TAMPERE UNIVERSITY OF TECHNOLOGY

LASSE MÄÄTTÄ

FIRMWARE MANAGEMENT IN WIRELESS SENSOR NETWORKS

Master of Science Thesis

Examiners: Prof. Marko Hännikäinen,
Prof. Timo D. Hämäläinen
Subject approved by Department Council
13.01.2010

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
MÄÄTTÄ, LASSE OLAVI : Firmware Management in Wireless Sensor Networks
Master of Science Thesis, 76 pages
June 2010
Major: Embedded systems
Examiners: Prof. Marko Hännikäinen, Prof. Timo D. Hämäläinen
Keywords: Wireless Sensor Network

A Wireless Sensor Network (WSN) consists of autonomous wireless nodes that form an ad hoc network for monitoring environmental phenomena. Each node contains a firmware image, which consists of parameters, protocols and algorithms that are necessary for the node to function in a WSN.

The firmware images of deployed nodes can be updated to fix programming errors, introduce new features and add sensors to nodes or remove them. Analyzing the different situations and reasons results in a list of requirements for firmware management.

This thesis presents the design, implementation and experimental measurements of firmware management for the Tampere University of Technology WSN (TUTWSN). Firmware management consists of three server side and three node side components. On the server side, the user interface has been developed for network administration, a firmware management database for storing firmware images and parameters, and the Autoconfigurator for transferring images from the database into WSNs. On the node side, the firmware image transfer protocol disseminates firmware images between nodes, the bootloader monitors the integrity of the stored firmware image, and the firmware parameter transfer protocol is responsible for parameters.

Firmware management was implemented on a TUTWSN platform with an 8-bit 2 MIPS Microchip PIC18LF8722 microcontroller and a 2.4 GHz Nordic Semiconductors nRF24L01 radio. Firmware management requires less than 7 kilobytes of program memory. Experimental measurements with hundreds of nodes in practical WSNs have been executed. Based on the results, updating a single node wirelessly takes less than ninety seconds, while a large scale WSN of 268 nodes can be updated in five hours.

Firmware management has been shown to be a reliable method for resource constrained WSNs. It has reduced the amount of manual work, increased production yields, and added more flexibility in maintaining large WSNs. Although firmware management was implemented for TUTWSN, the presented design is not tied to it, but is applicable to other state of the art WSN architectures as well.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
MÄÄTTÄ, LASSE OLAVI : Sulautetun ohjelmiston hallinta langattomissa sensoriverkoissa
Diplomityö, 76 sivua
Kesäkuu 2010
Pääaine: Sulautetut järjestelmät
Tarkastajat: Prof. Marko Hännikäinen, Prof. Timo D. Hämäläinen
Avainsanat: Langattomat sensoriverkot

Langaton sensoriverkko koostuu autonomisista langattomista mittalaitteista, jotka muodostavat ad hoc verkon valvoakseen ympäristössä havaittavia ilmiöitä. Jokainen mittalaite sisältää sulautetun ohjelmiston, joka koostuu parametreista, protokollista sekä algoritmeista, jotka vaaditaan, että mittalaite voi toimia osana sensoriverkkoa.

Sulautettu ohjelmisto voidaan päivittää, jotta ohjelmistovirheitä voidaan korjata, uusia ominaisuuksia lisätä sekä liittää sensoreita mittalaitteeseen tai poistaa niitä. Kun tarkastellaan eri tilanteita saadaan aikaiseksi vaatimukset sulautetun ohjelmiston hallinnalle.

Tässä työssä esitellään langattomien sensorien sulautetun ohjelmiston hallinnan suunnittelu, toteutus sekä kokeelliset mittaukset käyttäen hyväksi Tampereen Teknillisessä Yliopistossa kehitettyä TUTWSN sensoriverkkoa. Sulautetun ohjelmiston hallinta koostuu kolmesta palvelinkomponentista sekä kolmesta mittalaitekomponentista. Palvelinpuolella on kehitetty käyttöliittymä hallinnoimaan verkkoja, tietokanta tallettamaan sulautettuja ohjelmistoja ja parametreja, sekä Autoconfigurator siirtämään ohjelmistoja tietokannasta sensoriverkkoon. Mittalaitteen puolella sulautettujen ohjelmistojen siirtoprotokolla siirtää ohjelmistoja mittalaitteiden välillä, käynnistyslataaja valvoo tallennetun ohjelmiston eheyttä ja sulautetun ohjelmiston parametrien siirtoprotokolla on vastuussa parametreista.

Sulautetun ohjelmiston hallinta on toteutettu käyttäen langattomia TUTWSN laitealustoja, joissa on 8-bittinen 2 MIPS:in Microchip PIC18LF8722 mikrokontrolleri sekä 2.4 GHz Nordic Semiconductors nRF24L01 radio. Sulautetun ohjelmiston hallinta vaatii alle 7 kilotavua ohjelmamuistia. Kokeelliset mittaukset suoritettiin todellisissa, satojen langattomien mittalaitteiden verkoissa. Tulokset osoittavat, että yksittäisen mittalaitteen päivitys kestää alle 90 sekuntia, kun taas laajan sensoriverkon, jossa on 268 mittalaitetta, päivittäminen kestää viisi tuntia.

Sulautetun ohjelmiston hallinnan on osoitettu toimivan luotettavasti resurssirajatuissa langattomissa sensoriverkoissa. Se on vähentänyt työmääriä, kasvattanut tuotannon saantia sekä lisännyt joustavuutta laajojen langattomien sensoriverkkojen ylläpidossa. Vaikka sulautettujen ohjelmistojen hallinta toteutettiin käyttäen TUTWSN sensoriverkkoja, työssä esitetty suunnittelu ei ole sidottu siihen, vaan on sovellettavissa myös muissa moderneissa sensoriverkkoarkkitehtuureissa.

PREFACE

The work for this thesis was carried out at Tampere University of Technology in the Department of Computer Systems in 2008-2010 at the DACI research group.

I would like to express my gratitude to my thesis supervisors Prof. Marko Hännikäinen, and Prof. Timo D. Hämäläinen for their invaluable coordination and support during the course of this work.

I am grateful to Mr. Teemu Laukkarinen, Mr. Jani Arvola, B.Sc. and Mr. Jukka Haaramo, M.Sc. for their help and guidance. Also, I would like to thank Mr. Mika Vuori, M.Sc., Mr. Tomi Jäntti, M.Sc., and Mr. Jukka Suhonen, M.Sc. for their expertise and insights during this work. In addition, I would like to thank Mr. Ilkka Kautto, B.Sc., for helping me in the evaluation phase. Finally, I would like to thank the rest of my colleagues at the DACI research group for creating a pleasant and inspiring working atmosphere.

Most of all, I would like to thank my parents Reijo and Elina, my grandmother Tuula, and my girlfriend Inge for their endless support during my studies.

Tampere, June 3rd, 2010

Lasse Olavi Määttä
Tekniikankatu 10 B 74
33720 Tampere
FINLAND

tel.	+358 50 361 2083
e-mail	lasse.maatta@tut.fi

CONTENTS

LIST OF ABBREVIATIONS

API	Application Programming Interface
APS	Application Support
EEPROM	Electronically Erasable Programmable Read-Only Memory
FFD	Full-Function Device
FTDMA	Frequency and Time Division Multiple Access
GWI	Gateway Interface
HAL	Hardware Abstraction Layer
IP	Internet Protocol
IrDA	Infrared Data Association
ISM	Industry, Science and Medicine
ITU	The International Telecommunication Union
LAN	Local Area Network
LED	Light Emitting Diode
LLC	Logical Link Control
LR-WPAN	Low-Rate Wireless Personal Area Network
MAC	Medium Access Control
MIB	Management Information Base
NCAP	Network Capable Application Processor
OSI	Open Systems Interconnection
PAN	Personal Area Network
PCB	Printed Circuit Board
PDU	Protocol Data Unit
QoS	Quality of Service
RFD	Reduced-Function Device

SNAP	Sensor Network Application Platform
SNMP	Simple Network Management Protocol
SSP	Security Service Provider
STIM	Smart Transducer Interface Module
TCP	Transmission Control Protocol
TEDS	Transducer Electronic Data Sheet
TUTWSN	Tampere University of Technology Wireless Sensor Network
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network
WSN	Wireless Sensor Network
XML	Extensible Markup Language
ZDO	ZigBee Device Object

LIST OF TABLES

LIST OF FIGURES

LIST OF PROGRAMS

1. INTRODUCTION

Ubiquitous computing is a term that covers a broad range of technologies that enable computers and their interfaces to be embedded seamlessly within our natural environment [49]. Embedded ubiquitous devices have different capabilities and limitations, thus they need to cooperate to perform tasks and to gather, process and present information to the end-user. One promising technology in the field of ubiquitous computing is Wireless Sensor Networks (WSNs).

A WSN is a network of independent wireless *nodes* that are equipped with sensors that measure local physical quantities, such as temperature or humidity, and actuators that control other devices [1]. Each node communicates with its neighbors with a radio to form an ad hoc wireless communication link, as shown in Figure 1.1. These links are then used to transfer measurement data and commands between the nodes. In a *multihop* WSN, nodes route data from one node to another through the WSN, until the data reaches a gateway node at the edge of the network. The transferred data is then processed by a network gateway and presented using end-user services.

Resource-constrained WSNs use hundreds of tiny low-cost nodes with very little computation and storage capabilities [25]. These nodes are battery powered and therefore have a limited amount of available energy further restricting their performance. Conversely, application scenario requirements for a WSN often require nodes to operate for months or even years. This requirement must be addressed in both hardware and software development for WSNs. In this thesis, the term WSN is considered to describe these resource-constrained multihop WSNs.

1.1 Node Firmware

In addition to the hardware, each node contains software. In this thesis, node software is divided into *applications* and *firmware*. Applications perform high-level tasks, such as performing measurements or gathering diagnostics data. Firmware contains critical protocols and algorithms that enable a node to communicate with its neighbors and execute applications. An instance of a compiled firmware source code is called a *firmware image*. Each node must be programmed with a firmware image before it can function in a WSN.

Multiple different parameters and options define how the firmware should function, as shown in Figure 1.2. First, WSNs might use different versions of protocols,

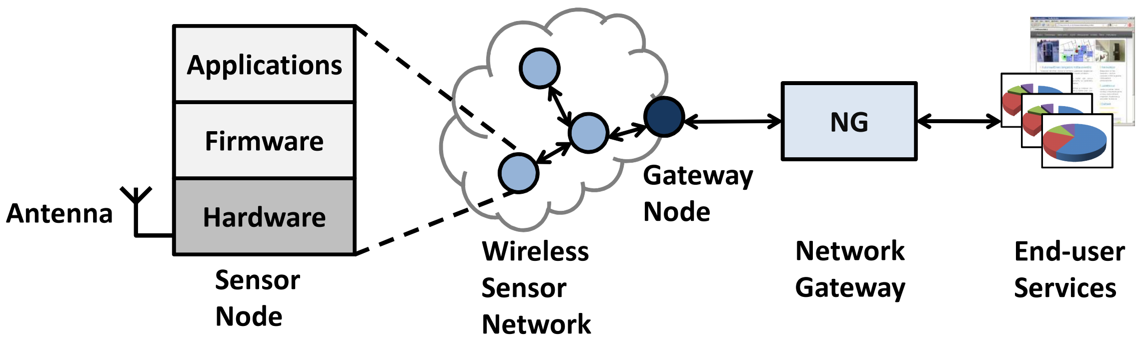


Figure 1.1: The basic components of a WSN. Data gathered by node applications is transferred through a WSN and presented as charts and diagrams with end-user services.

algorithms and interfaces depending on when and how they were deployed. These global parameters must be the same in multiple WSNs at the same time. Second, WSNs can have different application requirements for latency, throughput, lifetime or reliability. These WSN-specific parameters must have the same value in each node in a particular WSN. And finally, nodes may be equipped with various sensors or actuators, and they may be chosen to perform additional tasks, such as generating additional network traffic for testing purposes. These node-specific parameters only apply to a limited number of nodes in a particular WSN. In this thesis, this set of different parameters that are required to configure a firmware image are called *firmware parameters*.

1.2 Firmware Management

WSNs are often deployed for months or years in harsh conditions as they have a long lifetime, require very little existing infrastructure and exhibit a high level of redundancy. In addition, access to the nodes and manual maintenance can be difficult or even impossible. The long lifetime makes it highly likely that algorithm and programming errors are discovered and fixed while multiple WSNs are still in active use. Furthermore, it is not uncommon for the requirements of a WSN to change during the deployment as new features and capabilities are requested by end-users. These reasons also apply to smaller development WSNs that are used for testing WSN algorithms and protocols. Although these testing WSNs are often small in scale and easy to access, they must be updated much more frequently as new software errors are found and new features are tested. Thus, a wireless reprogramming mechanism is required for updating the firmware images in deployed nodes. Not all WSNs architectures support reprogramming nodes and, therefore, only those WSNs that allow it are discussed in this thesis.

