Energy-efficient Management of Heterogeneous Wireless Sensor Networks

Markus Anwander, Gerald Wagenknecht, Torsten Braun

Abstract

The first work package defines the overall wireless sensor network (WSN) management and code distribution framework. This includes the determination of appropriate sensor node platforms and an appropriate sensor operating system. Furthermore, it includes the definition and implementation of a management architecture for WSNs. In a first step several types of sensor nodes and sensor operating systems have been evaluated. Based on the identified management tasks required in a WSN a management architecture has been defined. Management tasks to be supported are: monitoring the WSN, configuration of the WSN, code updates and managing sensor data. 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. All management tasks are controlled by the management station. Management nodes are implemented as wireless mesh nodes. The next steps include the adaptation of Contiki, the chosen operating system, to have it running on all used sensor platforms. Further, to access and interconnect the different sensor sub-networks, gateways running on the mesh nodes have to be developed. In parallel, the implementation of the management architecture will start. IP support is planned for all communication to and within the heterogeneous WSN. Reliable transport protocols will be developed for unicast, multicast and broadcast communication. Therefore, TCP and existing multicast and broadcast protocols have to be adapted as described in WP2.

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

This project focuses on the investigation of efficient and reliable communication mechanisms that are required for the efficient operation of a wireless sensor network (WSN) management framework. An appropriate WSN management architecture needs to be identified and implemented. The tasks of the first work package include among others to define an appropriate management framework for WSNs as well as to identify appropriate sensor node platforms and operating systems. In this technical report we present the selected sensor node platforms and an adequate operating system. Furthermore, we define the management tasks and the management architecture to perform these tasks. This report is structured as follows: the second Section introduces related work in the topics of code distribution, middleware, and management of wireless sensor networks. The third Section presents the hardware that will be used, i.e. the different types of sensor nodes and the management node (wrap board). The fourth Section covers the evaluation of several operating systems. In Section 5 the definition of the management framework is described. Section 6 gives an outlook and points out the next tasks of the project.

2. Related Work

2.1 Code Distribution

Software and code updates are critical tasks, in particular in enterprise networks [1]. Software distribution protocols can be divided into three steps: selection of target nodes, delivery of software updates, and verification of the update completion.

The authors of [2] propose efficient code distribution in wireless sensor networks. The focus is on reducing the total amount of data for code update by only transmitting the differences between the old and new code. Several optimizations like address shifts, padding and address patching are made. This makes the approach very suitable for incremental updates. A simple point-to-point communication is used for code distribution and an efficient distribution algorithm is an open task.

Other code distribution mechanisms rather apply flooding or broadcast mechanisms, which do not take advantage of more energy-efficient multicast communication concepts. Multi-hop Over-the-Air Programming (MOAP) [3] 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. When the whole image has been 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. SPIN (Sensor Protocols for Information via Negotiation) [4] disseminates information among sensors in an energy-constrained wireless sensor network efficiently. Nodes running a SPIN communication protocol name their data using meta-data and use meta-data negotiations to eliminate the transmission of redundant data throughout the network. SPIN is a 3-stage-handshake protocol using advertisements, requests and final data transmissions. Special mechanisms for achieving reliability are not considered in more detail.

Trickle [5] 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.

While Trickle addresses single packet dissemination, Deluge [6] extends it to support large data objects. Deluge shares many ideas with MOAP, including the use of negative acknowledgements, unicast requests, broadcast data transmission, and windowing to efficiently manage which segments are required. However, Deluge considers additional key design options such as fragmentation, spatial multiplexing, and asymmetric link support.

2.2 Middleware

Most code distribution approaches support homogenous sensor node environments and distribute specific code for such sensor node platforms. However, advanced WSNs are typically composed of rather heterogeneous sensor nodes, since the functionality required is highly versatile. For example, in hierarchical WSN architectures, cluster leaders might be more powerful than specialized sensing units. The sensor nodes may have several sensing units and the hardware platform may depend on the sensing task.

Promising concepts to hide hardware heterogeneity in WSNs are middleware approaches [7] based on either scripting language interpreters or virtual machines. Virtual machines offer an abstract interface to applications not only hiding hardware heterogeneity, but also providing more powerful operations on a higher abstraction level. This allows applications to be written in a more compact way and more importantly to deliver programs using a lower number of packet transmissions resulting in energy-efficient code transport. In addition, virtual machines typically allow dynamic reprogramming and self-configuration of the sensor nodes. Safety is supported when programs are interpreted in a sand-box limiting access to certain node resources. The advantages come with higher costs regarding code execution efficiency, but since transmitting a single byte costs approximately processing 1 Kbytes, this drawback seems to be tolerable. Several middleware approaches have been recently proposed: Sensorware [8] is a high-level scripting abstraction based on Tcl with special extensions for sensor applications such as communication and sensing. It offers an expandable runtime environment providing multiple services and uses multi-threading services from the underlying operating system, but has rather high memory requirements.

Impala [9] is a middleware architecture that enables modular application updates and repair ability in WSNs. Switching between multiple applications is supported in order to choose the most appropriate application depending on the nodes current state, or information it has gathered from its vicinity. Each module and each application has a version number. The module-based version system allows selective software transmissions. Impala uses reliable local multicast to propagate the code to the appropriate neighbors. Drawbacks of Impala are its design to run on rather powerful personal digital assistants and its code distribution is restricted to vicinity updates. Furthermore, Impala transmits native code and is therefore not suitable for heterogeneous WSNs.

Maté [10] is a byte-code interpreter (virtual machine) running on TinyOS and allowing run-time reprogramming. Code is broken into capsules of 24 instructions (bytes). Larger programs are composed of several capsules. A drawback of this approach is that it provides a static set of execution events and only limited assembly-level programming. Furthermore, logical partitioning of the network is supported by means of group IDs, but has the drawback that it is not possible to share code modules between several groups. Code is diffused into the network in a virus-like approach, i.e. the code is injected in the network and then propagates automatically to all others.

Tofu [11] is based on TinyOS and provides an architecture to build application specific virtual machines for active sensor networks. Its architecture consists of three basic elements: Threads are the units of concurrency, operations are the units of execution functionality, and capsules are the units of code propagation. Code is propagated in an epidemic-like approach where only the needed data is transmitted. This approach uses a flooding-like code distribution mechanism, making it less usable for our approach.

The Sun SPOT project [12] introduced the Squawk virtual machine, which is a fully capable J2ME Java VM with operating system functionality. As any Java VM Squawk has significant memory and processing requirements. Squawk is not a option for heterogeneous sensor networks because it is tailored to and only running on the specific hardware it is shipped with.

2.3 Management of Wireless Sensor Networks

MANNA [13] presents an information architecture and a functional management architecture for WSNs. The management architecture provides functions to establish configurations for the sensor network entities. In [14] it is proposed to deploy several manager nodes in a hierarchical way based on clustering. Each manager node is responsible for a cluster of sensor nodes. Manager nodes communicate with access points. The information architecture defines the information units and the information exchange among the entities. Currently an implementation does not exist for MANNA. Guidelines are proposed, but the communication model and other issues are not yet defined.

