the neighbors of the sensor node and sends it to the given sensor nodes (sn_id[]).
- getHopDistance(sn_id[]): This function generates a packet, which includes a request for the hop distance from the sensor node to the gateway and sends it to the given sensor nodes (sn_id[]).
- confSN(sn_id[], prop_id, conf cmd): This function generates a packet, which includes the configuration command (conf cmd) to configure a given property (prop_id) and sends it to the given sensor nodes (sn_id[]).

- uploadImg(sn_id[], img, upd cmd): This function generates a packet, which includes the updated image (img) and the update command (upd cmd) and sends it to the given sensor nodes (sn_id[]).

These requests are sent over TCP/IP to the sensor nodes, using unicast, multicast or broadcast protocols.

4.2 Sensor Node Manager

The sensor node manager handles the management tasks and runs as an application of Contiki on the sensor nodes. The sensor node manager consists of a sensor node monitor, a sensor node configurator, and a code updater. This is shown in Figure 4.

The **sensor node monitor** is responsible for handling the monitor requests. It receives monitor requests, generates packets containing the requested values and sends them back to the mesh node. When a sensor node switched-on, the values of all available properties are collected and stored in an appropriate manner (e.g. in a global variable in the memory). Possible properties are: node.type, node.role, chip.name, transceiver.name, battery.name, battery.capacity, battery.curr value, memory.name, memory.size, memory.free space, os.name, os.version, os.last update, routing table, neighbors.ID[], ip address, hosting mesh node.ID, and many more. The following functions implement the functionality:

- InitSensorNodeMonitor(): When a sensor node starts, this function determines all available properties and their values. These values are stored in an appropriate manner, e.g., in an array. The array consists of the property id, the value and a time stamp.
- ReceiveMonRequest(pid[], sender_ip): When a request reaches the monitor, this function is called with an array of the requested properties (pid[]) and the IP address of the sender. This function determines the requested values (by calling GetPropertyValue()) and send it back to the sender (sender_ip), by calling SendSensorValue()).
- value[] GetPropertyValue(pid[]): This function reads the values of all requested properties (pid[]) from the table.
- SendSensorValue(pid[], sid, value, receiver_ip): This function creates one or more packets with the values and sends this to the receiver.
- value UpdatePropertyList(pid): This function determines the value of a selected property (pid), stores this in the table and gives the value back

The **sensor node configurator** executes the configuration commands. It receives a packet containing the attribute, which has to be configured and the configuring command. It performs the configuration, e.g. by writing a configuration file or by setting a flag. Then it creates a notification packet and sends it to the mesh node. The following functions implement the functionality:

- ReceiveConfRequest(conf_cmd, sender_ip): This function is called when a configuration request has been received. It starts the configuration process by calling Configure().
- Configure(conf_cmd): This function performs the configuration by executing the configuration command (conf_cmd).
- RestartSensorNode(): If a configuration demands a restart, this function restarts the sensor node.
- SendNotification(message, receiver_ip): This function sends the notification of the success of the configuration.

The **code updater** is responsible for the code updating process. The program code can be updated in different ways: either the complete code of the system, or parts of it (e.g. a selected application, parts of the operating system, or several bytes in the memory). Furthermore, the code can be delivered uncompressed, compressed, or as a differential patch. The code updater receives the packets and decides how to handle the code: decompress the image, recalculate the differential patch, copy the code into the memory, shift addresses, and so on. If

necessary, the updated application or the whole sensor node has to be restarted. Finally it sends a packet to notify the mesh node of the success of the update. The following functions implement this functionality:

- ReceiveUpdRequest(img, upd_type, sender_ip): After receiving a packet with code, this function adds the incoming packets and stores it in the memory or in an extern flash memory. Then it starts the image building process, by calling the function BuildImage().
- BuildImage(img*, upd_type): This function creates the image according to the type of the update (upd_type).
- RestartSensorNode(): This function restarts the sensor node.
- RestartApplication(app_id): This function starts (or restarts) the selected application (app_id).
- SendNotification(message, receiver_ip): This function sends the notification of the success of the update. If it succeeds, the new version number and a timestamp of the update is sent to the mesh node. If not, the message contains a request for repeating the update.