In addition to changes in the firmware, there are also several reasons why the

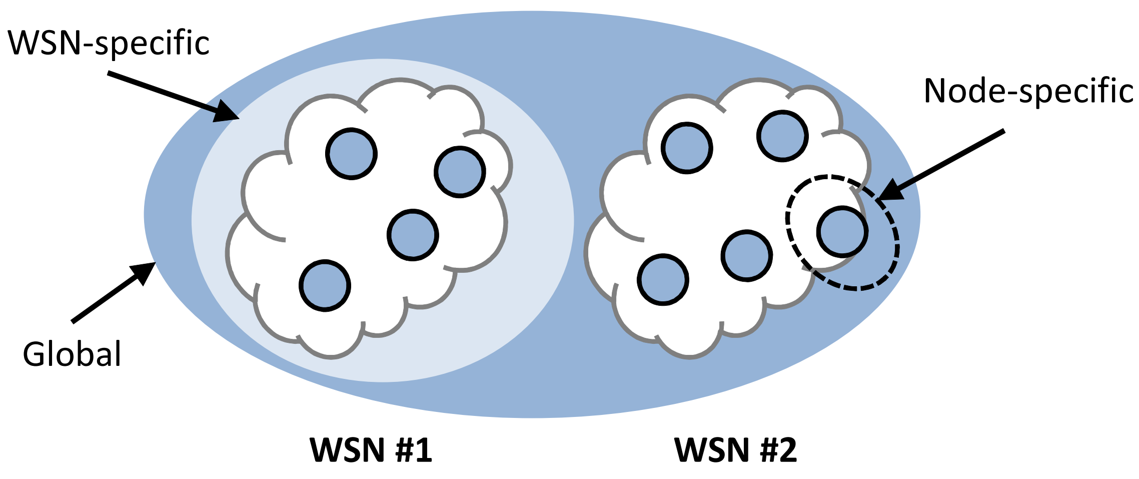


Figure 1.2: A firmware parameters value may either apply to all nodes in multiple WSNs (Global), to all nodes in a single WSN (WSN-specific) or to only specific nodes in a WSN (node-specific) depending of the parameters scope.

firmware parameters of deployed nodes must sometimes be updated. End-users often wish to extend their existing WSNs by adding new nodes or by attaching new sensors or actuators to existing nodes, which requires the firmware parameters of those nodes to be changed. The manufacturing costs of current node designs [9] have been too high for the nodes to be disposable in actual commercial use. Therefore, nodes from one deployment are usually reused in later ones by updating their firmware parameters to match the new WSN.

Updating the firmware parameters requires that the corresponding firmware image is also updated. This can be done by recompiling the source code of the old firmware images and then reprogramming the nodes. However, this can take significant time and effort, if hundreds of nodes must be reprogrammed. In addition, as the number of deployed WSNs and nodes rises, the task of tracking the different firmware images and their parameters for long periods of time becomes difficult and error prone.

This thesis presents the design, implementation and experimental measurements of *firmware management* for WSNs. Firmware management consists of three server side components and three node side components, as shown in Figure 1.3:

Firmware management user interface: Firmware images and parameters are added, configured and removed with a graphical user interface running on a personal computer.

Database storage: Firmware images and parameters are stored and mapped to WSNs and nodes in a relational database, where they can be later retrieved or updated.

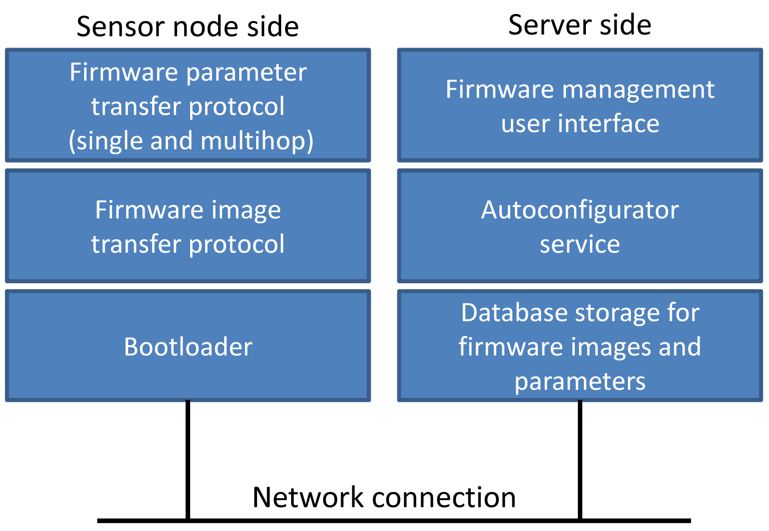

Sensor node side
Server side
Firmware parameter transfer protocol (single and multihop)
Firmware management user interface
Firmware image transfer protocol
Autoconfigurator service
Bootloader
Database storage for firmware images and parameters
Network connection

Figure 1.3: The scope of this thesis is the design and implementation of both node side and server side firmware management components.

Autoconfigurator service: The autoconfigurator service monitors firmware images and parameters in the database and makes sure that new firmware images and parameters are transferred to the correct nodes.

Firmware parameter transfer protocols: Firmware parameters are retrieved from the database with the autoconfigurator service and transferred to a target node using a firmware parameter transfer protocol. Initial firmware parameters are transferred to nodes using a single-hop protocol when they are powered on for the first time. Once the node has joined a WSN, a multihop protocol can be later used to reconfigure the deployed node by sending the firmware parameters through the multihop WSN.

Firmware image transfer protocol: Firmware images are injected into a WSN and disseminated between nodes using an energy efficient firmware image transfer protocol. nodes use software advertisements to compare information about their firmware images and then use the transfer protocol to exchange firmware images if necessary.

Bootloader: The bootloader increases the reliability of firmware transfers by making sure that only successfully received firmware images are executed. If the firmware image is missing or it is damaged, the bootloader automatically begins executing the firmware image transfer protocol to fetch a new firmware image.

The presented firmware management components have been designed with a practical point of view to meet the requirements of real world WSN deployments with hundreds of nodes deployed in remote locations. The firmware management components were implemented using Tampere University of Technology Wireless Sensor Network (TUTWSN) [12], which is a state of the art multihop WSN technology. The limitations and capabilities of TUTWSN are comparable to many other WSN technologies in general.

TUTWSN includes simulation tools for WSNs, energy-efficient communication protocols, an embedded operating system, several different platform designs and a comprehensive software infrastructure consisting of both existing applications and services, and methods for designing and implementing new applications [25].

Several thousands of TUTWSN nodes have been deployed in almost a hundred different WSN installations since 2007. Although the presented firmware management components were created for WSN deployments with TUTWSN, the design of firmware management is applicable to other WSN technologies as well.

This thesis is organized as follows. Chapter 2 presents an overview of WSN concepts, technologies and standards that form the technical constraints when designing and implementing software for WSNs. The architecture of TUTWSN is presented in Chapter 3. Requirements for firmware management are presented in Chapter 4 by following the lifetime of a node from manufacturing to deployment and reuse. Related technologies and proposed firmware transfer protocols and firmware management mechanisms offered by existing WSN operating systems are discussed in Chapter 5. Chapter 6 presents the design of firmware management in TUTWSN. Evaluation of firmware management in TUTWSN is given in Chapter 7. This thesis is concluded in Chapter 8.

2. WIRELESS SENSOR NETWORKS

To understand the background behind TUTWSN and firmware management, the features of typical WSN must be first explored. This chapter first presents the characteristics, capabilities and limitations of WSNs. Second, the layout of a deployed WSN is discussed. Third, the characteristics of node hardware and software are presented with a comparison to the Open Systems Interconnection (OSI) model. At the end of this chapter, the Institute of Electrical and Electronics Engineers (IEEE) 802.15 standard and the ZigBee specification are presented as examples of popular WSN standards.

2.1 WSN Characteristics

There has been a significant research interest in WSNs, as they offer major benefits over traditional wired sensors. nodes can be freely deployed around the area of interest as they can communicate wirelessly and contain their own power source. Thus, WSNs require very little existing infrastructure. Due to the low cost of nodes, the network coverage and the level of redundancy can be increased by deploying sufficient quantities of additional nodes. In addition, the loss of individual nodes, due to e.g. hardware failure, is tolerable due to this high level of redundancy. Several common characteristics of typical WSNs are discussed in [25] and in [1].

Fault tolerance: Individual nodes are not reliable, as they may be destroyed by environmental conditions: moisture, shock, and freezing, or may run out of energy. WSNs may be deployed in harsh conditions and while some nodes may fail, the WSN itself must remain operational. To achieve this fault tolerance and level of redundancy, nodes must be deployed in sufficiently large quantities.

Self-organizing: As nodes are deployed in a adhoc manner, it is imperative that WSNs are able to generate the necessary routes through the network automatically. This self-organizing capability also has a significant impact on the fault tolerance of WSNs. If one or more nodes become inoperable or a section of the WSN is suffering from temporary communication interference, the WSN must be able to reroute new paths around the affected area.

Performance: Unlike traditional wireless network technologies like Wireless Local Area Networks (WLANs), the performance evaluation of WSNs is not based upon latency or throughput. Instead, WSNs are evaluated on how well they fill the application requirements, such as reliability, lifetime and cost.

Rapid deployment: nodes can be freely placed around the area of interest, as long as each node is close enough to the rest of the WSN to form a radio link. This allows for rapid deployment compared to traditional sensor networks, as no manual cabling or choosing precise node locations is required.

Long lifetime: Extending the lifetime of nodes is one of the goals of WSN research [4]. WSNs may be deployed in locations where power supply is limited and the maintenance of nodes is difficult or impossible. Replacing node batteries is impractical, due to the large number of nodes in a typical WSN. One possible solution is to harvest energy from the environment using solar panels or peltier elements, but this is often inapplicable due to the high cost or if the nodes are installed indoors. A more robust solution is to minimize the energy consumption of nodes, which requires significant effort in both hardware and software design.

Application specific: The benefits of WSNs over traditional sensors give them a large application area, which cannot be sufficiently covered by any single WSN technology. Therefore, WSNs are customized with application-dependent communication protocols, hardware platforms, and end-user applications to meet the requirements of the target application.

This combination of high reliability, low maintenance and rapid deployment make WSNs a promising technology for many usage scenarios. These benefits of WSNs, combined with a wide range of applications such as environmental monitoring, object tracking and classification and actuator control of external devices, give WSNs a wide application area.

Military: WSNs can be used in military situations to monitor activities on a battlefield. WSNs can detect enemy combatants and vehicles, or they may be used to monitor the presence of hazardous gasses. Friendly units may be equipped with mobile sensors to provide realtime location information. In one instance, a WSN was deployed to estimate the trajectory of incoming projectiles [24].

Health and safety: WSNs are very suitable for monitoring the working conditions in office environments. nodes can monitor e.g. air quality and noise levels [39] or control ventilation and lighting [38]. High scalability allow WSNs to be deployed in both small offices and large office centers spanning several floors. Their short deployment time enables temporarily deployed WSNs to be rapidly moved from one office to another.

Environment: Outdoor deployments of large-scale WSNs enable monitoring of wildlife: the nesting of birds [27] or tracking of animals [18], and agricultural use in farms [35] or orchards [52]. WSNs may be used to detect and analyze natural phenomena: forest fires [23] or volcanic activity [50].

2.2 Wireless Sensor Network Architecture

Nodes have very little storage space and an user interface that often consists of only a few push buttons and Light Emitting Diodes (LEDs). Therefore, additional components are needed to store and present the gathered data, control access to the WSN and monitor the WSN performance. In this thesis, the term *WSN architecture* is used to describe the minimum set of necessary components that are needed to allow end-users to interact with the WSN. Besides the physical WSN, this architecture contains a network gateway software, which may be connected to multiple WSNs, and external applications. The physical layout of a WSN application scenario is shown in Figure 2.1.

A network gateway presents a high-level interface to a WSN. Thus, external applications may access the WSN without knowing the specific details of the underlying WSN technology. A network gateway can offer multiple standard interfaces for accessing a WSN, such as the Extensible Markup Language (XML) or the Simple Object Access Protocol interfaces. A network gateway may perform access control in either direction. Thus, a network gateway can limit which nodes may send measurements to the external applications and which applications or users are allowed to access the WSN.

External applications retrieve information from WSNs through network gateways. This information can be refined and stored in a database. In addition, multiple applications may share the refined data. External applications have three main duties. First, they present the measured data to the end user. Measurement data can be presented by generating periodical reports, where measurements from multiple sensors are presented in graphs or tables. Second, applications can visualize the network topology by graphing the node locations on a map view. And third, external applications allow end users to manage the WSN and the network gateway. Applications can organize WSNs by selecting which nodes belong to which WSNs, by giving nodes names or descriptions and by choosing which user accounts may access those WSNs. In addition, applications can control WSN measurement services by choosing what measurements are performed and what the measurement interval is.