TinyCubus [15] presents a management and configuration framework for WSNs. It is also based on a clustered architecture and assigns certain roles to the sensor nodes. Another focus of TinyCubus is on code distribution minimizing the code fragments to be distributed in a WSN. Reliability shall be supported by implicit acknowledgments and retransmissions. The code distribution mechanism has been evaluated in rather friendly environments without high error rates that can easily happen in WSNs.

Another paradigm for network management in general and for WSNs in particular are mobile agents. Mobile agents travel through a WSN and may carry mobile code to be processed on wireless sensor nodes. The mobile code can perform sensor data processing but can also perform configuration of sensor nodes. In particular, mobile agents can carry code to be installed on the sensor nodes. Agilla [16] is a mobile agent based middleware platform for moving or cloning code throughout a WSN. Tofu [11] uses so-called capsules as units of code propagation in the application-specific virtual machine concept.

2.4 Reliable Transport in Wireless Sensor Networks

Several research works have already been performed in the context of wireless sensor networks. Many of them describe new sophisticated protocols for WSNs. Some explicitly focus on reliable transport of sensor data from sensor nodes to base stations (upstream), while only a few support data transport of configuration and program data towards sensor nodes (downstream). The reliability requirements differ significantly, since sensor data often have some redundancy and some loss can be tolerated. However, packet loss cannot be tolerated in the case of distributing program code to the sensor nodes.

An alternative to introducing new protocols for reliable data transfer in WSNs is to use, adapt, and enhance standard protocols such as TCP [17]. Running TCP/IP on each device in the sensor network allows using a single standard protocol suite on all nodes and connecting the sensor network directly to IP based network infrastructures without proxies or middle-boxes. Data to and from the sensor network can be routed via any device with Internet connectivity rather than via special protocol proxy nodes only typically implemented in base stations. The sink (base station) can then even be located inside the fixed network, not necessarily at the edge between fixed network and wireless sensor network. Further, experiences from industrial researchers point out that using a standard such as TCP/IP for (on-body) WSNs has facilitated application development and system integration in terms of data collection and configuration [18]. While UDP may be used for transferring sensor data and other information that do not need reliable stream transport, TCP should be used for administrative tasks that require reliability and compatibility with existing application protocols. Examples are configuration and monitoring of individual sensor nodes as well as download of binary code and data aggregation descriptions to sensor nodes. In particular, downloading code to designated nodes such as cluster heads in a certain geographical region requires a reliable unicast protocol. One might argue that TCP/IP implementations consume too many sensor node resources. However, it has been shown that TCP/IP can be implemented on sensor nodes with limited processing power and memory [19]. TCP/IP may result in relatively large headers that may add significant overhead in case of short packets. However, if TCP is mainly used for configuration and programming tasks, a rather high amount of data is transferred and packets become rather large. To deal with the problems of TCP in wireless multi-hop networks such as end-to-end retransmissions, new approaches such as Distributed TCP Caching (DTC) [20] have been developed. DTC uses local caching of TCP segments and local retransmission at intermediate nodes reducing drastically the transmissions of TCP segments and acknowledgements.

3. Hardware Platforms

A WSN consists of a huge number of nodes, often randomly distributed in a large area. Generally, a sensor node consists of a micro-controller, some sensors, and a low-power radio for communication. Currently available sensor nodes are mainly prototypes for research purposes. The RVS group owns about 30 Embedded Sensor Board (ESB) [21] nodes. An additional number of sensor nodes have been evaluated, from which three more sensor nodes have been chosen to build a heterogeneous sensor network. Beside the available ESB nodes, tmote SKY, BTnodes and micaZ have been chosen. They are widely used in the research community, are well documented and have the adequate properties (memory, energy-efficiency, etc.). For the management backbone a Wireless Router Application Platform Board (WRAP) [22] has been selected. The individual hardware is described in more detail in the subsequent Sections.

3.1 Embedded Sensor Boards

Figure 1 shows an ESB [21] sensor node. It provides several communication interfaces and some sensors for monitoring the environment. The ESB platform has been developed at ScatterWeb GmbH, a spin-off company of the Freie Universität Berlin. The platform has already been used in several research projects at University of Bern.

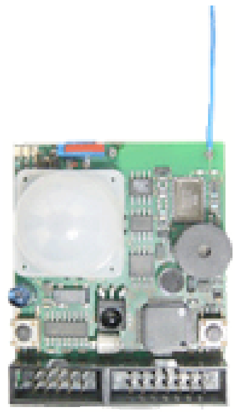

Figure 1. ESB node

Microcontroller	MSP430F149 from Texas Instruments
Radio	TR1001, 868.35 MHz hybrid transceiver
Memory	60 KB flash memory, 2 KB RAM, 64 KB EEPROM
Interfaces	A serial and a JTAG Interface

3.2 Tmote SKY

Figure 2 shows a tmote SKY [23]. It provides an IEEE 802.15.4 radio interface and a USB Port for development and communication. The Moteiv Corporation in San Francisco develops these sensor boards. An 802.15.4 data frame can either include 90 bytes payload of an UPD/IP packet or 41 bytes payload of a TPC/IP packet.

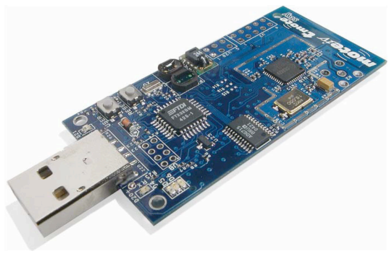

Figure 2. Tmote SKY

Microcontroller	MSP430F1611 from Texas Instruments
Radio	CC2420, 2.4 GHz IEEE 802.15.4 transceiver
Memory	48 KB flash memory, 10 KB RAM, 1024 KB serial storage, 16 KB EEPROM
Interfaces	USB Interface

3.3 BTnode

Figure 3 shows a BTnode [24]. It provides several communication interfaces, including a Bluetooth radio. The sensor boards are developed in the BTnode project at ETH Zurich. They basically come with the same hardware features as the widely used Mica2 Mote from Crossbow. However, they have more SRAM and an additional Bluetooth radio interface.

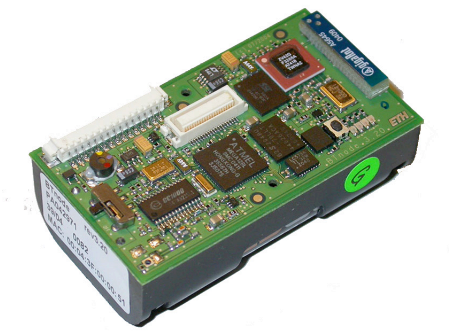

Figure 3. BTnode

Microcontroller	ATMega128L from Atmel
Radio	CC1000, 315, 433, 868, 915 MHz transceiver
Memory	128 KB flash memory, 64 or 4 KB RAM, 4KB EEPROM
Interfaces	Several interfaces available

3.4 MicaZ

A MicaZ [25] node is depicted in Figure 4. It again provides several communication interfaces. Like the tmote sky node it has an integrated IEEE 802.15.4 radio transmitter. These sensors have been developed by the Crossbow Corporation in San Jose.