5. Sensor Node Platforms and Operating System

As described in [1] we selected four heterogeneous sensor node platforms: tmote sky, ESB, BTnode, and MicaZ nodes. All these sensor nodes can be connected over serial interfaces to a mesh node. These connected nodes can be used as sensor gateway nodes to the according sensor networks. Contiki has been selected as the operating system.

5.1 Sensor Gateway Connection

To connect a sensor network with the wireless mesh network, we use a sensor node, which is connected over a serial interface to a mesh node [15]. Contiki is running on the sensor node as the operating system and provides a minimal TCP/IP stack (uIP) for communication. The Linux based Secure Remote Management and Software Distribution for Wireless Mesh Networks [2] is running on the mesh node and provides a standard TCP/IP Stack. To interconnect these two TCP/IP stacks we use the Serial Line Internet Protocol (SLIP) [16], see in Figure 5.

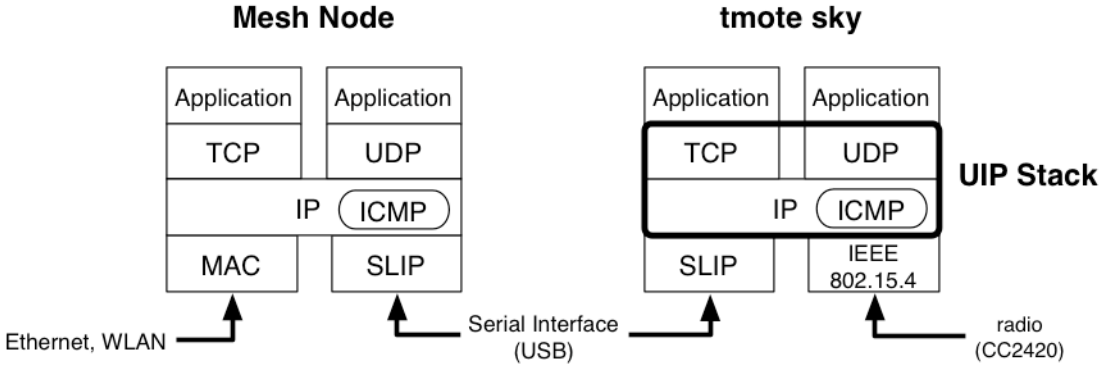

Figure 5. TCP/IP stacks of a tmote sky that is connected to a mesh node

SLIP connects the host IP layer with the client IP layer. It simply sends byte per byte over a serial interface. There is no addressing or error detection implemented in the SLIP protocol. The SLIP protocol defines two special characters: ESC and END. ESC is 333_8. END is 300_8. After the last byte of a packet the END character is sent. Therefore the data byte 3008 is substituted with a two bytes sequence of ESC and 334_8 $(300 + 334)_8$). The data byte 333_8 is substituted with a two bytes sequence of ESC and 335_8 $(300 + 335)_8$. It is shown in Figure 6.

```
#define SLIP_END      0300
#define SLIP_ESC      0333
#define SLIP_ESC_END  0334
#define SLIP_ESC_ESC  0335
```

Figure 6. ESC and END in the SLIP protocol

A SLIP device has been implemented in the Contiki core. The function u8_t slip_send(void) sends the 40 bytes of the IP and TCP headers out of the uip_buffer and the appending application bytes out of the uip_appdata buffer.

The function int slip_input_byte(unsigned char c) is called by the RS232/SIO device driver to pass incoming bytes to the SLIP driver. The function can be called from an interrupt context. SLIP on the mesh node side has been implemented in the file tuneslip.c. It uses the TUN/TAP driver, which is a part of the Linux kernel. To enable the TUN/TAP driver, the kernel must be compiled with the configuration option CONFIG_TUN=m. TUN/TAP provides packet reception and transmission for user space programs.