The application scenario defines the goals that the WSN must meet, such as measurement interval or network lifetime. A WSN consisting of multiple nodes is deployed around a phenomenon. Nodes are placed in an adhoc manner without extensive planning or link quality measurements. Each node contains multiple different

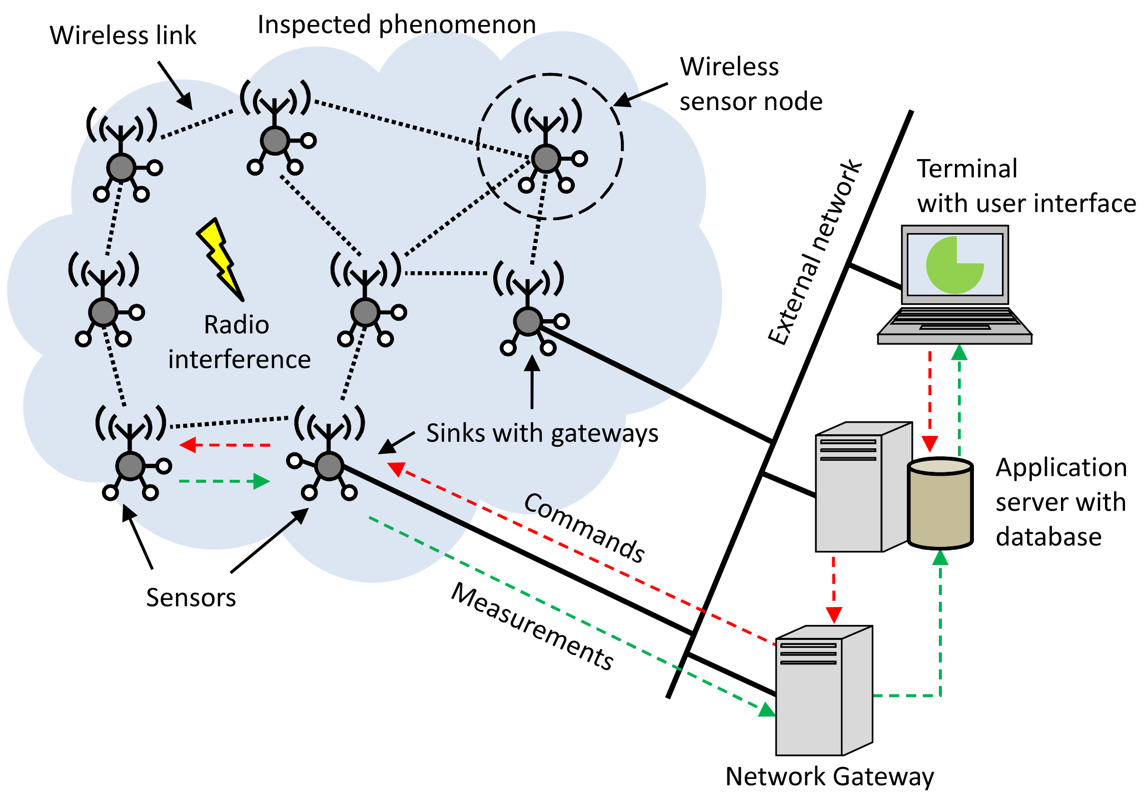


Figure 2.1: The physical layout of a WSN application scenario [21].

sensors that monitor the phenomenon. In addition, each node forms a communication link with its neighbors. The number of nodes in the WSN is not constant, as nodes may be added or removed at any time. The routing paths for the data also change with time, as sporadic radio interference causes nodes to find new neighbors and routes.

WSNs can be heterogenous, as nodes often contain different sensors. In addition, some of the nodes can be equipped with a gateway module that allows these *gateway nodes* to transfer data to external networks, such as Local Area Networks (LANs).

Gateway nodes form a connection to an external network. This allows gateway nodes to act as so called *sinks*, as the measurement data flows towards them. The gateway nodes then forward the data to the application server on the external network. It is important to note that a sink and a gateway node are not always the same thing. A handheld mobile terminal could directly access the WSN and gather measurement data without transferring the data to other networks. Thus, the mobile terminal would act as a sink but not as a gateway to another network technology.

The application server hosts the network gateway software and external applications, which refine the measurement data and store it in a database. End users may use terminals to access applications running on the application server. Con-

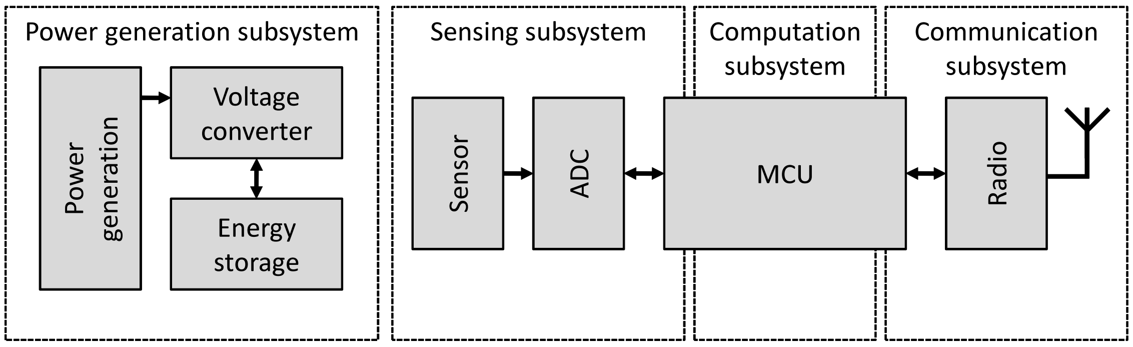


Figure 2.2: General hardware architecture of a WSN node [22].

versely, end users may use local applications running on the terminal to retrieve the measurement data straight from the database.

2.3 Wireless Nodes

The elementary building block of any WSN technology is the node. A typical hardware platform has four main hardware subsystems, as shown in Figure 2.2: the power generation, sensing, computation and communication subsystems. The power generation component contains a power source, which typically is a battery or a solar cell, and the necessary voltage regulation components. The sensing component contains sensors that measure both external and internal quantities, such as the battery voltage or temperature. The computation component contains a microcontroller, which stores and executes the node software, accesses the sensor devices and communicates with other nodes through the communication component, which includes a radio unit and optional interface busses.

node design and capabilities are limited by three primary requirements [25] [1].

Low-cost: Increasing the performance, reliability and lifetime of a node may increase the manufacturing costs substantially. In addition, this could lead to WSNs that consist of a small number of expensive nodes and a limited network size and coverage. Instead, a more natural approach is to increase the amount of nodes by decreasing their cost and performance.

Small size: A typical goal of node design is to minimize the physical size as it makes storage and deployment easier. In addition, small size is valuable when it is desirable that nodes blend in with the environment. Conversely, minimizing the physical size sets limitations on the size of the components and the energy storage capacity. An early example of node miniaturization was the Smart Dust-project [20], which was based on using Micro-Electro-Mechanical System

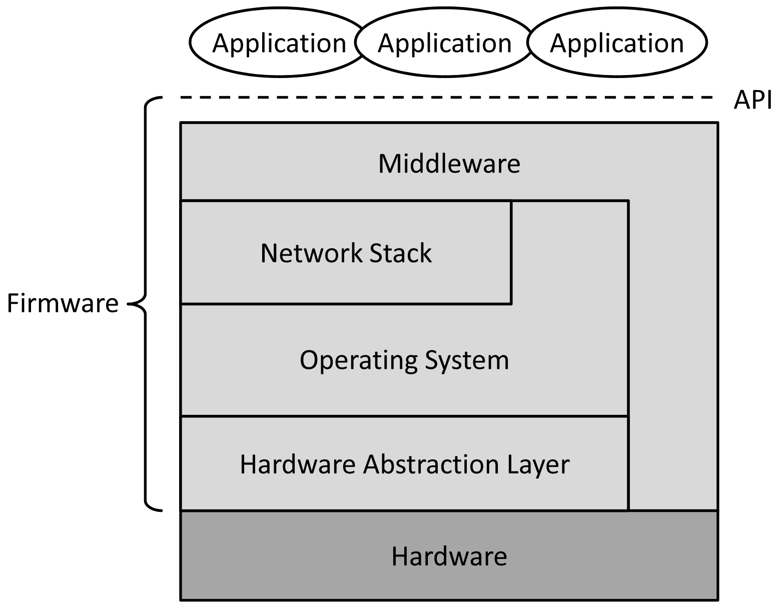


Figure 2.3: General software architecture of a WSN node [25, page 104].

technology. The target was to create nodes, which were the size of a grain of sand.

Long lifetime: Extending the node lifetime is limited by the energy consumption. Any increase in performance, whether it is increased computational performance, storage space or communication capacity, inevitably leads to increased energy consumption and often also increases the physical size and cost of nodes. Thus, it is crucial that a balance is found between node lifetime, performance and requirements of the application scenario.

The node software stack can be divided into six parts, as shown in Figure 2.3. The Hardware Abstraction Layer (HAL) presents the hardware platform with an abstract interface, which other software components use to access the platform. Abstracting the platform-specific details allows porting the upper layers of the software stack to other node platforms that are compatible with the same HAL. The operating system in WSNs makes development easier by managing the resources of the node, such as memory and timers, and by containing device drivers, algorithm libraries and data structures [25]. The network stack contains communication protocols that allow a node to reliably communicate with other nodes. The middleware connects together other software components and implements an Application Programming Interface (API) that allows applications running on the node to access the operating system services and the hardware components, regardless of the underlying operating system or the WSN architecture.

The network stack of the node can be further divided into layers according to

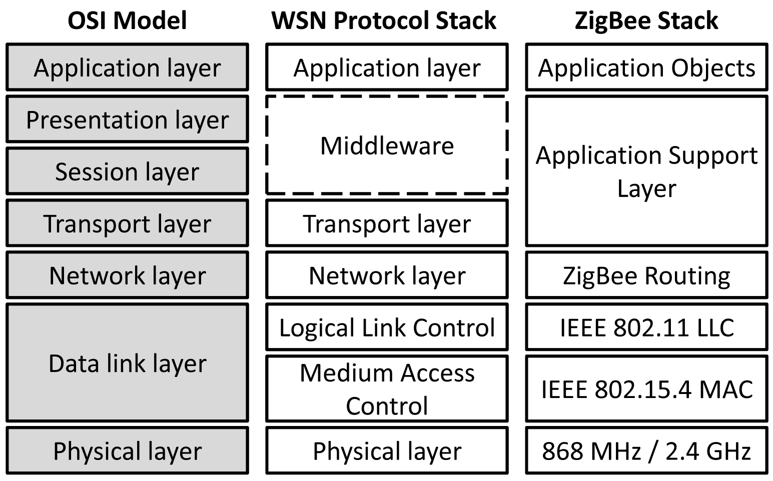


Figure 2.4: The OSI model compared to a WSN protocol stack and the ZigBee protocol stack.

the OSI model [17]. Comparison of a WSN protocol stack with the OSI model is shown in Figure 2.4. As the OSI model is designed for general purpose networking solutions, all of its layers are not necessary for WSNs. Thus, although the OSI model consists of seven layers, WSN protocols only use five of them. The protocol stack layers of the OSI model are:

Physical layer: As the lowest layer of the OSI model, the physical layer defines the physical specifications of the communication medium, such as cable specifications and voltages. In WSNs, the radio implements the physical layer by selecting the correct frequency channel, transmission power and modulation method.

Data link layer: The data link layer performs medium access, error detection and correction and sends data over the physical link. In WSNs the data link layer often contains a Medium Access Control (MAC) and a Logical Link Control (LLC) sublayer. The MAC decides when and how the physical layer is operated. Using the radio may consume significant amounts of energy. In addition, simultaneous transmissions from different nodes should be avoided as they may collide and corrupt each other. Thus, the MAC layer has a significant impact on the energy efficiency and performance of the WSN. The LLC layer frames message segments and manages link layer addressing between nodes by e.g. marking each message with a sender and receiver address. In addition, the LLC can handle sequencing information and CRC checking.