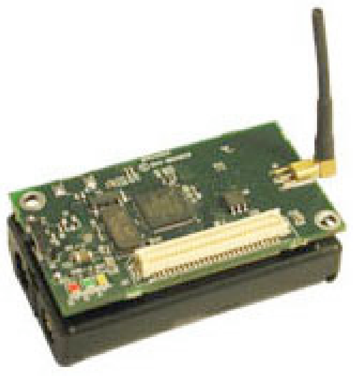

Figure 4. MicaZ

Microcontroller	ATMega128L from Atmel
Radio	CC2420, 2.4 GHz IEEE 802.15.4 transceiver
Memory	128 KB flash memory, 4 KB RAM 512 KB serial storage, 4 KB EEPROM
Interfaces	Several interfaces available

3.5 WRAP Borad

Figure 5 shows a WRAP board [22] covered with an outdoor enclosure. The WRAP board provides several communication interfaces. The WRAP boards can be used to build a fully meshed network using IEEE 802.11 radio transmitters. The boards have been developed by the PC Engines GmbH in Zürich.

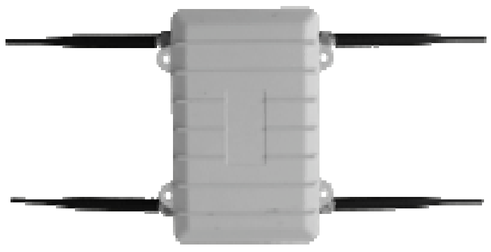

Figure 5. WRAP Borad

Microcontroller	AMD Geode 233
Radio	2 mini PCI for IEEE 802.11
Memory	128 MB RAM, up to 8 GB Flash Memory
Interfaces	USB, serial, LPC, I2C

4. Operating Systems

In this section different operating systems for wireless sensor nodes are presented. Based on the evaluation of the advantages and disadvantages of an operating system the most appropriate solution is chosen.

Hardware limits require the development of simplified versions of traditional operating systems for sensor nodes. In the following, four different operating systems for sensor nodes are evaluated.

4.1 ScatterWeb

The ScatterWeb [21] operating system has been developed for the ESB nodes. It supports a native CDMA protocol for wireless communication. The sensor values (temperature, noise, vibration, etc.) can be sent over the network to a sink.

4.2 MANTIS

MANTIS [26] is an open source operating system written in C for WSN platforms. It is a classical layered operating system supporting multi-threading, pre-emptive scheduling with time slicing, and a network stack. The thread management is done by a static thread table with a fixed size. Each program must have stack space allocated from the system heap. A locking mechanism is required to maintain mutual exclusion of shared variables.

4.3 TinyOS

TinyOS [27] is an open source operation system written in nesC for WSN platforms. Its architecture is component-based and it bases on an event-driven kernel with static memory allocation. In TinyOS the system components are statically linked to the kernel. System modifications after deployment are basically not possible. To support code updating for TinyOS at run-time, the virtual machine Maté [10] has been developed. Additionally, a network simulator for TinyOS called TOSSIM is available as an open source project.

4.4 Contiki

Contiki [28] [29] is a dynamic operating system with special focus on portability. It is written in C and supports over 14 platforms and 5 CPUs. A small TCP/IP stack (μIP) has been implemented [19]. Protothreads, a novel thread-like construct on top of the event-driven kernel, reduces the complexity of event-driven programs by removing state machines. Moreover, Contiki supports preemptive multi-threading, inter-process communication and dynamic run-time linking of standard Executable Linkable Format (ELF) files. ELF is a standard format for relocatable object code and executable files. Program modules can be updated and loaded at run-time. Contiki and the network simulator COOJA for Contiki are open source projects and run under BSD license.

4.5 Evaluation

A detailed evaluation of the project requirements and the available operating systems has shown that the development of an own operating system solution is neither feasible nor needed. It is more efficient to adapt an existing operating system to the used sensor hardware.

From the software architecture and the documentation of MANTIS OS it was obvious that the usage of MANTIS OS in our project would require a lot of adaptations. In comparison to TinyOS, Contiki appears to be more appropriate for our project. The kernel properties and the high portability of Contiki are important factors to support a management infrastructure for heterogeneous WSNs. For example, the implemented TCP/IP stack allows an adapted support of the Scalable Reliable Multicast (SRM) [30] protocol. For reliable point-to-point transport the Distributed TCP Caching (DTC) [20] mechanism and TCP Support for Sensor networks (TSS) [17] can be extended. Both approaches are compared in [31]. Lately, the developers of Contiki arranged the first international Contiki workshop in Stockholm on the 26-27 March this year [28]. Discussions with the developers and other workshop participants confirmed the choice of Contiki as the most appropriate operating system for our purposes. The ESB and tmote SKY nodes already run under Contiki.

5. (Wireless Sensor) Network- and Node Management Framework

In this section, the framework for managing a heterogeneous wireless sensor network is described. First, a typical management scenario is presented. In the second part the typical management tasks which have to be supported are defined. The structure of the management architecture is given in part three. Finally, we describe how the management tasks are handled in the management framework.

5.1 Management Scenario

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 6.

The sensor nodes, depicted in green, turquoise and red circles, might have multiple sensors for monitoring the environment. All sensor nodes of one type are able to communicate with each other using wireless radio modules. Accordingly, they build a sensor subnet. Existing sensor platforms in general have different radio modules and are thus not able to communicate with each other. The tmote SKY and the Mica2 use the same radio module and should be able to communicate with each other.

To interconnect the different subnets, wireless mesh nodes are used (grey circles). A sensor node can be plugged into a mesh node via USB, serial, or another interface. The sensor node plugged into the mesh node operates as gateway to the sensor subnet. Among each other the mesh nodes communicate via IEEE 802.11. Thus, communication between several sensor subnets is guaranteed. Additionally to the inter-subnet communication, the mesh nodes have further tasks such as performing management tasks, as well as aggregating and/or processing data from the sensors, etc. 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.

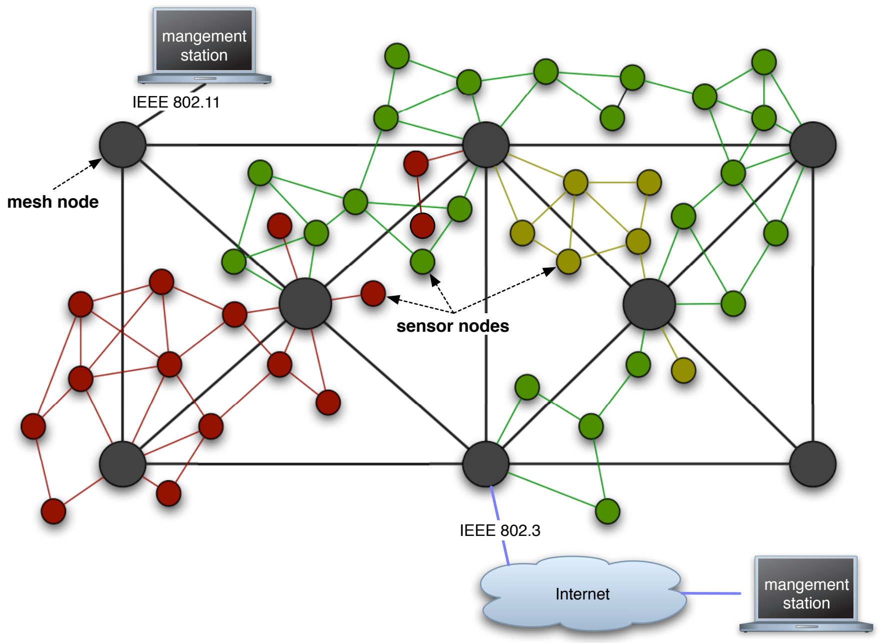

Figure 6. A possible management scenario

5.2 Management Tasks

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 first task includes - according to the scenario (described above) - that a ll sensor nodes in the several subnets are displayed at the management station. Furthermore, all information about the several sensor nodes has to be collected and displayed. This includes sensor node hardware details (chip, memory, transceiver, etc), sensor node software details (operating systems versions, protocols, applications, etc), dynamic properties (battery, free memory, etc) and, if available, localization information.