Figure 7 shows a SLIP connection between a mesh node and a tmote sky node. The mesh node has the IP 192.168.123.80. The tmote sky node is connected over USB on /dev/ttyUSB0.

```
mesh_node0343:~ root# ./tunslip 172.16.0.1 255.255.0.0
—>> ttyUSB0
slip started on ``/dev/ttyUSB0''
opened device ``/dev/tun0''
ifconfig tun0 inet `hostname` 172.16.0.1 up
route add -net 172.16.0.0 -netmask 255.255.0.0 -interface tun0
add net 172.16.0.0: gateway tun0
sysctl net.inet.ip.forwarding=1
sysctl: must specify -w to set variables
ifconfig tun0

tun0: flags=8851<UP,POINTOPOINT,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet 192.168.123.80 —> 172.16.0.1 netmask 0xffffff00
        open (pid 3046)
route add -net 192.168.1.2 -netmask 255.255.255.255 -interface tun0
add net 192.168.1.2: gateway tun0
```

Figure 7. SLIP connection between mesh node and sensor node

5.2 Sensor Node Hardware

The following sections give a short description of our four different sensor node platforms and there serial interface. All sensor node platforms can be connected over their serial interface to a mesh node.

5.2.1 Tmote Sky

The tmote sky nodes [17] provide an IEEE 802.15.4 compliant CC2420 radio module, which is able to communicate with the MicaZ nodes. The tmote sky nodes can also be used as sensor gateway nodes for the MicaZ subnetworks. All tmote sky nodes provide a USB interface. A FT232BL chip from FTDI [18] connects the Universal Asynchronous Receiver Transmitter (UART) bus of the MSP430F1611 microcontroller [19] with the USB interface. A tmote sky node is shown in Figure 8.

Figure 8. tmote sky node

5.2.2 ESB and eGate

The ESB node is like the tmote sky node well supported by the Contiki operation system. An ESB node can be connected to a mesh node over the serial RS232 interface. To connect the ESB sensor node network with a mesh node is also possible to use an eGate stick (shown in Figure 9) from Scatterweb [20]. The eGate stick is identical to the ESB node, except for two issues. First, an eGate does not provide any sensors. Second the eGate stick provides (like the tmote sky) a FT232BL chip from FTDI to connect the USB interface with the MSP430F149 [19] microcontroller.

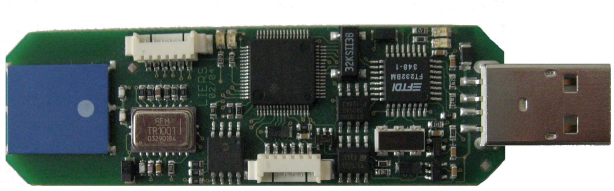

Figure 9. eGate

5.2.3 MicaZ

MicaZ [21] nodes (MRP2600, MP2600) do not provide any sensors or a serial interface. Sensors must be attached separately. For example a MTS400CA sensor board can be attached a MicaZ. It provides the following sensors:

- Temperature & Humidity
- Humidity Accuracy $< 3.5\%$
- Temperature Accuracy < 0.5 Deg. C
- Barometric Pressure 300mbar to 1100mbar, 3% Accuracy
- Ambient Light Sensor (400-1000nm) response
- 2-Axis Accelerometer (ADXL202)

The MicaZ nodes and the MTS400CA sensor board are shown in Figure 10.

To have a serial interface, an USB board can be attached can to a MicaZ node. The USB board uses a FT2232C chip from FTDI to connect the USB interface with the ATMega128L microcontroller. Figure 11 shows a MRP2600 MicaZ node connected to the USB board.