Network layer: The primary task of the network layer is to form a routing path from the source to the destination by connecting neighboring nodes on the data

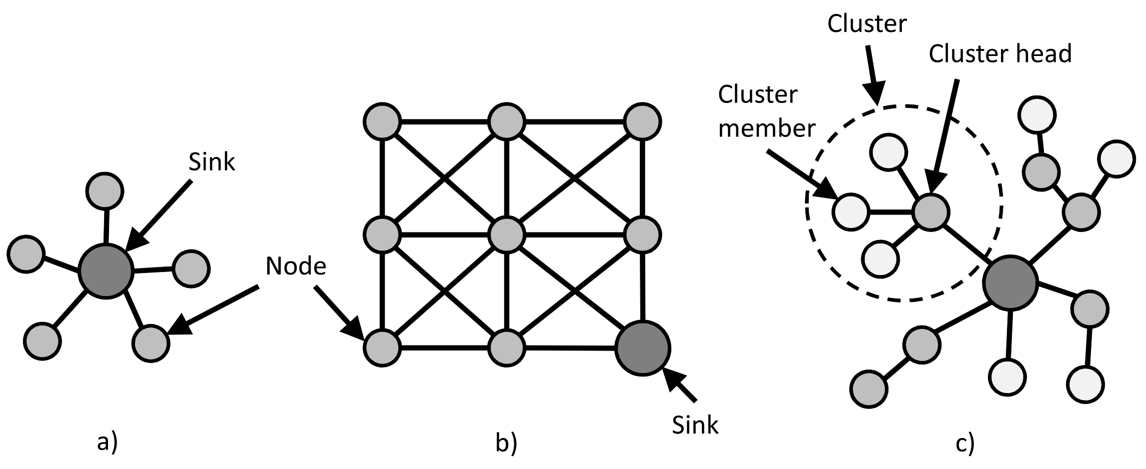


Figure 2.5: Typical WSN topologies. The star topology in a) is limited in coverage but requires little energy and resources from the nodes. The mesh topology in b) is scalable but increases the energy usage and communication overhead. The cluster tree topology in c) is a viable alternative that allows the leaf nodes of the tree to be resource limited without limiting the network coverage.

link layer. In WSNs, the network layer collects neighbor information from the data link layer, chooses the optimal neighbors and then forms communication links with them, as shown in Figure 2.5. The star topology limits the network coverage to the radio range of the network sink. Mesh networks offer extended coverage but also increase overhead costs as each node must communicate with all of its neighbors. The cluster tree topology is a compromise between the simplicity of the star topology and the scalability of the mesh topology. Furthermore, the network layer performs self-configuration by monitoring the reliability of individual communication links. When routing data, the network layer selects the next hop destination by comparing the current neighbors. Choosing the best next hop destination is not trivial, as the requirements for different application types differ. High priority alarm data may prefer a low latency connection while low priority measurement data should use links that conserve energy.

Transport layer: The transport layer performs two tasks. First, it performs flow control to avoid network congestion. Second, it performs additional error checking to detect transmission errors that were not detected by the LLC in the data link layer. Link reliability in WSNs is much worse than in wired networks and the data memory capacity for buffering data in nodes is severely limited. Thus, flow and error control are typically performed separately for each link.

Session layer: The session layer manages logical connections by opening, closing and restarting connections between endpoints. This layer is not typically used in WSNs, as connections from nodes to sinks are not explicitly opened and closed [25].

Presentation layer: The presentation layer maintains the compatibility of data formats between the endpoints. It may modify the byte order of the data or alter the format of strings. Encryption is often performed at the presentation layer. One of the goals of WSN software design is to present the underlying hardware and network protocols for applications by using a high level interface provided by the middleware [25]. The middleware can also manage the Quality of Service (QoS) for end-user applications. Thus, WSN middleware can be seen as a part of the presentation layer.

Application layer: At the highest layer of the OSI model, the application layer communicates with the end-user applications to receive and transmit messages. The application layer in WSNs typically offers an interface that supports tasks running in parallel, such as measuring different sensors or monitoring the available resources of the node.

2.4 WSN Standards and Proposals

The purpose of standards is to enable interoperability between the protocol stacks of devices from different manufacturers [25]. As WSNs are a new field of technology, the standardization of WSN components has only been started recently. Therefore, many existing WSN products are based upon proprietary solutions that are incompatible with each other [28].

The IEEE has begun the standardization of WSN technologies and created the IEEE 802.15 standard [45] for Wireless Personal Area Networks (WPANs). WPANs are low range wireless networks formed between personal devices, such as mobile phones or Personal Digital Assistants. WPANs allow these devices to both transmit data between each other, and access infrastructure networks such as LANs. Examples of WPAN technology are Bluetooth [11] and Infrared Data Association (IrDA) [2].

The IEEE 802.15 standard contains seven task groups for defining different types of WPANs. The IEEE 802.15.1 task group has created a standard for the physical and MAC layers for WPANs using the Bluetooth specification. The 802.15.2 task group is working on the issues of coexistence between WPANs and other wireless networks.

The IEEE 802.15.3 standard describes the physical and MAC layers in high rate WPANs for bandwidth intensive digital imaging and multimedia devices. Conversely, the IEEE 802.15.4 standard for Low-Rate Wireless Personal Area Networks

Table 2.1: Frequency bands and data rates in IEEE 802.15.4-2003 as defined by ITU [25]

Band	868 MHz	915 MHz	2.4 GHz
Region	EU, Japan	US	Worldwide
Channels	1	10	16
Data rate	20 kbps	40 kbps	250 kbps

(LR-WPAN) targets at defining the physical and MAC layers of WPANS with low-power and low-cost devices, such as WSNs.

Requirements for mesh networking with WPANs are determined in the 802.15.5 task group. Short range and low frequency body area networks are being defined in the 802.15.6 task group. The 802.15.7 task group defines the physical and MAC layers for visible light communication WPANs.

2.4.1 IEEE 802.15.4 Standard

The IEEE 802.15.4 [44] standard enables LR-WPANs to be used in a wide range of application areas, such as industrial, medical and residential applications, while requiring little or no infrastructure. The main goals of IEEE 802.15.4 are to allow ease of installation, reliability of short range data transfers, low cost and long lifetime of devices. IEEE 802.15.4 uses the Industrial, Scientific and Medicine (ISM) frequency bands defined by the International Telecommunication Union (ITU), as shown in Table 2.1. The 2006 version of the standard also supports 250 kbps transfer speeds in the 868 MHz and 915 MHz bands.

A IEEE 802.15.4 compliant network has two types of network devices [25]. First, Full-Function Devices (FFDs) can act as Personal Area Network (PAN) coordinators or regular coordinators. Each network must contain a single PAN coordinator that is responsible for initializing the network. In addition, the PAN coordinator can act as a gateway node to other networks. Regular coordinators route traffic between other devices. Coordinators can also act as alternative PAN coordinators, which can assume the role of the PAN coordinator, if the original PAN coordinator leaves the network or malfunctions.

In addition to the FFDs, a IEEE 802.15.4 compliant network contains Reduced-Function Devices (RFDs). RFDs have very limited resources compared to FFDs and they can only communicate with one FFD at a time. RFDs are meant for small cheap devices, such as light switches or low-powered sensors.

Each device in an IEEE 802.15.4 network has a unique 64-bit address that is used when communicating with other devices on the data link layer. Each device may also be assigned a shorter 16-bit address by the PAN coordinator. In addition, each

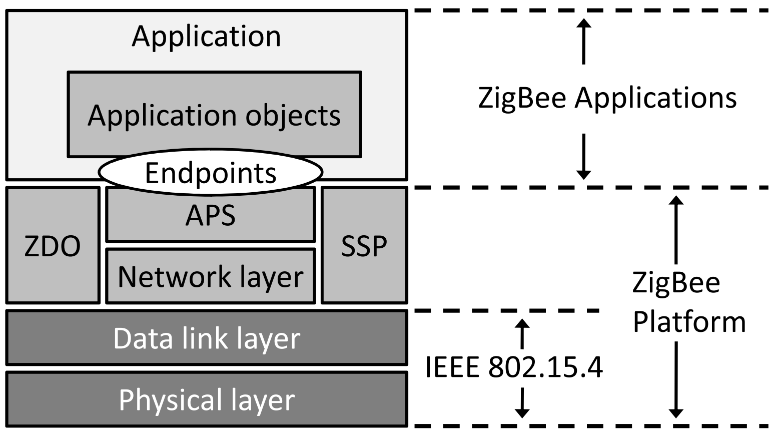


Figure 2.6: ZigBee protocol stack [25].

independent PAN also uses a unique PAN identifier, that can be used to identify different networks. The IEEE 802.15.4 standard does not define how the PAN identifier should be chosen.

2.4.2 ZigBee

As the IEEE 802.15.4 standard only defines the physical and MAC layers of a WSN stack, additional layers must be specified before a fully working WSN stack can be implemented. Zigbee [34] is an open specification based on the IEEE 802.15.4 standard that the defines the necessary WSN protocol layers that are not included in the IEEE 802.15.4 standard. The open and nonprofit ZigBee Alliance develops the ZigBee specification and also provides certification, marketing and user education [25]. ZigBee targets low-powered and low-cost applications, such as wireless monitoring and control in home automation, building control and industrial automation.

The protocol stack of a ZigBee node is shown in Figure 2.6. The physical and data link layer follow the IEEE 802.15.4 standard. The network layer of ZigBee supports the star, the mesh and the cluster tree topology. In ZigBee, FFDs and RFDs are called ZigBee routers and ZigBee end-devices, respectively. In the star topology all devices form a direct connection with the PAN coordinator, which is called the ZigBee coordinator in a ZigBee network. Thus, the network size is limited by the communication range of the ZigBee coordinator. On the other hand, the star topology allows for better control of latency.

In the multihop mesh topology any router may communicate with its neighboring routers. Thus, the coverage of the network can be extended by deploying more routers. Although this increases the redundancy in the network, it also increases the latency. In the cluster topology the routers act as cluster heads, while the leaf nodes are either other routers or end-devices. At the root of every cluster tree is a single coordinator, which initiates the network and chooses which routers are allowed to act as cluster heads.

Communication between ZigBee devices is conducted between endpoints. Each ZigBee device contains from 0 to 255 endpoints, which are connected to application objects. Endpoint 0 is connected to the ZigBee Device Object (ZDO), which is used to configure the whole device, and endpoint 255 is a broadcast endpoint, which is used to broadcast messages to all endpoints. The Security Service Provider (SSP) controls the security aspects of the protocol stack. ZigBee supports data encryption and uses 128-bit Advanced Encryption Standard for key generation. Finally, the Application Support layer (APS) connects together the network layer, the SSP and the application endpoints.

3. TUTWSN

TUTWSN has been developed at the DACI Research Group at Tampere University of Technology since 2002. The goals of TUTWSN development have not only been to further advance WSN research but also to create viable commercial products that utilize the state of the art technology. By the fall of 2009, almost 6000 nodes have been deployed in research WSNs [13].

This chapter presents the architecture of TUTWSN that consists of the TUTWSN network, the TUTWSN Gateway and the Sensor Network Application Platform (SNAP). Second, this chapter also illustrates the hardware platform of resource-constrained TUTWSN nodes. The protocol stack and the packet structure of TUTWSN are presented at the end of this chapter.

3.1 Architecture

The architecture TUTWSN can be divided in two sections, as shown in Figure 3.1. The WSN domain contains the actual physical WSN, which is implemented with TUTWSN technology. The information controller domain, on the other hand, is not tied to a particular WSN technology but can interact with multiple different WSNs from various manufacturers.

Nodes in a TUTWSN network can be divided into wireless nodes and wired gateway nodes. Unlike in the IEEE 802.15.4 WSNs, TUTWSN networks may contain multiple gateway nodes which increases the reliability of the WSN and distributes the traffic load to multiple gateway nodes. Gateway nodes form a connection with a TUTWSN Gateway using the TUTWSN Gateway Interface [19].

The TUTWSN Gateway periodically sends *data requests* to the WSN. These data requests define what sensor measurements should be performed and what is the interval between measurements. Most commonly used measurement intervals are between 30 seconds and 5 minutes. Each node that is equipped with the proper sensor and the corresponding device driver and application begins performing measurements at the requested interval and sending the results to the nearest gateway node. The gateway nodes forward these measurement results to the TUTWSN Gateway as GWI messages, which the TUTWSN Gateway translates into Java objects, which are then sent to the SNAP for storage.

Each TUTWSN Gateway can handle multiple WSNs. Large WSNs can generate

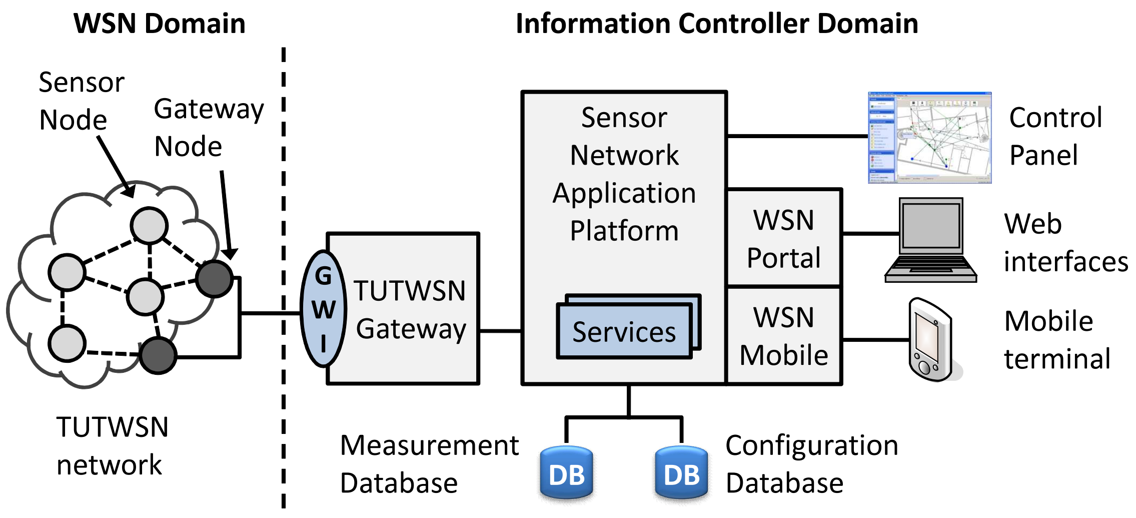


Figure 3.1: Infrastructure of TUTWSN. SNAP provides centralized management of multiple TUTWSN Gateways and WSNs.

so many GWI messages that such a WSN is assigned to a dedicated TUTWSN Gateway running on a separate server. WSNs that use different versions of the GWI interface must also be assigned to different Gateways.