In normal use a sensor node sends its information if an event is detected or a timer has expired. Furthermore, a sensor node sends its ID when it joins the network. Then it sends the static information, which does not change during lifetime such as chip, transceiver, etc. Additionally the management station of course may query sensor nodes.

The (re)configuration part includes sensor node configuration and network configuration. Sensor node configuration includes configuring the sensors (e.g. sensing intervals, etc), configuring applications, etc. Configuring the network means for example changing routing tables.

Code distribution means to update the operating system, parts of it, or updates of the applications. Mechanisms to handle incomplete, inconsistent and failed updates have to be provided.

The main task is managing the sensor data. This includes mechanisms to store and distribute the collected data from the sensor nodes. The collected data contain measurements from the sensors, the sensor node properties and the WSN infrastructure information.

5.3 Management Architecture

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.

5.3.1 The Management Station

The management station is divided into two parts. It consists of a laptop or remote workstation with a user interface via web browser and a mesh node running a management system for Wireless Mesh Networks (WMNs) developed by our research group [32]. This management system includes a HTTP server with PHP and several other components. See also Figure 7.

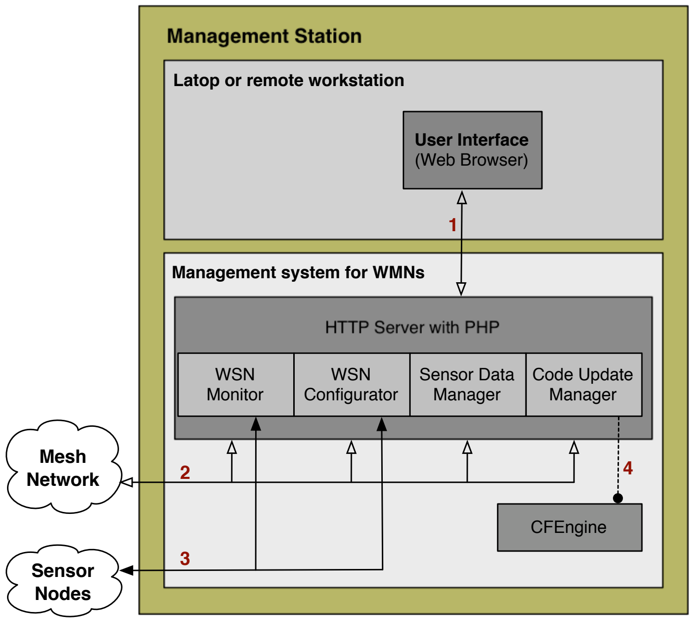

Figure 7. Management station architecture

The **user interface** consists of several windows for displaying the WSN overview with the mesh nodes and the subordinated sensor nodes. Information about the sensor nodes (chip, memory, battery, sensors, etc.)

is displayed. Furthermore, it contains a window for configuring the sensor nodes, a window to upload new program images and a window to handle the updating process. Finally, a window to handle or download the sensor data is provided. The communication between the user interface and the management system for WMNs is done via HTTPS (depicted as **1** in Figure 7).

The **management system for WMNs** contains a small Linux distribution including all required applications, especially a HTTP server supporting PHP. The HTTP server maintains several modules to handle the requests from the user interface and sends them to the mesh nodes, the sensor nodes or the CFEngine [33], [34]. Communication with a mesh node is done via TCP/IP with HTTPS servers running on the mesh nodes (depicted as **2** in Figure 7). The communication between the management station and the sensor nodes again is done via TCP/IP (depicted as **3** in Figure 7). If data has to be distributed within the mesh network, the CFEngine is used and the data is copied into a special directory (depicted as **4** in Figure 7).

The **WSN monitor** is responsible for all requests concerning the monitoring of the WSN and its sensor nodes. It shows which mesh nodes with which subordinate sensor nodes exist. Furthermore, it shows all available information of all available sensor nodes. And, if the user wants, it requests information of a single sensor node. The **WSN configurator** is responsible for all requests concerning the configuration of the sensor nodes. It sends a configuration command to a selected sensor node. The **code update manager** first copies the uploaded image to the CFEngine to propagate this into the mesh network. Further it shows all available program versions and finally starts the updating process. The **sensor data manager** queries sensor data, which are collected and stored in the mesh nodes.

5.3.2 WSN Manager

The WSN manager consists of three databases and four program modules. The databases are: the program version DB, the WSN information DB, and the sensor data DB. The program modules consist 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 8.

The **sensor data DB** 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 (depicted as **4** in Figure 8):

- getSensorData(sn_id, prop_id, time_interval) $\leftarrow$ 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) $\leftarrow$ value: This function gives the last value of a given sensor (prop_id) on a given sensor node.
- getAllSensorData() $\leftarrow$ 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 DB** 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 (depicted as **5** in Figure 8):

- getAllMeshNodes() $\leftarrow$ mn_id[]: This function returns a list of all available mesh nodes (mn_id).
- getAllSensorNodes() $\leftarrow$ (sn_id, mn_id)[]: This function returns a list of all available sensor nodes (sn_id) including the responsible mesh node (mn_id).

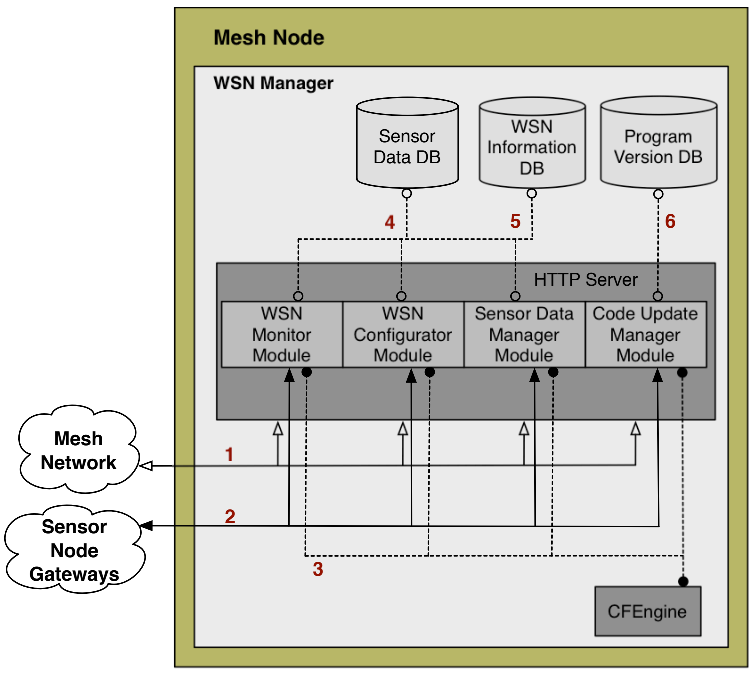

Figure 8. Mesh node architecture

- 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 functions 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 functions 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 DB** 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 (depicted as **6** in Figure 8):

- 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 functions 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 (depicted as **3** in Figure 8):

- 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 DB and to the sensor data DB 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 DB and the WSN information DB 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 DB. 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 DB. 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 DB. 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 (depicted as **2** in Figure 8):

- 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.