Like tmote sky nodes all MicaZ nodes use an IEEE 802.15.4 compliant CC2420 radio module for wireless communication. These radio modules use the same frequency range as the IEEE 802.11 radio implemented

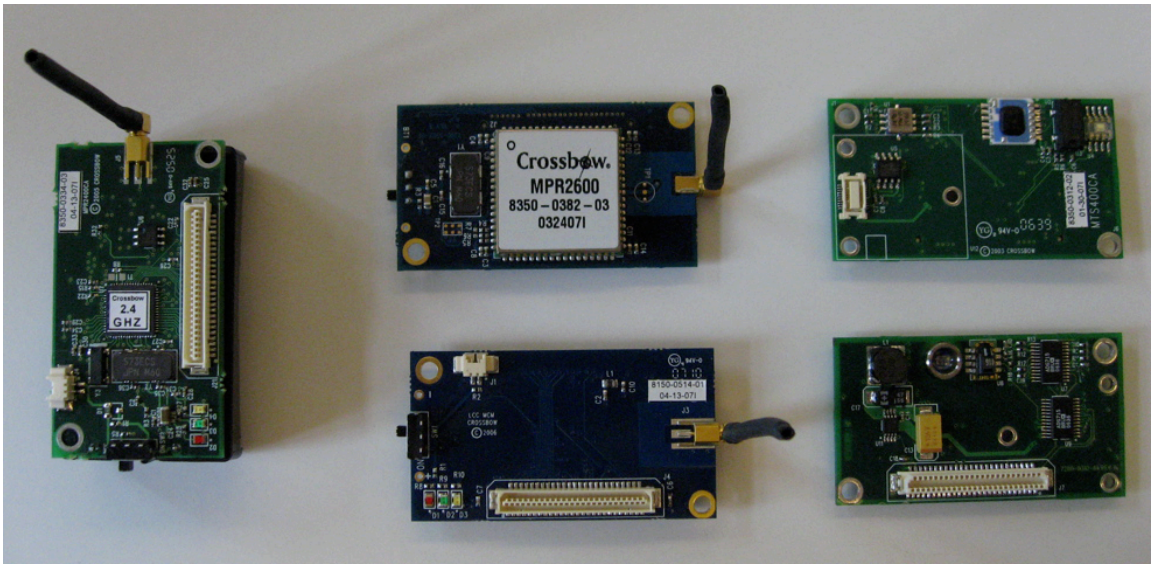

Figure 10. Left: MRP2600. Center: MP2600. Right: MTS400CA

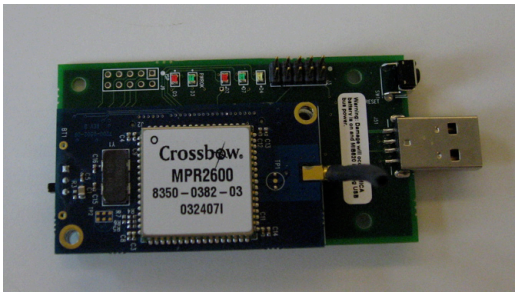

Figure 11. MRP2600 MicaZ node connected to the USB board

on the mesh nodes. As shown in Figure 12 IEEE 802.15.4 has 17 channels and IEEE 802.11 three channels. Channel 25 and 26 of IEEE 802.15.4 are outside the frequency range of IEEE 802.11.

Possible interferences between the IEEE 802.15.4 radio modules and the IEEE 802.11 radio modules have to be taken care of. Another source of interference are the USB powered sensor nodes. The USB voltage is not constant. Power supply units are powered by alternating current. This affects the frequency of the radio modules.

5.2.4 BTnode

BTnodes [22] have two wireless interfaces: a CC1000 radio module and a Zeevo ZV4002 Bluetooth system. Using a USB adapter a sensor node gateway can be connected to a mesh node. Figure 13 shows two BTnodes, to the left one an USB is attached

5.3 Operation System Contiki

To realize the reliable communication mechanisms for wireless sensor networks (described in work package 2) the Contiki operating system has to be modified, especially the integrated communication stacks. In this subsection we describe the general development process, parts of the Contiki core and the communication stacks.