As the number of nodes, WSNs and TUTWSN Gateways grows, the challenge of managing them becomes evident. SNAP provides centralized management for TUTWSN Gateways and WSNs by storing configuration and measurement information in a database and offering user interface for monitoring and controlling TUTWSN Gateways. In addition, SNAP provides service libraries for application developers in order to rapidly create feature rich applications for WSNs. WSN Mobile, WSN Portal and WSN Control Panel are existing TUTWSN applications that utilize these services. WSN Mobile is a lightweight WSN monitoring interface for mobile devices. WSN Portal is a framework for hosting web-based WSN applications. WSN Control Panel is a Java application for monitoring and controlling WSNs. WSN Mobile and WSN Portal are implemented with Java Server Pages.

3.2 Hardware Platforms

TUTWSN nodes and gateway nodes share a common hardware platform but the gateway nodes are equipped with an additional ethernet module. Both classes of devices either use the 2.4 GHz or the 433 MHz ISM bands. Each node contains a limited physical interface that consists of a single push button and two LEDs. Every physical TUTWSN node is identified with a globally unique serial number. TUTWSN nodes may contain a Dallas Semiconductors DS620 [36] digital temperature sensors, an Avago Technologies APDS-9002 [41] luminance sensor or a three-axis VTI SCA3000 accelerometer [47]. Additional external sensors may be attached with

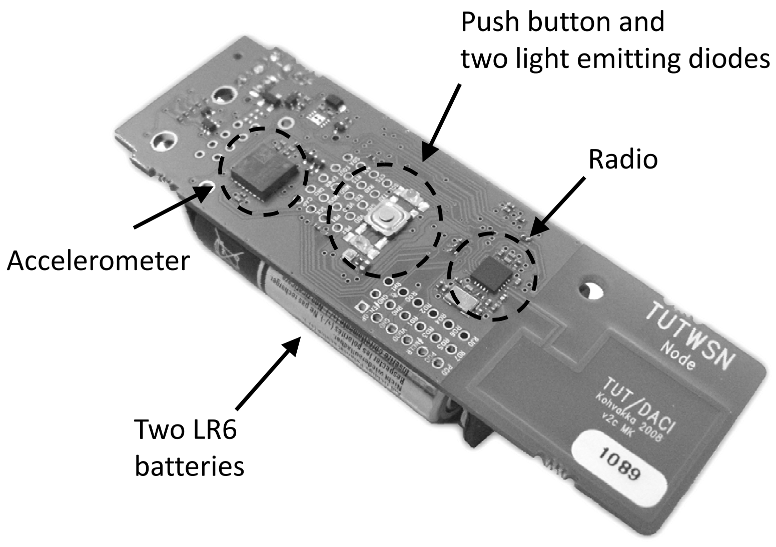


Figure 3.2: TUTWSN 2.4 GHz node.

a special connector. A battery powered 2.4 GHz TUTWSN node is shown in Figure 3.2.

TUTWSN nodes contain a Microchip PIC18LF8722 8-bit microcontroller [29] with 128 kilobytes of program memory and 3936 bytes of data memory. In addition, the microcontroller has a 1024 byte Electronically Erasable Programmable Read-Only Memory (EEPROM). The microcontroller operates at 4 MHz and has an execution speed of 2 million operations per second. Furthermore, TUTWSN gateway nodes use an additional PIC18F67J60 8-bit microcontroller [31] to execute the TCP/IP protocol stack in parallel to the TUTWSN protocol stack.

TUTWSN nodes use two types of radios depending on the operating frequency. The Nordic Semiconductors nRF24L01 [32] radio operates on the 2.4 GHz ISM band, which is divided into 126 discrete channels. The radio offers a transfer rate of 2 Mbps with a payload size of 32 bytes. In addition, the radio can use four different transmission powers, ranging from -18 dBm to 0 dBm. TUTWSN node platforms that use the nRF24L01 radio typically have a communication range of 20 to 30 meters.

The Nordic Semiconductors nRF905 [33] radio is used for the 433 MHz ISM band. The radio offers a transfer rate of 50 kbps and a payload size of 32 bytes. The transmission powers of the nRF905 range from -10 dBm to 10 dBm. TUTWSN node platforms that use the nRF905 radio can have a communication range up to one kilometer but terrain conditions usually limit the range to a few hundred meters. Neither radio supports RSSI.

The Texas Instruments CC1101 [42] radio is another choice for the 433 MHz band TUTWSN platforms. The CC1101 offers a transfer rate of 50 kbps and a variable

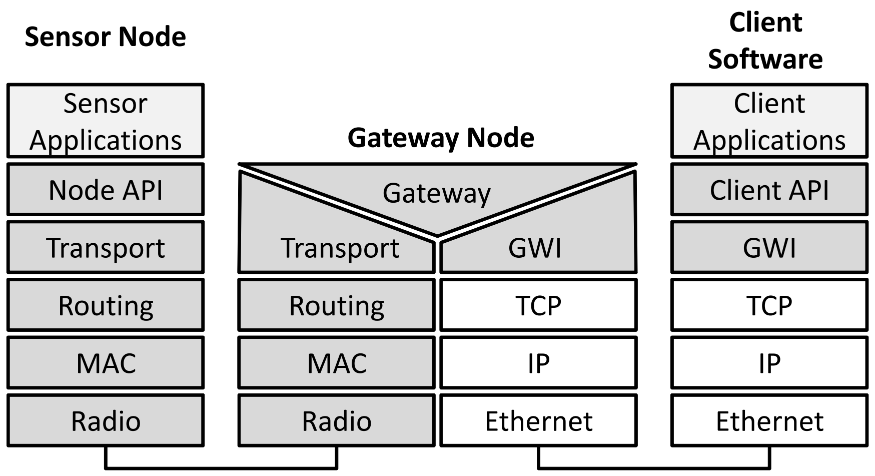


Figure 3.3: The TUTWSN protocol stack of a node, a gateway node and a client application.

payload size between 1 and 255 bytes. The transmission powers of the CC1101 range from -30 dBm to 10 dBm. The CC1101 supports both RSSI and a link quality indicator, that estimates how easily the received signal can be demodulated.

3.3 Protocol Stack

The protocol stacks of a TUTWSN node, a TUTWSN gateway node and a client application are shown in Figure 3.3. Applications running on a node use an event-based Node API [19] to transmit messages over the network. In addition, the Node API makes sure that the applications do not conflict with the strict timing requirements of the MAC layer. Gateway nodes receive Node API messages and transmit them to client applications using a Gateway Interface (GWI) protocol, the Internet Protocol (IP) and the Transmission Control Protocol (TCP).

The routing layer of TUTWSN uses a QoS aware routing protocol [40]. As TUTWSN is designed to work with multiple applications with different QoS requirements, such as latency, reliability and energy usage, the routing protocol defines multiple *traffic classes*. Each application can choose which traffic class to use. Headnodes calculate the routing cost for each sink and each traffic class, which are then advertised to the cluster members.

The MAC layer of TUTWSN uses a cluster-tree network topology to maximize scalability and minimize energy usage. Similarly to the FFDs and RFDs in the IEEE 802.15.4 standard, node roles are divided into *headnodes* and *subnodes*. Headnodes perform routing and act as cluster heads or cluster members. Subnodes do not route and can only act as cluster members. Gateway nodes act as sinks for the data streams and always act as cluster heads. Each node is assigned with a 24-bit *node ID* for addressing individual nodes on the MAC layer. The node ID is not

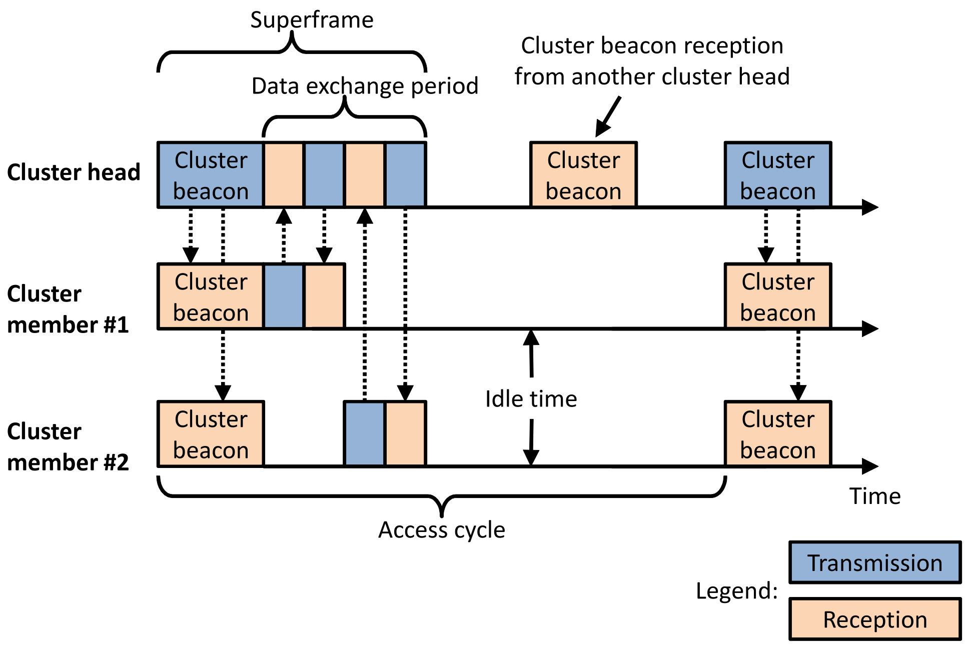


Figure 3.4: Communication between nodes in a cluster.

necessary unique, as nodes in different WSN may use the same node ID. The node ID is also used on the application layer to identify nodes and to act as a key to map measurement data in a database to a particular node.

TUTWSN significantly reduces the energy usage of the data link layer and the radio by using a synchronized Frequency and Time Division Multiple Access (FTDMA) MAC [25, Chapter 13.], where the frequency band is divided into discrete clsuter channels and timeslots. These timeslots are called *access cycles*, as shown in Figure 3.4. Each access cycle begins with a superframe, which is used by the headnode to communicate with its cluster members. In the beginning of the superframe the headnode broadcasts a cluster beacon to the cluster members. Cluster members need the data in the beacon to identify which slots are allocated to them during the data exchange period. After sending the cluster beacon, the headnode communicates with the cluster members one by one. The superframe is followed by an idle period to preserve energy. While the cluster members are idle, the cluster head may need to communicate with other clusters to forward the data to them.

The FTDMA schedule of a headnode, which contains the cluster channel and the start time of the access cycle, is advertised periodically with network beacons on a preselected *network channel*. Thus, headnodes and subnodes perform neighbor discovery of nearby clusters by listening to the advertisements on the network channel, as shown in Figure 3.5. In addition, the accurate timing information allows the cluster heads and members to wake up from sleep precisely at the right moment, thus maximizing the time spent in power-saving sleep modes.

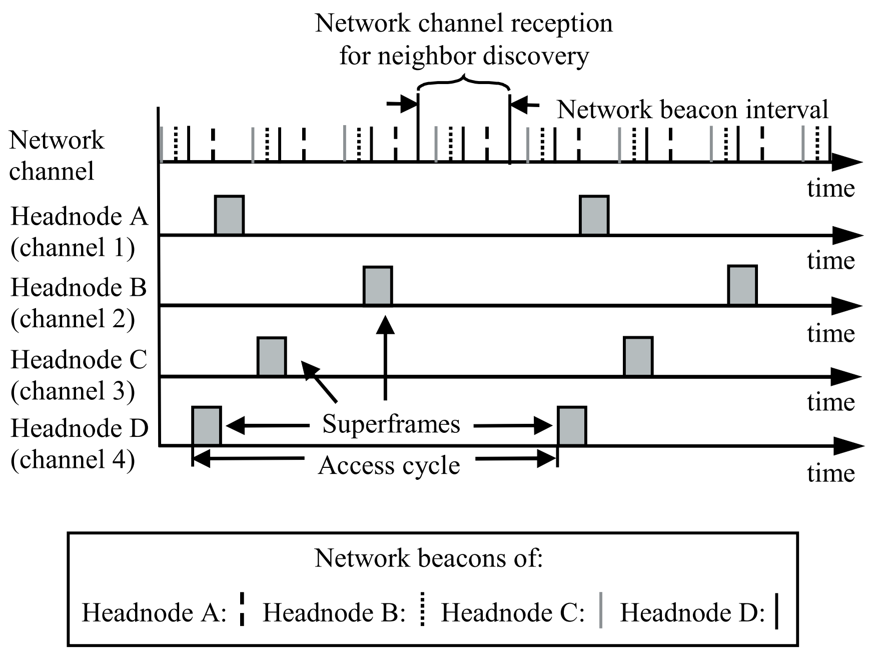


Figure 3.5: Neighbor discovery in TUTWSN [25].