5.3.3 Sensor Node Manager

The sensor nodes are - as described above - simple nodes, which can measure sensor data. As it can be seen in Figure 9, the sensor nodes have a **sensor node manager** to handle the management tasks. The sensor node manager consists of a sensor node monitor, a sensor node configurator, a sensor data sender, and a code updater.

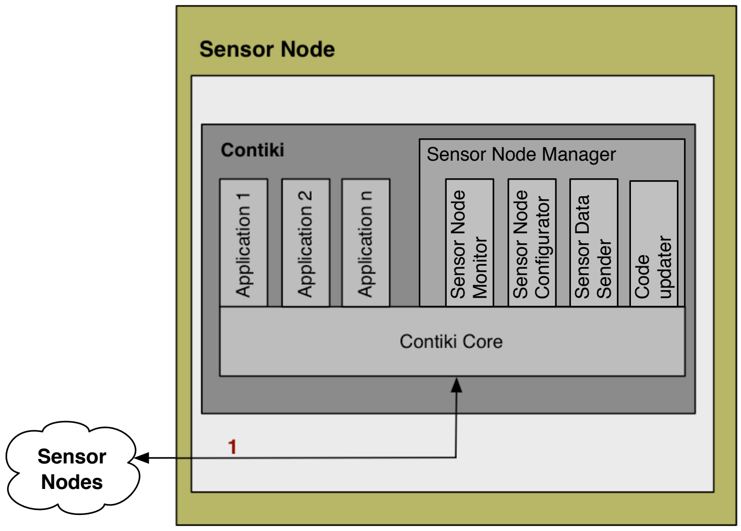

Figure 9. Sensor node architecture

The **sensor node monitor** is responsible for handling the monitor requests. It receives requests and generates packets containing the requested values and sends them back to the mesh node via the sensor node gateway using TCP or UDP (depicted **1** in Figure 9).

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 via the sensor node gateway via TCP or UDP (depicted as **1** in Figure 9).

The **sensor data sender** is responsible for sending measured sensor data. It collects the newly measured data from a sensor. Then it generates a packet with the value, the id of the sensor (prop_id) and the id of the sensor node ands sends it to the mesh node via the sensor node gateway (depicted as **1** in Figure 9).

The **code updater** has different tasks. It receives the packet with the image (differential patch, compressed, or uncompressed) and the update command. Then, if necessary, it calculates the new image from the differential patch and the old image. Afterwards it copies this into the memory and restarts the application or the operating system. And finally it sends a packet to notify the mesh node of the success of the update via TCP or UDP (depicted as **1** in Figure 9).

The **sensor node gateway** is a sensor node, which is plugged into the mesh node using USB or serial). As can be seen in Figure 10 the sensor node gateway communicates with the sensor nodes via TCP or UDP (depicted as **1** in Figure 10) and with the mesh node via TCP/IP using tunslip (depicted as **2** in Figure 10).

5.4 WSN Management Protocols

In this section the defined management tasks are described in more detail. As defined above there are four management tasks: monitoring and configuring the WSN, code updating and sensor data aggregating.

5.4.1 WSN Monitoring Protocol

Monitoring of the WSN can be divided into 3 cases. The first and the second case describes, how the management station explores the mesh network and the subordinate sensor node networks. The third case describes the situation, when the user wants to query a selected sensor directly.

When the management station joins the mesh network, it connects to the next available mesh node. Then, the management station queries the mesh node for the network topology and all stored sensor node information.

The protocol works as follows:

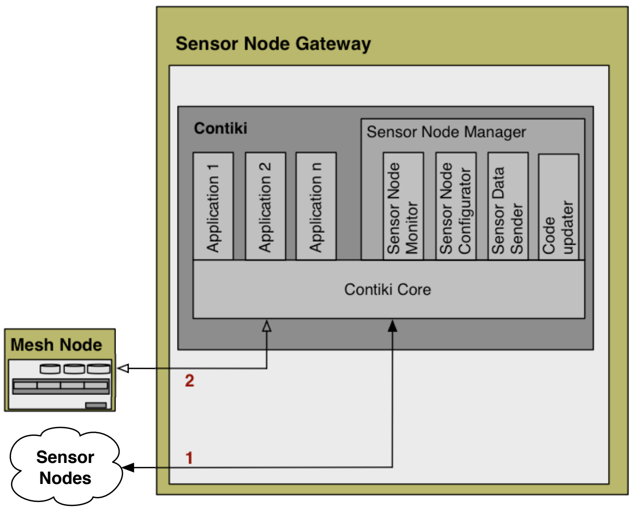

Figure 10. Sensor node gateway architecture

1. The WSN monitor asks the next mesh node via HTTPS about all available mesh nodes and the subordinate sensor nodes and all available information about the sensor nodes.
2. To answer the request, the WSN monitor module queries the database WSN information DB (using getAllMeshNodes() and getAllSensorNodes()).
3. Afterwards the management station also queries current sensor data of every single sensor node via HTTPS.
4. To answer the request, the WSN monitor module queries the sensor data DB using getSensorData().

Because of the information distribution using the CFEngine within the mesh network, the information about the WSN topology of distant mesh nodes is several minutes old and can be obsolete. For example there could be newer and still unknown sensor nodes subordinate to another mesh nodes. Therefore the management station queries all other mesh nodes for their information (Figure 11).

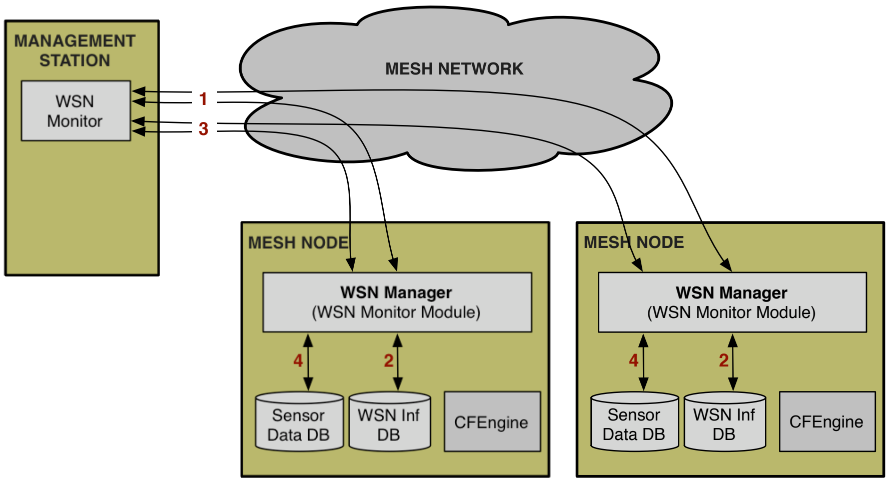

Figure 11. WSN monitor queries the mesh nodes

The protocol works as follows:

1. The management station asks via HTTPS all mesh nodes about their subordinated sensor nodes.

2. The WSN monitor modules queries the WSN information DB, but is restricted to own sensor nodes (using getAllOwnSensorNodes()).
3. Afterwards the management station also requests the current sensor data of every subordinated sensor node via HTTPS too.
4. The WSN monitor module queries the sensor data DB using getSensorData().