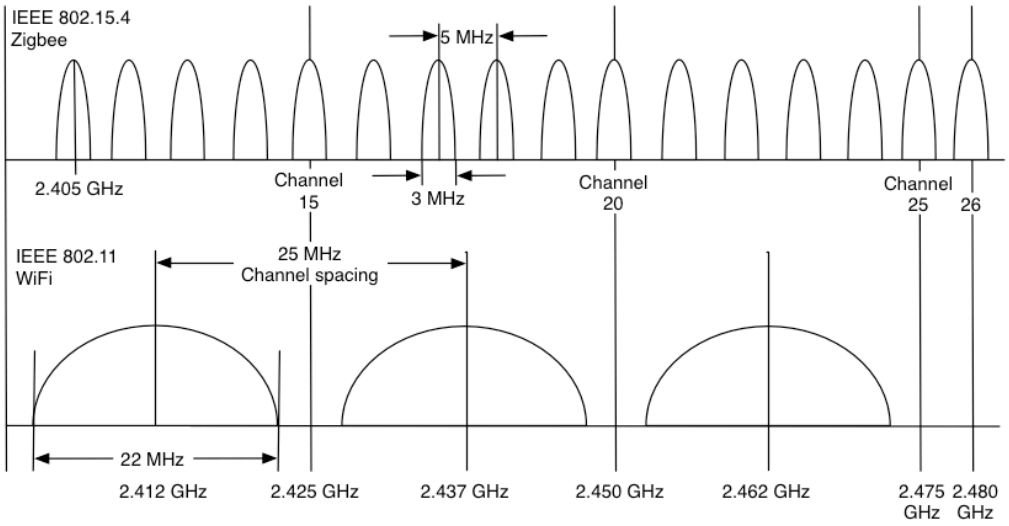

Figure 12. Frequency range of IEEE 802.11 and 802.15.4

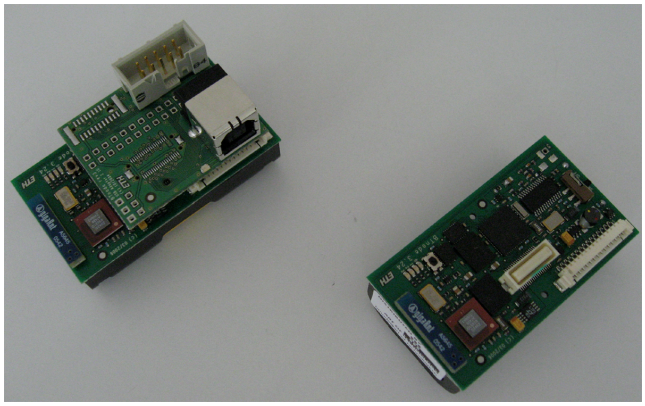

Figure 13. BTnode with connected USB adapter

5.3.1 Contiki Core and Application Development

Code for applications and the Contiki core are strictly separated. To recompile applications for different platforms different make commands have been defined. Ideally, no changes are needed for the application to port it to another platform. To develop new applications also two virtual platforms (native and netsim) have been defined. The native platform simulates a single node. It is much easier and less tedious to debug code first for the native platform, than directly on the concrete hardware platform. The second virtual platform, netsim, simulates 25 nodes. In contrast to the native platform with netsim communication between different sensor nodes can be debugged. Figure 14 shows the graphical interface of the netsim platform.