Communication between nodes happens in small packets called Protocol Data Units (PDUs). Each PDU is 27 bytes long and may contain fields from different protocol layers. The structure of a PDU from the TUTWSN temperature application is shown in Figure 3.6. The first fields of the PDU belong to the MAC layer. The header describes the PDU type and also contains information about the transmission power that was used to send this packet. The source and the destination fields contain the node ID of the sender and the receiver. The type field describes the type of the packet contained in the MAC payload while the sequence field contains the sequence number of the packet. On the Node API layer, the application field identifies which application created the packet. The target field contains additional

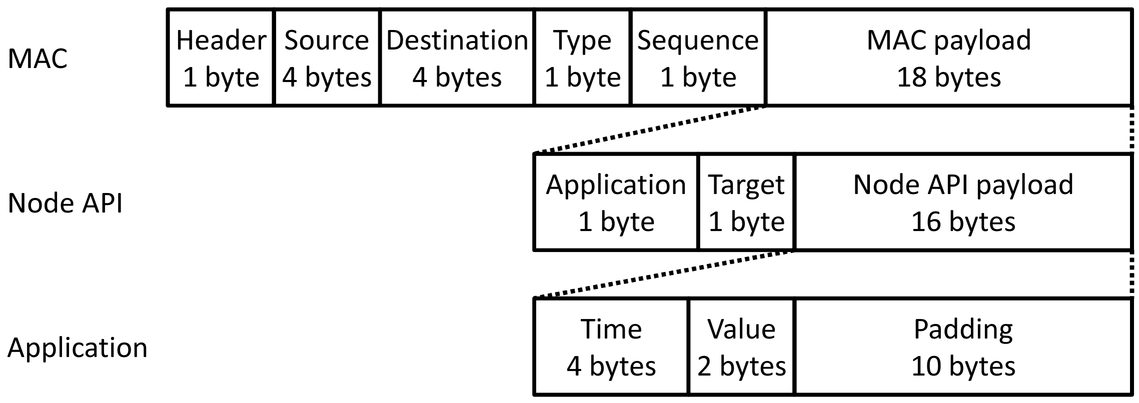


Figure 3.6: A measurement PDU from the temperature application.

information, for example if the packet was targeted at applications in nodes that belong to a particular group. The last 16 bytes are left for the application. The time field stores the time stamp when the measurement was made while the value field contains the actual measurement value. Rest of the PDU is marked as padding. It is important to note, that as the number of protocol layers increases, the number of bytes left for higher level applications can decrease dramatically.

The physical layer of the TUTWSN protocol stack is implemented with the nRF24L01 and nRF905 radios. Both radios use a 3 to 5 byte long *radio address* to control communication as different radio devices can only communicate with each other if they use the same radio address. In TUTWSN, each WSN is assigned with a unique radio address. This separates different WSNs, so that nodes from one WSN cannot communicate with nodes in another WSN.

The TUTWSN protocol stack is implemented using the C programming language and the MPLAB C compiler [30] is used to compile the protocol stack to a runnable firmware image.

4. REQUIREMENTS FOR FIRMWARE MANAGEMENT

In order to understand the requirements for firmware management, the entire lifespan of a TUTWSN node is explored. First, a summary of the requirements is presented. Second, the motivation behind firmware management is explained through TUTWSN deployments. Third, the most significant parameters associated with the TUTWSN protocol stack are identified. Fourth, TUTWSN nodes and WSNs are followed from platform manufacturing to deployment and all the way to disposal and redeployment, while specifying what requirements each phase has on firmware management and how it can be used.

The following is a short summary of the requirements for firmware management:

Moving nodes between WSNs: Nodes must be easily moved from one WSN to another.

Replacing nodes: The parameters of a defective node must be easily transferred to replacement node.

Adding or removing sensors: It must be possible to add external sensors or actuators to nodes or to remove them.

Manual programming: Manually programming nodes is expensive and must be avoided.

Persistent storage: Firmware images and their parameters must be stored so that they may be later retrieved or modified.

Configuring modules: The parameters and options of individual applications, device drivers or libraries must be configurable.

Firmware image changes: Firmware images in deployed nodes must be remotely and reliably updatable.

Testing of nodes: Firmware images in nodes should support self testing.

Table 4.1: TUTWSN deployments.

Monitoring type	Number of nodes	Frequency band	Coverage	Location
Teaching	280	2.4 GHz	University campus	Tampere, Finland
Real estate	130	2.4 GHz	Large college	Tornio, Finland
Person tracking	110	2.4 GHz	Hospital	Kainuu, Finland
Person tracking	100	2.4 GHz	Police Academy	Tampere, Finland
Outdoor	60	433 MHz	Fields, 6 km^2	Kangasala, Finland
Industry	30	2.4 GHz	Green house	Turku, Finland
Domestic	30	2.4 GHz	A home	Tampere, Finland
Industry	25	433 MHz	Pipe line, 5 km	Voikkaa, Finland
Total	*765*			

4.1 TUTWSN Deployments

The main motivation behind developing firmware management has come from experiences in deploying TUTWSN networks, as shown in Table 4.1. The teaching network and the real estate network are example of large scale, 2.4 GHz TUTWSN monitoring networks for indoor use. Large TUTWSN networks have also been used for tracking nurses in a hospital and police students in a training area. Outdoor 433 MHz networks have been used to cover large areas. The scale of TUTWSN installations justifies the need for firmware management.

4.2 Firmware parameters in TUTWSN

The protocol stack of a TUTWSN node is controlled by seven major firmware parameters, as shown in Table 4.2. The *Node API version* defines the format of application layer packets, which must be the same within a WSN. The Node API version must also be compatible with the TUTWSN Gateway and, thus, multiple WSNs typically share the same Node API version.

Each WSN is identified with a unique combination of a *radio address* and a *network channel.* These two parameters allow nodes to communicate with other nodes that belong to the the same network. As TUTWSN supports multiple concurrent services and applications with different QoS requirements, the routing layer offers them different routing classes. The *routing cost* controls how data belonging to a specific traffic class is routed through the WSN.

The *node ID* is used to identify a single node within a specific WSN. On the other hand, multiple nodes in separate WSNs may use the same node ID. The *node role* parameter controls whether a node operates as a headnode or as a subnode. In addition, the node role can be set to *automatic* whereby the node will autonomously

Table 4.2: Node firmware parameters in TUTWSN.

Parameter	TUTWSN Layer	Scope	Size (bits)
Node API version	Application	Global	8
Radio address	Physical	WSN	24
Network channel	MAC	WSN	8
Routing cost	Routing	WSN	96
Node ID	MAC	Node	48
Node role	MAC	Node	2
Enabled applications	Application	Node	64

switch between headnode and subnode roles when needed.

On the application layer, a TUTWSN node supports multiple concurrent applications that control different sensors. These applications can be enabled and disabled depending on which physical sensors are attached to a particular node.

TUTWSN gateway nodes share the same parameters as nodes, but also include additional parameters related to their role as gateways, as shown in Table 4.3. As each gateway node has a physical ethernet port, it must be assigned with a unique ethernet MAC address. Each gateway node must also know the IP address and the TCP port of the destination TUTWSN Gateway. Furthermore, each gateway node implements a particular version of the GWI, which must be compatible with the TUTWSN Gateway that it is being connected to. As each TUTWSN Gateway handles multiple WSNs, the GWI version must be the same in all those WSNs.

4.3 Node Lifetime

In this section the lifetime of a TUTWSN node is analyzed. In addition, special care is taken to recognize the tasks that could be automated with firmware management to minimize the amount of manual work and reduce delays in the process. To clarify the different tasks and responsibilities associated with WSN deployments, four different roles are defined, as shown in Figure 4.1: the electronics contract

Table 4.3: Additional firmware parameters for gateway nodes.

Parameter	Protocol	Scope
MAC address	Ethernet	Node
Gateway destination address	IP	WSN
Gateway destination port	TCP	WSN
GWI version	TUTWSN Gateway	Global

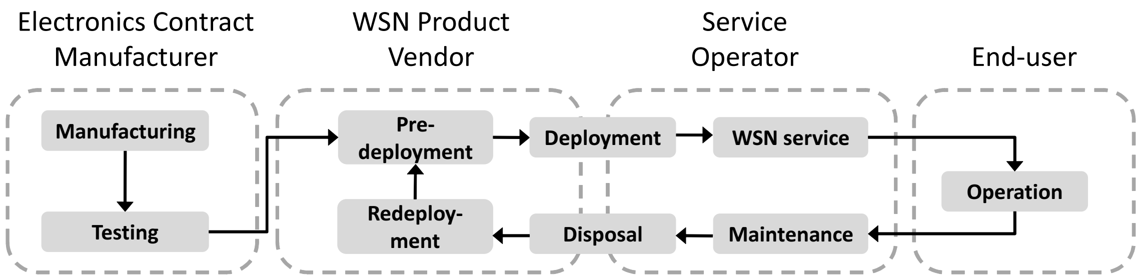


Figure 4.1: Lifetime of a TUTWSN node.

manufacturer, the WSN product *vendor*, the service *operator* and the *end-user*. The manufacturer is responsible for producing nodes. The vendor creates WSN products that the operator uses to create application-specific WSNs and services in accordance to the requirements given by the end-user, who uses the WSN and the services.

4.3.1 Manufacturing and Testing of Nodes

Before the manufacturing of nodes begins, the vendor places an order to the manufacturer, as shown in Figure 4.2. The manufacturer is also supplied with a list of available serial numbers for the nodes and the necessary design documents. The vendor stores the serial number and platform type of each ordered node in a database. Thus, each TUTWSN node is tracked before it is manufactured.

After the manufacturer has received the order, the hardware components and Printed Circuit Boards (PCBs) are ordered from component suppliers and the nodes are assembled. In addition, physical sensors are attached to node and each node is enclosed in a chassis. Each node is also marked with a serial number for identification. The finished nodes are then shipped to the operator. Firmware management has three challenges during this phase.

First, the manufactured TUTWSN nodes must be programmed before shipping them to the operator, so that they may immediately be used in deployments. As manually programming a single node takes several minutes, it is impractical and expensive to do so for several hundreds of nodes. Instead, the microcontrollers that are used in the nodes must be pre-programmed with a TUTWSN firmware image by the microcontroller manufacturer. This firmware image should contain a general purpose TUTWSN protocol stack, so that the pre-programmed nodes may be used in WSNs immediately.

Second, the firmware image must support wireless runtime configuration of the firmware parameters and update of the firmware itself, so that the pre-programmed nodes may be updated and configured later by the vendor and the operator. If the target WSN is already installed and operational, the vendor and the operator must

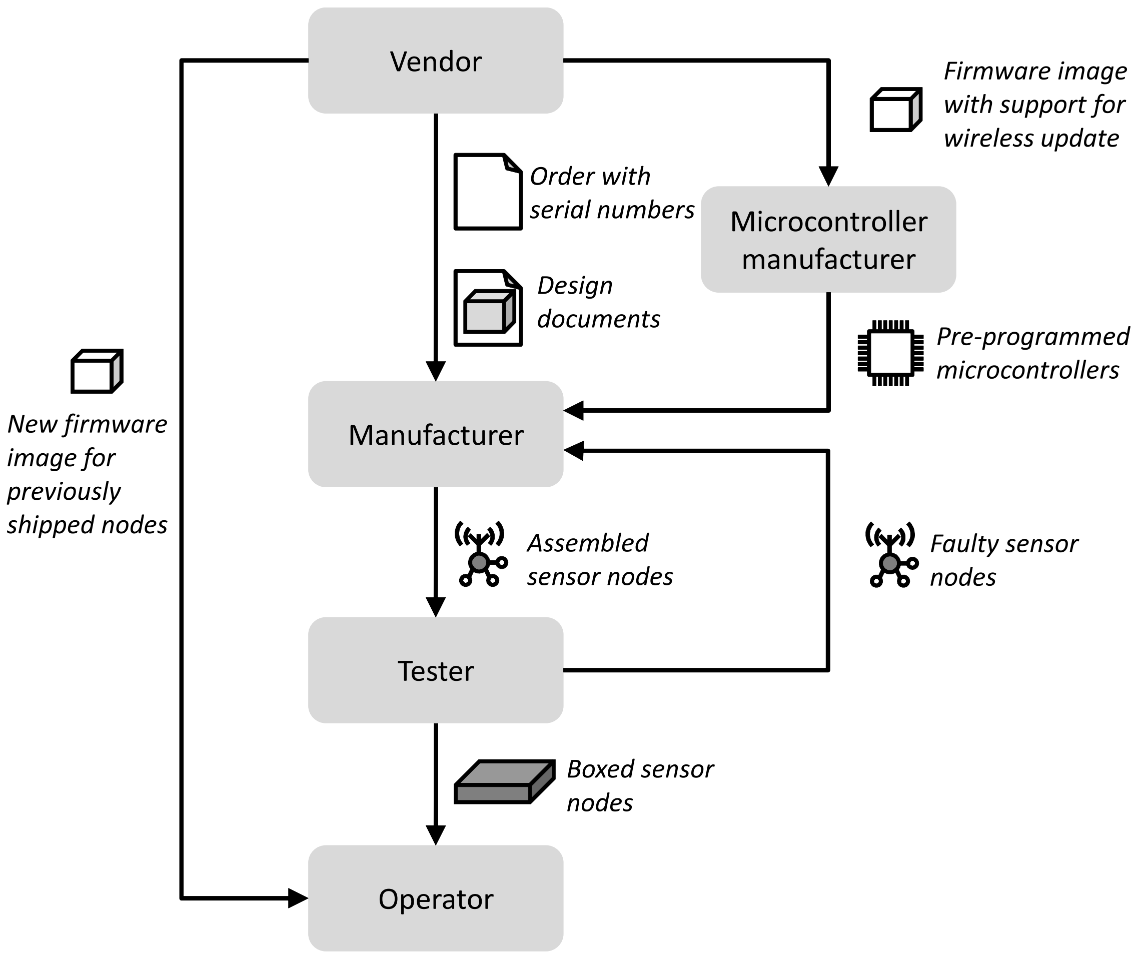

Vendor
Firmware image with support for wireless update
Order with serial numbers
Microcontroller manufacturer
Design documents
Pre-programmed microcontrollers
New firmware image for previously shipped nodes
Manufacturer
Assembled sensor nodes
Faulty sensor nodes
Tester
Boxed sensor nodes
Operator