In the third case, the user wants to request information of a sensor node directly and not query the WSN information database on the mesh node. This is shown in Figure 12.

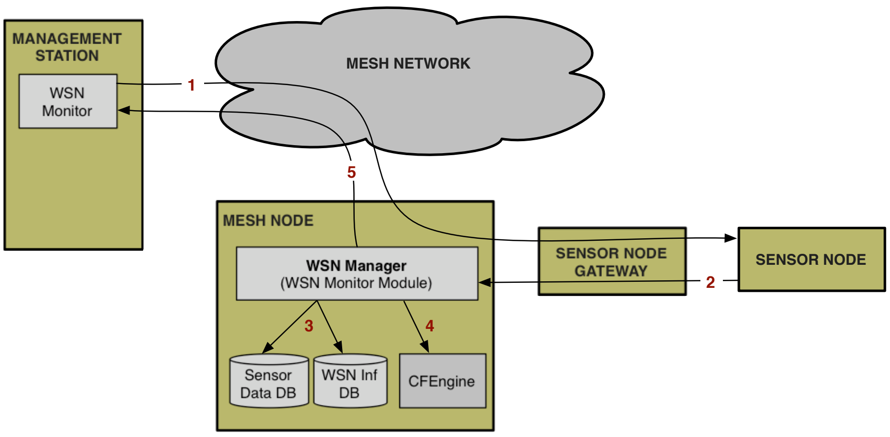

Figure 12. User requests sensor node information directly

The protocol works as follows:

1. The user requests either sensor node information (e.g. routing table) or sensor data (e.g. temperature) from a single sensor node or from a group of sensor nodes. The TCP/IP request created by the WSN monitor is forwarded to the affected sensor node gateway. The sensor node gateway sends this request via a unicast, multicast or broadcast transport protocol to the queried sensor nodes.
2. The sensor generates the response (measures data or reads information) and sends this via UDP or TCP to the mesh node, forwarded by the sensor node gateway.
3. The WSN manager module writes the new information into the according database (WSN information DB and/or sensor data DB), using insertSensorNodeProperty() and insertSensorData().
4. The WSN manager module also saves this information to the CFEngine in order to be distributed within the mesh network.
5. Furthermore, the WSN sends a confirmation to the WSN monitor.

5.4.2 WSN Configuration Protocol

The WSN configuration protocol is shown in Figure 13. Possible configuration scenarios are: turn the sensor node sensors on/off, change sensing cycles (e.g. a measurement every second minute), changing routing tables, configuring protocols, etc.

The protocol works as follows:

1. The user selects a configuration attribute, which should be configured and one single sensor node (or a group of sensor nodes), which is affected. The WSN configurator creates a TCP/IP request, which is forwarded to the involved sensor node gateway. The sensor node gateway sends this request via a unicast, multicast or broadcast transport protocol to the queried sensor nodes.

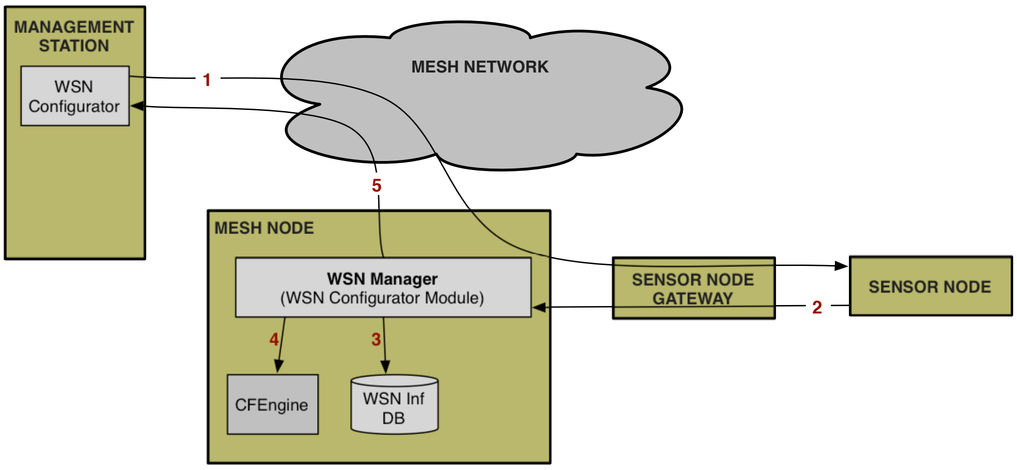

Figure 13. The WSN configuring protocol

2. On the sensor node the SN configurator performs the configuration (using configureSN()) and generates a notification (with the new configuration of the property. This notification is forwarded via the sensor node gateway to the mesh node, using UDP or TCP.
3. The mesh node stores the new configuration in the WSN information DB (using insertSensorNodeProperty()). To verify the new configuration the management station queries periodically the WSN information DB (via HTTPS and getSensorNodeProperty()).
4. On the mesh node the WSN configurator module copies the new configuration to the CFEngine (using propagateData()), which propagates it to the other mesh nodes.
5. Furthermore, the WSN sends a confirmation to the WSN configuratior.

Figure 14 shows the case of a new sensor node joining the sensor network.

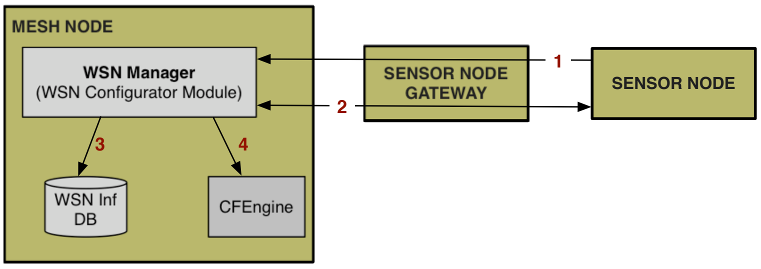

Figure 14. A new sensor node joins the sensor network

The protocol works as follows:

1. When a new sensor node joins the sensor network, it broadcasts a Hello message, which is forwarded to the WSN configurator module.
2. The WSN configurator module and the sensor node negotiate the necessary network configuration. In a second step, the WSN configurator module requests all available Information (chip, transceiver, battery, operating system, etc.) the sensor node via TCP.
3. The WSN configurator module registers the sensor node in the WSN information DB, using insertNewSensorNode() and insertSensorNodeProperty().
4. All available Information about the sensor node is also propagated to the CFEngine for distributing it within the mesh network.

5.4.3 Code Update Protocol

The code updating protocol consists of three main subtasks: uploading the new image and distributing it within the mesh network, notifying the management station about the available programs and finally performing the update. The first part of the protocol, the uploading process is shown in Figure 15.