On the hardware side the tmote sky and ESB platforms are well supported by Contiki. Contiki supports the AVR microcontroller of both MicaZ and BTNodes, but the MicaZ and BTNodes platforms are yet not defined in Contiki. For example to build, execute and upload the hello-word example for the different platforms, the following commands can be used (see Figure 15):

5.3.2 Protothreads

The kernel of Contiki is event-based. A kernel based on multithreading requires more memory than an event based kernel. Normally one thread needs one stack. A problem with events is an unstructured code flow, caused by a huge state machine. Therefore, Contiki provides a programming abstraction called Protothreads

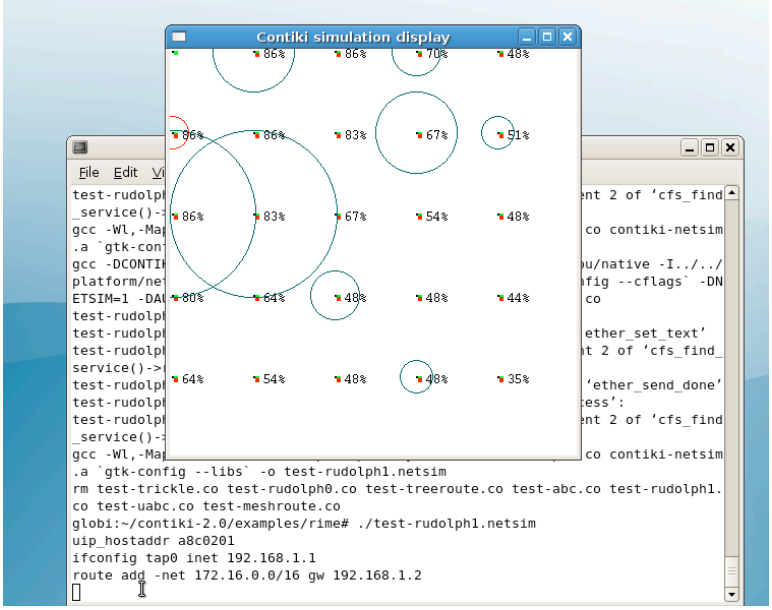

Figure 14. Netsim debug plattform

```
make TARGET=native                    Build monolithic system for native
./hello-world.native                  Run entire Contiki system + app

make TARGET=netsim                    Build netsim simulation
./hello-world.netsim                  Run netsim simulation

make TARGET=sky                       Build monolithic system image
make TARGET=sky hello-world.u         Build & upload system image
make TARGET=sky hello-world.ce        Build loadable module

make TARGET=esb                       Monolithic system image for ESB
make TARGET=esb hello-world.u         Build & upload image
make TARGET=esb hello-world.ce        Build loadable module
```

Figure 15. Different make targets

[23]. It is a combination of events and threads. It has one stack with low memory usage (like events). It provides a sequential flow control and no state machine is needed (like threads). PT_WAIT_UNTIL(condition) is a programming primitive for a conditional blocking wait. In addition to the unconditional blocking wait PT_YIELD() and PT_SPWAN() to initial a protothread child process it is possible to implement memory conserving algorithms for communication and code updates. Figure 16 shows an example skeleton of a protothread. The implementation of Protothreads uses only the C preprocessor. Therefore, it is fully portable to all C compilers.

5.3.3 Network

Contiki supports two different communication stacks: Rime [23] and uIP [4]. uIP has been developed for the communication with TCP/IP-based networks. Rime has been designed for wireless communication. Contiki

```
int protothread(struct pt *pt) {
        PT_BEGIN(pt);
        PT_WAIT_UNTIL(pt, condition1);
        if(something) {
                PT_WAIT_UNTIL(pt, condition2);
        }
        PT_END(pt);
}
```

Figure 16. Protothread skeleton

applications can use uIP or Rime or both stacks simultaneously for communication. Rime can also run over uIP and vice versa as depicted in Figure 17.