Figure 4.2: Manufacturing of TUTWSN nodes with pre-programmed microcontrollers. Shipped nodes can be later updated wirelessly.

be able to remotely inject a new firmware image into the WSN so that the new firmware image will propagate through the rest of the WSN without manual work. This requires that nodes must have a protocol for sharing information about their own firmware image and transferring a newer image to a neighboring node.

Third, the pre-programmed firmware image must support testing of the hardware platform for physical defects. The hardware designs of different TUTWSN platform types are updated regularly to meet new customer requirements or to reduce manufacturing costs and energy usage. These updates often require making small changes to the PCB schematics or even replacing complete parts, which requires that the assembly parameters, such as the amount of solder, are adjusted accordingly. On the other hand, these adjustments of the manufacturing process also increase the chances of physical defects, as the proper assembly parameters are not always known a priori. Thus, a reliable testing process is required to allow the manufacturer to discover problems in the manufacturing process before they can affect a large number of platforms. The pre-programmed firmware image must contain a self-test application, that allows the manufacturer to validate that the platform is operating correctly. This self-test application can be wirelessly configured by the manufacturer to account for possible external sensors, without requiring manual programming or different firmware images for each platform type.

4.3.2 Pre-Deployment Phase

Before a WSN can be created and deployed, the operator and the end-user specify the application requirements for the WSN, such as the target lifetime of the WSN and the size of the monitored area. On the other hand, the operator may have several predefined WSN products that fit typical customer requirements, such as small WSNs for home monitoring or large WSNs for outdoor environmental monitoring. In either case, once the application requirements have been specified, the operator orders the WSN components from the vendor and begins building the WSN by configuring the necessary nodes.

As TUTWSN nodes are pre-programmed with a general purpose TUTWSN protocol stack before the operator receives them, they can be immediately used in WSNs. If the standard TUTWSN protocol stack is not applicable with regard to the application requirements given by the operator and the end-user, the vendor must adjust the protocol stack accordingly. An end-user may request that a particular node application transmits measurement packets in a specific format. This requires that the corresponding application and the packet structure are changed in all of the nodes of the WSN and that the firmware images of the nodes are recompiled. The compilation process must produce a firmware image that can be easily stored in a database. After the compilation, the vendor must store the customized firmware images in a database, so that they may be retrieved later by the operator.

Once all of the nodes have the correct firmware the operator needs to update the firmware parameters in the database by giving each node the necessary parameters, such as the node ID and the network information. Once the firmware parameters in the database are updated, the operator configures the nodes to use the updated parameters by wirelessly transferring the parameters to the nodes. In addition, gateway nodes are configured by giving them Ethernet MAC addresses and the connection parameters of the operators TUTWSN Gateway that handles the WSN.

4.3.3 Deployment, Service and Operation Phase

Once a WSN has been created and the nodes have been configured, the operator or the end-user deploys the WSN around a measurement area. The operator sells the end-user WSN services that gather and present information from the WSN to the end-user. At the same time, the vendor may gather diagnostic information on the WSN that can be used to diagnose faulty nodes or problems in the communication. As the length of the operation phase can be several months or even years, it is not uncommon that the requirements for the WSN change during this time.

First, new nodes may be added to the WSN if the network coverage, node density or level of redundancy is to be increased. End-users may also request that new sensors are added to the existing nodes in order to add, for instance, security monitoring capabilities to an environmental monitoring WSN. As the firmware images and parameters for each WSN are stored in a database, the operator can easily locate the correct firmware images and transfer them into the new nodes.

In addition, the operator needs to change the firmware parameters of the nodes to match the network parameters of the WSN and the new sensor and then transfer these updated parameters to the nodes. As the nodes are already deployed, the operator must transfer the updated parameters to the nodes by using the WSN itself as the communication medium.

Second, in addition to changes in the physical features of a WSN, such as the number of nodes, the developer and operator may want create new software features, new node applications and improved protocols and offer them using existing WSNs, which requires updating them. The long lifetime of a WSN makes it highly probable that firmware images are updated several times while the network is still deployed. Moreover, as WSNs can be deployed almost everywhere, physical access can be expensive or impossible.

4.3.4 Maintenance, Disposal and Redeployment

During the deployment, nodes may be damaged or break down. In these situations, the end-user sends these damaged nodes back to the operator. The operator may then ask the vendor for a replacement node. When replacing a malfunctioning node, the replacement node is added to the WSN as a normal node with the exception that the replacement node is assigned with the same node ID as the malfunctioning node. Thus, sharing the node ID allows the new replacement node to generate measurements that are automatically combined with the earlier measurement data of the malfunctioning node.

The final phase of a WSN deployment is the disposal of the WSN, where the deployed nodes are gathered and shipped back to the operator. During long deployments it is typical that some of the nodes are damaged. During disposal, these broken nodes are either shipped to back to the manufacturer for repair, or discarded. The rest of the nodes are reused in later WSN deployments, once their firmware images have been updated and their firmware configurations are cleared.

5. RELATED TECHNOLOGIES

In this chapter the existing proposals for managing both wired and wireless heterogenous networks are discussed and analyzed based on how well they are suitable for WSNs. First, the IEEE 1451 standard is included as it contains methods for storing and discovering the parameters of sensor equipment. Second, the Simple Network Management Protocol (SNMP) is presented as an example of a parameter management architecture for wired networks. Third, TinyOS and Contiki are presented as state of the art WSN operating systems and their solutions for handling firmware management tasks are discussed.

5.1 IEEE 1451 Smart Transducer Interface Standard

Sensors and actuators from different manufacturers have a wide range of parameters that define how the measured data is to be interpreted: calibration data, value and error ranges, measurement units and scales. Before a sensor can be measured by a controlling device, the device must know these parameters. The IEEE 1451 standard family [43] defines how sensors and actuators from different manufacturers can exchange this sensor information with measurement devices, such as nodes in a WSN [25] to enable interoperability. The IEEE 1451 standard defines two device classes: Smart Transducer Interface Modules (STIMs) and Network Capable Application Processors (NCAPs). STIMs are sensing devices that contain the necessary components to perform measurements and transfer the result to an NCAP device. Each STIM contains a Transducer Electronic Data Sheet (TEDS), which describes the sensor parameters of the STIM device in detail and is defined in the IEEE 1451.0 standard. The TEDS is usually implemented as an EEPROM memory module. As opposed to STIMs, NCAPs are network connected devices that receive measurement data from multiple STIMs. Accessing an NCAP device from an external network is done according to the IEEE 1451.1 standard.

IEEE 1451 does not define how NCAPs and STIMs communicate with each other on the physical or the data link layer, as shown in Figure 5.1. Instead, the standard family supports interface definitions for several commonly used physical and data link layers. First, the IEEE 1451.2 standard defines an interface and a TEDS for a point-to-point wired link between an NCAP and a STIM using a Serial Peripheral Interface bus. This standard is currently being revised to add support for Universal

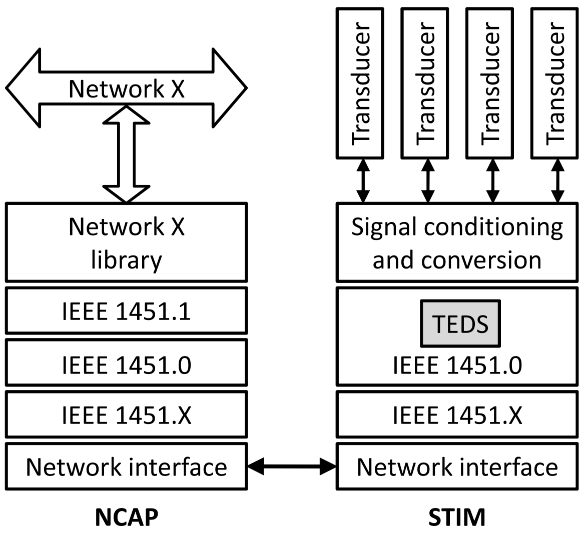


Figure 5.1: The interfaces between STIM and NCAP devices as defined in the IEEE 1451 standard [25, Figure 3.1].

Asynchronous Receiver / Transmitter and Universal Serial Bus busses. Second, the IEEE 1451.3 standard defines an interface for multi-drop STIMs using a wired Home Phone Network Alliance connection to connect multiple STIMs to a single NCAP. Communication on a mixed-mode bus with both analog and digital modes is defined in the IEEE 1451.4 standard. The IEEE 1451.5 standard defines communication over wireless links using other IEEE standardized mediums: IEEE 802.11 compatible WLAN, 802.15.1 compatible Bluetooth and 802.15.4 compatible ZigBee. Finally, the IEEE 1451.6 standard defines interfaces for a Controller Area Network bus to connect an NCAP and multiple STIMs.

Although the TEDS in the IEEE 1451.5 standard is designed to store a variety of parameters, it is not suitable for firmware management. The standard implies that each ZigBee RFD and FFD will act as a STIM and contain a single TEDS, while a single ZigBee coordinator will act as an NCAP. On the other hand, this requires that the TEDS in each node should contain all the possible parameters of each available external sensor, so that the node can update its TEDS at runtime to match the currently equipped sensors. However, storing TEDS parameters for every possible sensor type would require a significant amount of memory and, thus, this solution would not be scalable.

5.2 Simple Network Management Protocol

Managing networked devices has been a challenge long before WSNs. SNMP [46] is a protocol for managing TCP/IP-based networked devices and it belongs to the Internet Protocol Suite. SNMP defines two groups of devices. First, *network elements* are devices such as routers, terminals or desktop computers that are remotely managed. Each network element contains an agent, which is a software component that stores management information about the network element, such as the current operating temperature or memory usage. This management information is stored as variables in hierarchical structures known as Management Information Bases (MIBs). The second group of devices are *network management stations*. These management stations manage network elements by using a software component called the network management system to connect to the agents running inside the network elements and performing queries on the MIBs.

When the relationship between network elements and network management stations is analyzed, it can be seen that the intelligence of the system exists within the network management stations that perform queries, while the network elements act as database nodes. This structure is not suitable for WSNs [37] as the demand for data transfer between MIBs in network elements and network management stations is too high. Instead, the proposed solution in [37] is to execute the management functionality within each node in order to minimize the amount of data transferred.

5.3 Support for Firmware Management in WSN Operating Systems

Current methods for transferring and reconfiguring firmware images in WSNs can be divided into four groups according to their level of flexibility and energy costs, as discussed extensively in [3]. First, *full image binary upgrades* allow the highest degree of flexibility in configuring the firmware, as the whole firmware image is reprogrammed at once. Conversely, the energy and storage cost of the update procedure is also significant, as any change in the firmware image requires that the whole image is disseminated. *Modular binary upgrades* allow segments of the firmware image, such as libraries or applications, to be updated individually, thus forming a compromise between the flexibility of the upgrade and the energy costs. However, this method requires that nodes are able to relocate and link these modules dynamically during runtime, which can be computationally intensive. Higher level functionality can be configured with *virtual machines*, where applications are described as scripts and interpreted during runtime. While transferring scripts rather than full binary images requires less energy, the energy cost of the runtime interpretation can outweigh the benefits. Finally, *parameter tuning frameworks* allow fine tuning the firmware images without requiring much energy. Conversely, they offer the least amount of flexibility to configure the firmware images.