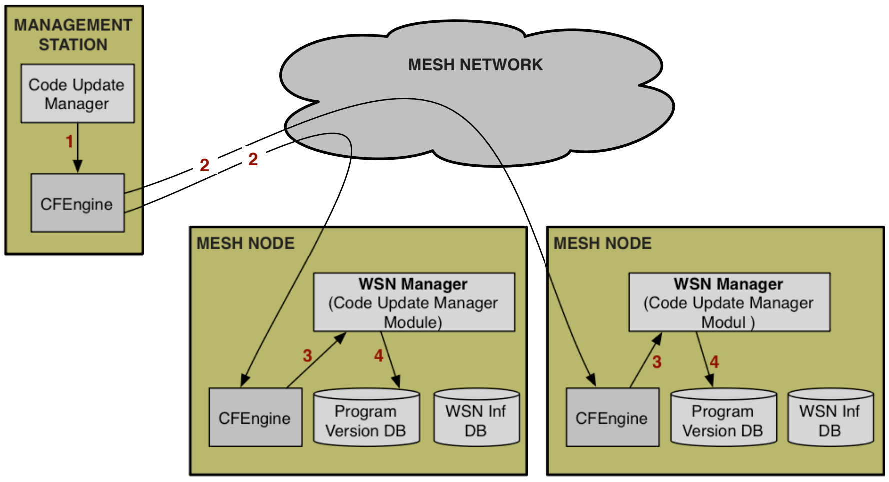

Figure 15. The CFEngine distributes the new image

The protocol works as follows:

1. Via the user interface the user selects the new image and the according information (version, components, etc), which should be uploaded. Using propagateData() the image is copied to the CFEngine.
2. Via the CFEngine these data is distributed within the mesh network to all mesh nodes.
3. The CFEngine notifies the WSN manager that a new image of a program is available (using newProgAvailable()).
4. The code update manager module in the WSN manager writes the new image plus the associated information to the program version database (using insertNewProgVersion()).

The second part of the protocol describes the notification of the management station about which programs are available in the program version database. This is shown in Figure 16.

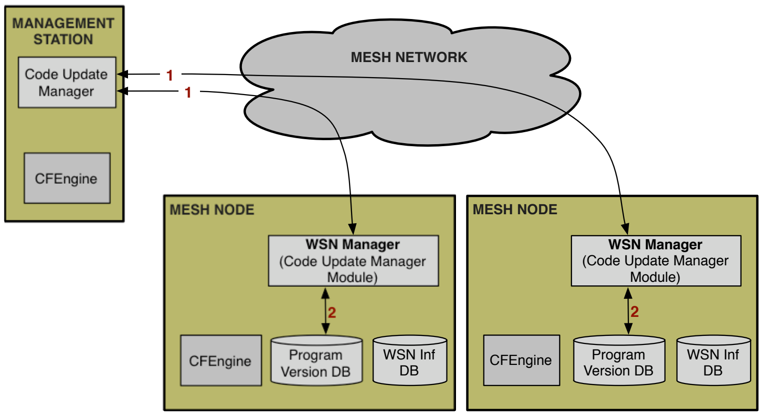

Figure 16. The user asks for all available images

The protocol works as follows:

1. In the user interface all available program versions are listed. To update this list the code update manager queries via HTTPS the WSN manager of all mesh nodes in the mesh network.
2. The code update manager module in the WSN manager queries the program version database, using getAllProgVersions().
3. Furthermore, the WSN sends a confirmation to the code update manager.

The main part of the protocol is updating the sensor nodes. This is shown in Figure 17.

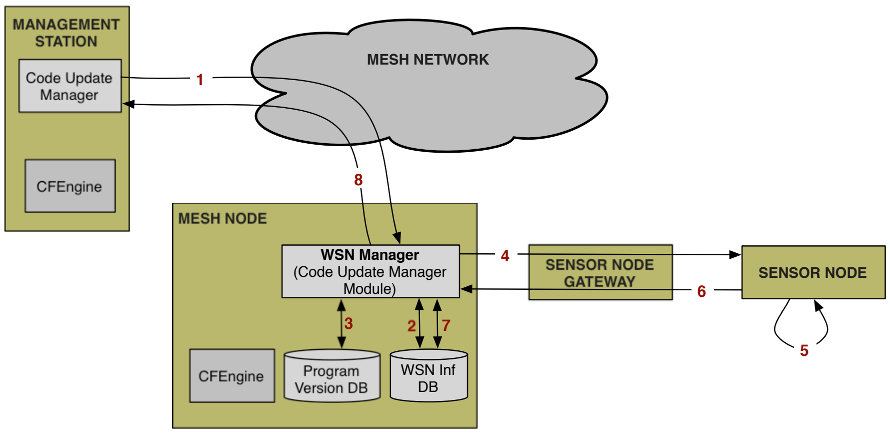

Figure 17. The user initiates the code update for the senor node

The protocol works as follows

1. In the user interface the user selects the new program version and the sensor nodes, which should be updated. The code update manager sends this request via HTTPS to the affected mesh nodes.
2. The code manager module on the mesh node checks which program version is installed on the selected sensor nodes, by quering the WSN information database (using getPropertyValue())
3. The code update manager module takes the images of the new version and the image of the old version of the selected program (using getImage()) and creates a differential patch between the new and the old image (using generateDiffPatch()). Alternatively, only the new image loaded and compressed (using compressImage()).
4. The differential patch (respectively the compressed or uncompressed new image) is sent via the sensor node gateway to the targeted sensor nodes, using the adequate unicast, multicast or broadcast transport protocol.
5. On the sensor node the update is installed. Either creating the new image from the old image and the differential patch or decompressing the new image or just copying the uncompressed new image into the memory.
6. If the update was successful the sensor node verifies this by sending the new program version via the sensor node gateway to the mesh node.
7. The code update manager module stores this new information in the WSN information database, using setPropertyValue(). The management station requests periodically the WSN information database to verify whether the update was successful.

5.4.4 Sensor Data Aggregation Protocol

The sensor data aggregation is divided into two parts. The first protocol describes in which way the sensor node sends its data to the mesh node and the mesh node distributes the data to the other mesh nodes. The second protocol describes the insertion of new data from other mesh nodes. Figure 18 shows the first protocol.

The protocol works as follows:

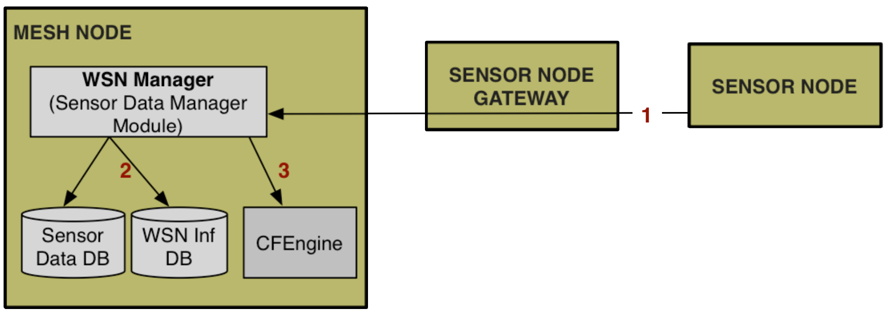

Figure 18. The SN manager module stores the senor node data in the DB

1. The sensor node generates sensor data and sends it via UDP or TCP to the mesh node, forwarded by the sensor node gateway.
2. The sensor data manager module stores the data in the sensor data DB or respectively in the WSN information DB (using insertSensorData() and insertSensorNodeProperty()).
3. For propagating the data within the mesh network, the sensor data manager module also saves this data in the CFEngine.