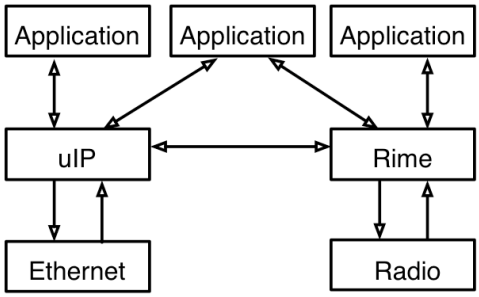

Figure 17. Rime and uIP communication stacks

In the **uIP** stack, the TCP connection can be opened by using tcp_connect() and tcp_listen(). A UDP flow is established by udp_new(). When a new connection has been established, data arrives, or connections are closed, a tcpip_event is thrown. TCP connections are periodical polled for data. UDP packets are sent with uip_udp_pecket_send(). There are two uIP APIs. The event-driven raw uIP API works well for small applications, because of using an explicit state machine. Protosockets work better for larger applications. Using protothreads, protosockets are using low memory and provide a sequential flow control. Protosockets work only for TCP connections.

Rime, a lightweight, layered communication stack is used for communication over low-power radio modules. Rime has a lot of very thin communication layers to reduce the complexity.

- Anonymous best-effort single-hop broadcast (abc)
- Identified best-effort single-hop broadcast (ibc)
- Stubborn identified best-effort single-hop broadcast (sibc)
- Best-effort single-hop unicast (uc)
- Stubborn best-effort single-hop unicast (suc)
- Reliable single-hop unicast (ruc)
- Unique anonymous best-effort single-hop broadcast (uabc)
- Unique identified best-effort single-hop broadcast (uibc)
- Best-effort multi-hop unicast (mh)
- Best-effort multi-hop flooding (nf)
- Reliable multi-hop flooding (trickle)

- Hop-by-hop reliable data collection tree routing (tree)
- Hop-by-hop reliable mesh routing (mesh)
- Best-effort route discovery (route-disovery)
- Single-hop reliable bulk transfer (rudolph0)
- Multi-hop reliable bulk transfer (rudolph1)

Figure 18 shows the dependency between the different rime communication layers. These layers are currently under development by SICS [24].

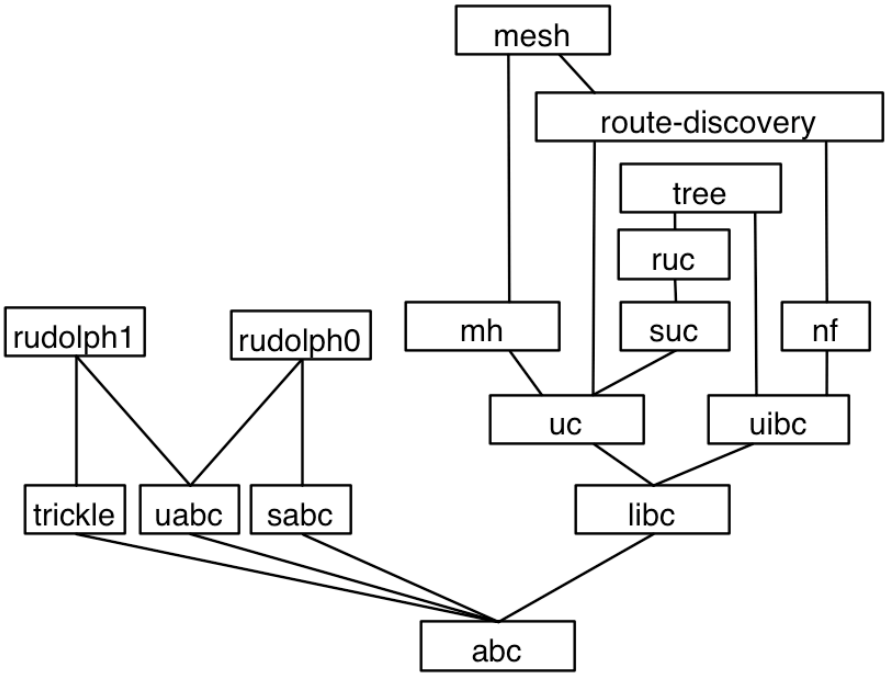

Figure 18. Dependency between the different rime communication layers

5.4 Future Work

Contiki has to be ported to the MicaZ and BTnode platform. The implementation of the management architecture on the mesh nodes has to be finished and tested in an adequate manner. For all sensor node platforms, the efficiency of the different code update mechanisms has to be analyzed.