The second protocol continues at step 3 of the previous protocol. It is shown in Figure 19.
The protocol works as follows:

1. Via the CFEngine the new information is automatically distributed within the mesh network.
2. When new data arrives at the mesh node, the CFEngine notifies the Sensor Data Module using newDataAvailable() .
3. The Sensor Data Module inserts this new data into the according databases, using insertSensorData() and insertSensorNodeProperty().

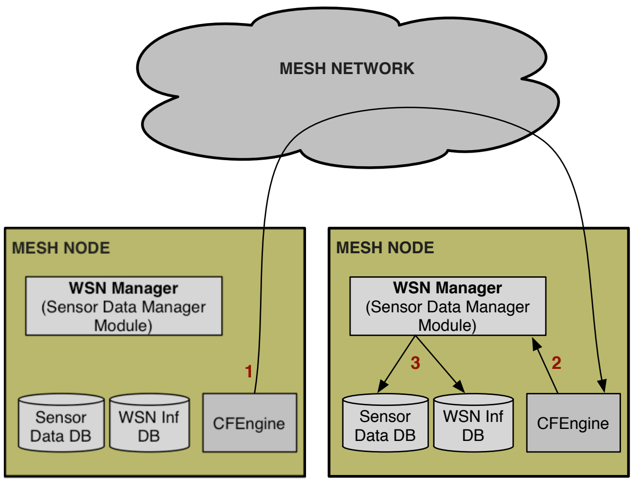

Figure 19. The CFEngine distributes all stored data to the other mesh nodes

6. Future Work

After defining the management architecture and the selection the appropriate sensor node platforms and operating system we need to accomplish next tasks. The next step is the implementation of the sensor node management architecture supporting the defined management tasks. This includes several subtasks.

The operating system Contiki has to be adapted to the different sensor node hardware. Furthermore, the IP connection between the sensor node gateway and the mesh node over a serial line internet protocol (SLIP) has to be realized. Afterwards the HTTP server with the management mechanisms on the management station, the WSN manager on the mesh nodes and the sensor node manager on the sensor node have to be implemented. Work package 2 contains the development of reliable communication mechanisms for WSN. This includes reliable transport protocols for unicast, multicast and broadcast communication. The next step is to develop a reliable point-to-point transport protocol. Concretely, this means to extend the Contiki TCP stack for Distributed TCP Caching (DTC) [20] mechanism and TCP Support for Sensor networks (TSS) [17] [31].

References

[1] C. Han, R. Kumar, R. Shea and M. Srivastava. Sensor network software update managemenet: a survey *International Journal of Network Management*, 2005, Vol. 15, pp. 283-294

[2] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. in *WSNA 03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and application*, San Diego, CA, USA, pp. 60 - 67, ACM Press, September 2003.

[3] T. Stathopoulos, J. Heidemann and 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.

[4] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In the *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom99)*, Seattle, WA, August 1999.

[5] P. Levis, N. Patel, D. Culler and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. *Proceedings of the Fi6rst USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004.

[6] Hui, D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. *SenSys 2004.*

[7] K. Rmer, O. Kasten, F. Mattern. Middleware challenges for wireless sensor networks. *SIGMOBILE Mobile Compuing and. Commuications. Reiew, Vol. 6, No. 4, October 2002, pp. 59-61.*

[8] A. Boulis, C.C. Han, M. B. Srivastava Design and Implementation of a Framework for Programmable and Efficient Sensor Networks *ACM MobiSys 2003.*

[9] T. Liu, M. Martonosi Impala: a middleware system for managing autonomic, parallel sensor systems. *in PPoPP 03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming. San Diego, California, USA: ACM Press, 2003, pp. 107 - 118.*

[10] P. Levis, D. Culler. Mat: A tiny virtual machine for sensor networks. *in Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X), 2002.*

[11] P. Levis, D. Gay, D. Culler. Active sensor networks. *in Proceedings of the Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI), 2005.*

[12] C. Cifuentes. Java(TM) on Wireless Sensor Devices. *Workshop on Wireless Sensor Networks for Enterprise Information Infrastructure, Sydney, Australia, February 2006.*

[13] L. B. Ruiz, J. Nogueira, A. Loureiro. Manna: A management architecture for wireless sensor networks. *IEEE Communications Magazine, vol. 41, no. 2, pp. 116 - 125, February 2003.*

[14] L. Ruiz, T. Braga, F. Sliva, J. Nugeira, A. Loureiro. On the Design of a Self-Managed Wireless Sensor Network. *IEEE Communications Magazine, July 2005, pp. 95-102.*

[15] P. Marron, A. Lachenmann, D. Minder, M. Gauger, O. Saukh, K. Rothermel. Management and configuration issues for sensor networks. *International Journal of Network Management, 2005, Vol. 15, pp. 235-253.*

[16] C. Fok, G. Roman, C. Lu. Mobile Agent Middleware for Sensor Networks. *An Application Case Study, In Proceedings of the 4th International Conference on Information Processing in Sensor Networks (IPSN'05), Los Angeles, California, 2005.*

[17] T. Braun, T. Voigt, A. Dunkels. TCP Support for Sensor Networks. *IEEE/IFIP (WONS 2007) , Obergurgl, Austria, January 24 - 26, 2007.*

[18] A. Chrisitan, J. Healey. Gathering Motion Data Using Featherweight Sensors and TCP/IP over 802.15.4. *Workshop on On-Body Sensing, Osaka, Japan, October 2005.*

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

[20] A. Dunkels, T. Voigt, H. Ritter, and J. Alonso. Distributed TCP Caching for Wireless Sensor Networks. *Annual Mediterranean Ad Hoc Networking Workshop, Bodrum, Turkey, June 2004.*

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

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

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

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

[25] 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 April 2007.

[26] S. Batti et al. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms, Mobile Networks and Applications. *(MONET) Journal, Special Issue on Wireless Sensor Networks, August 2005.*

[27] TinyOS. Open-source operating system designed for wireless embedded sensor networks. http://www.tinyos.net. Last visit April 2007.

[28] Contiki. A Dynamic Operating System for Memory-Constrained Networked Embedded Systems. http://www.sics.se/contiki. Last visit April 2007.

[29] 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.*

[30] S. Floyd, V. Jacobson, C. Liu, S. McCanne, L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking, December 1997, Volume 5, Number 6, pp. 784-803.*

[31] T. Braun, T. Voigt, A. Dunkels. Energy-Efficient TCP Operation in Wireless Sensor Networks. *Praxis der Informationsverarbeitung und Kommunikation (PIK), special issue on Wireless Sensor Networks, No. 2, 2005, ISSN 0930-5157, pp. 93 - 100.*

[32] 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.*

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

[34] Cfengine. A system configuration engine. http://www.cfengine.org. Last visit April 2007.