References

[1] M. Anwander, G. Wagenknecht and T. Braun: Energy-efficient Management of Heterogeneous Wireless Sensor Networks *Technical Report IAM-07-002, University of Bern*, April 30, 2007

[2] T. Staub, D. Balsiger, M. Lustenberger, T. Braun. Secure Remote Management and Software Distribution for Wireless Mesh Networks *in 7th International Workshop on Applications and Services in Wireless Networks (ASWN 2007)*, Santander, Spain, May 24 - 26, 2007.

[3] A. Dunkels, B. Grnvall, Th. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. *In Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004*, Tampa, Florida, USA, November 2004.

[4] A. Dunkels. Full TCP/IP for 8-bit Architectures. *ACM MobiSys*, pp. 85-98, San Francisco, May 2003.

[5] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. *Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, oulder, Colorado, USA, November 2006.

[6] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-Level Sensor Network Simulation with COOJA. *First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Tampa, Florida, USA, November 2006.

[7] P. Levis, N. Patel, D. Culler, and S. Shenker Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks *First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)* , March 2004.

[8] T. Stathopoulos, J. Heidemann, D. Estrin. A remote code update mechanism for wireless sensor networks, Tech. Rep. CENS-TR-30. *University of California, Los Angeles, Center for Embedded Networked Computing*, November 2003.

[9] P. J. Marron, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. *Third European Workshop on Wireless Sensor Networks (EWSN 2006)*, Zuerich, Switzerland, February 2006.

[10] N. Reijers, and K. Langendoen. Efficient code distribution in wireless sensor networks. *2nd ACM Intl. Conf. onWireless Sensor Networks and Appl*, (2003) 6067.

[11] J. Jeong, and D. Culler. Incremental network programming for wireless sensors. *First IEEE Comm. Soc. Conf. on Sensor and Ad Hoc Communications and Networks*, 2004.

[12] A. Tridgell Efficient Algorithms for Sorting and Synchronization. *PhD thesis, The Australian National University*, 1999

[13] M. Burgess. A tiny overview of cfengine: Convergent maintenance agent. *1st International Workshop on Multi-Agent and Robotic Systems MARS/ICINCO*, (Barcelona, Spain), September 2005.

[14] AAI. Authentication and Authorization Infrastructure. http://www.switch.ch/aai/. Last visit June 2007.

[15] WRAP. Wireless router application platform board. http://www.pcengines.ch. Last visit June 2007.

[16] RFC 1055. Romkey J. L. Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP.

[17] Tmote SKY. Reliable low-power wireless sensor networking platform for development. Platform for self-configuring wireless sensor networks. http://www.moteiv.com. Last visit June 2007.

[18] FTDI. FTDI offer a range of products to allow interfacing to devices over USB. http://www.ftdichip.com. Last visit June 2007.

[19] Texas Instruments Produces the MSP430, a low power 16-bit Flash microcontroller. http://focus.ti.com. Last visit June 2007.

[20] Scatterweb. Platform for self-configuring wireless sensor networks. http://www.scatterweb.net. Last visit June 2007.

[21] MicaZ. The MICAz is a 2.4 GHz, IEEE/ZigBee 802.15.4, board used for low-power, wireless, sensor networks. http://www.xbow.com. Last visit June 2007.

[22] BTnode. Versatile and flexible platform for fast-prototyping of sensor and ad-hoc networks. http://www.btnode.ethz.ch. Last visit June 2007.

[23] A. Dunkels. Rime - a lightweight layered communication stack for sensor networks. *European Conference on Wireless Sensor Networks (EWSN)*, Poster/Demo session, Delft, The Netherlands, January 2007.

[24] SICS. Swedish Institute of Computer Scince. http://www.sics.com. Last visit Juni 2007.