5.3.1 TinyOS

TinyOS [14] is a free open-source operating system for WSNs. TinyOS is written in nesC [10], which is based on the C programming language but optimized for resource-constrained devices. TinyOS supports concurrency through callbacks and by splitting long operations into shorter event-driven tasks. This enables TinyOS to have a high level of concurrency, but also increases program complexity. As TinyOS is both free and open-source, it is a popular choice for WSN research and a large number of reprogramming protocols exist for WSN that are built upon TinyOS. TinyOS code is statically compiled with nesC and it does not support loadable modules. Thus, the TinyOS firmware image must be transferred and loaded as a full binary image.

XNP [51] was one of the first reprogramming services for TinyOS and the MICA2 [5] platform. It featured a single-hop reprogramming scheme where the firmware image was sent as unicast to a particular node or broadcasted to a group of nodes. The single-hop nature limited the scalability of XNP and it only served as an alternative to manual wired reprogramming.

The successor to XNP is Deluge [16]. Deluge is an epidemic multihop protocol that allows nodes to store several different firmware images in an external EEPROM memory. One of these images can act as the so called Golden image, which is used as a backup image if the main firmware image is corrupted. The 2.0 version [15] also adds support for resuming incomplete firmware image downloads and additional firmware image verification.

The Maté virtual machine [26], which is built upon TinyOS, bypasses the lack of loadable modules in TinyOS by presenting a high-level virtual machine instruction set. Maté bytecode programs are significantly smaller than complete firmware images, which lowers the energy cost of disseminating them. The downside is that interpreting the bytecode creates an energy overhead. If new firmware images are disseminated only seldom, then the energy consumption of the code interpretation is dominant.

5.3.2 Contiki

Similar to TinyOS, Contiki [6] is a free open-source operating system for resource-constrained devices. Although Contiki is event-driven, it avoids the increased complexity by providing *protothreads* [8] over the event-driven kernel, which allows developers to avoid the excessive use of callbacks. Unlike TinyOS, Contiki supports modular updates [7] to the firmware image. Like other modular solutions, this saves energy as only parts of the whole firmware image need to be disseminated. Contiki uses a virtual filesystem to load executable files, which are dynamically linked and

Table 5.1: Comparison of related technology.

Technology	Operating system independent	Update of deployed nodes	Configuring deployed nodes	Requires additional resources
IEEE 1451	yes	no	yes	data memory for TEDS
SNMP	yes	no	yes	data memory for MIBs
XNP	no	no	no	program memory for images
Deluge	no	yes	no	program memory for image
Maté	no	partly	no	bytecode interpretation
Contiki	no	partly	no	-

relocated. This dynamic loading only applies to the higher level modules, such as libraries and applications. The core of Contiki, which contains the operating system, the program loader and the device drivers, can only be updated with a separate special image transfer program, which is a major restriction.

5.3.3 Conclusion

Table 5.1 contains a comparison of the presented technologies. IEEE 1451 and SNMP require too much data memory for TEDS and MIBs, respectively. The reprogramming protocols are either limited to a particular operating system, like Contiki, or require additional program memory for storing firmware images. In adition, Contiki and Maté can only update parts of the firmware image.

6. DESIGN OF FIRMWARE MANAGEMENT

This chapter presents the design of six firmware management components that have been created for TUTWSN. First, an overview of the components is presented. Second, a prototype of the user interface is presented. The structure of the firmware management database is described. Finally, the dissemination of firmware images and parameters into WSNs is explored.

6.1 Architecture

Firmware management components are shown in Figure 6.1. The server side components consist of a firmware management user interface, the Autoconfigurator service and the firmware management database. The WSN developer uses the firmware management user interface to add, remove and modify both firmware images and firmware parameters of nodes.

The Autoconfigurator is a service running on SNAP that receives the firmware images and parameters from the user interface and stores them in the firmware database, which is a part of the configuration database of SNAP. The Autoconfigurator also makes sure that new firmware images and parameter changes are transferred to the correct WSNs and nodes through the TUTWSN Network Gateway and the gateway nodes.

The node side components are the bootloader, the firmware image transfer protocol and the parameter transfer protocol. The bootloader manages the firmware image. If the bootloader detects corruption in the firmware image, it invokes the firmware transfer protocol. The firmware transfer protocol uses software advertisements to exchange firmware image information between nodes. The firmware transfer protocol transfers full binary images between nodes. The parameter transfer protocol is used by the Autoconfigurator service to transfer firmware parameters to individual nodes.

Individual firmware parameters are combined into a parameter stream, which is processed by a parameter handler interface in the receiving node. The single-hop version of the parameter transfer protocol is used for nodes that have not received their parameters at all. The multihop version of the parameter transfer protocol is used to transfer updated parameters to nodes that already are a part of a deployed WSN.

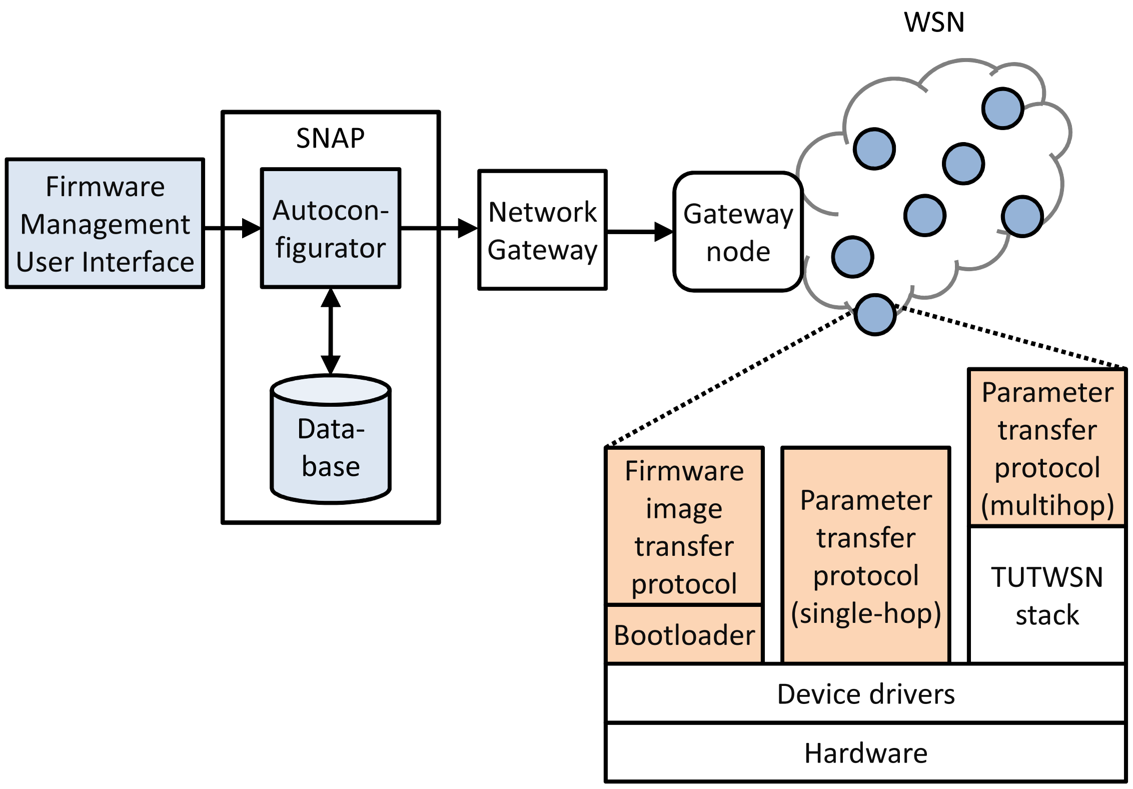

WSN
SNAP
Firmware Management User Interface
Autocon-figurator
Data-base
Network Gateway
Gateway node
Firmware image transfer protocol
Bootloader
Parameter transfer protocol (single-hop)
Parameter transfer protocol (multihop)
TUTWSN stack
Device drivers
Hardware

Figure 6.1: Overview of firmware management in TUTWSN. Server side components are highlighted in blue while the node side components are highlighted in orange.

6.2 Firmware Management User Interface

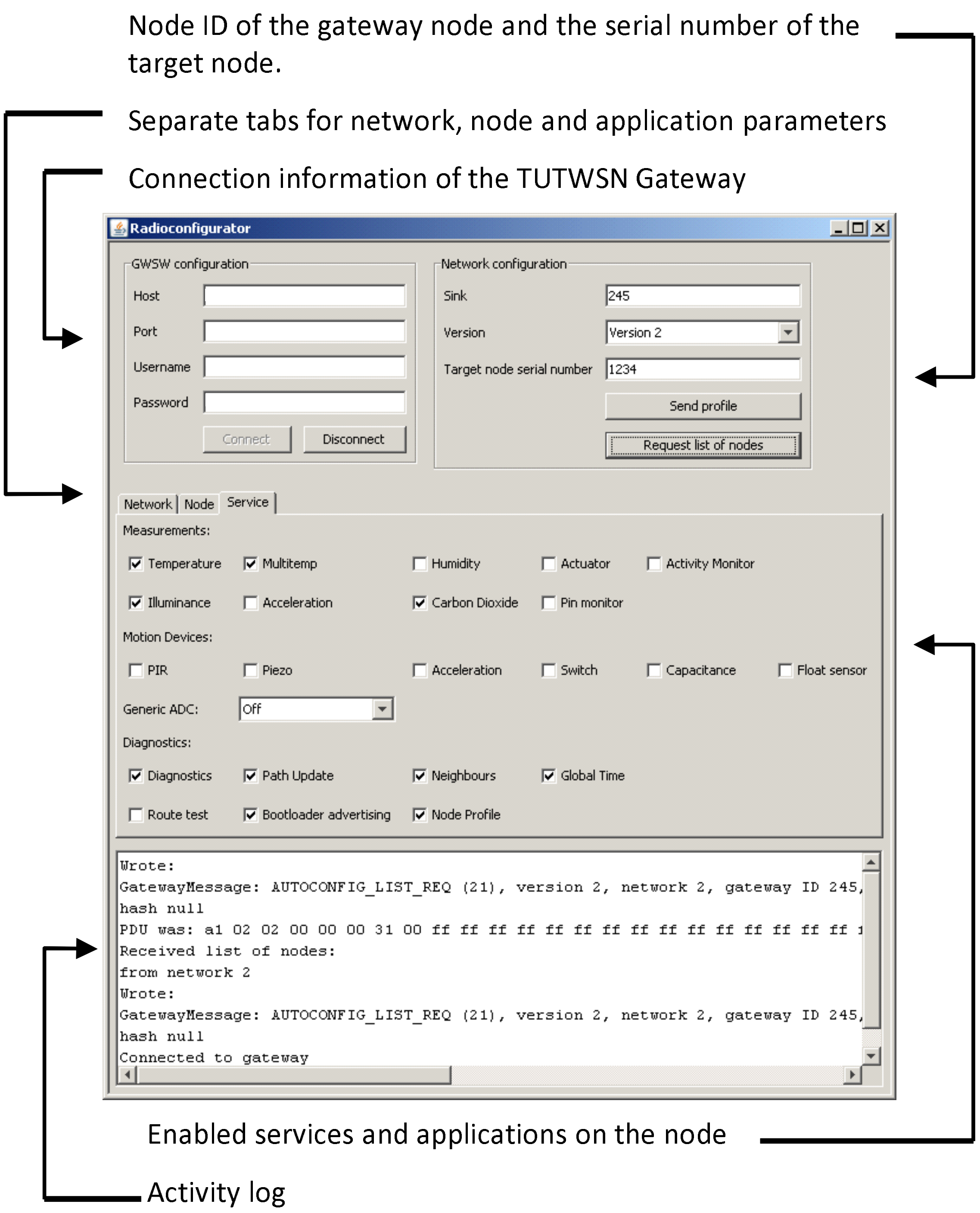
The firmware management user interface is used to add, remove and modify the firmware images and firmware parameters.

A prototype of the firmware management user interface was created in Java and allows a WSN operator to specify the firmware parameters, as shown in Figure 6.2. The created parameters are transferred to the target node through a TUTWSN Gateway and a gateway node using the single-hop parameter transfer protocol. This approach bypasses the firmware management database and the Autoconfigurator service. A WSN operator may also use the prototype user interface to detect uninitialized nodes.

6.3 Firmware Management Database

SNAP stores configuration information about nodes, WSNs and users as objects in a configuration database. The structure of the configuration database makes it easy to store firmware parameters in the existing objects. The primary classes of the configuration database are shown in Figure 6.3.

In the configuration database, the *Network* object contains WSN-specific parameters, such as the network channel of the TUTWSN MAC layer. Each network


Node ID of the gateway node and the serial number of the target node.
Separate tabs for network, node and application parameters
Connection information of the TUTWSN Gateway
Radioconfigurator
Enabled services and applications on the node
Activity log

GWSW configuration

Host:
Port:
Username:
Password:
Connect | Disconnect

Network configuration

Sink: 245
Version: Version 2
Target node serial number: 1234
Send profile | Request list of nodes

Network | Node | Service

Measurements:

[✓ Temperature] [✓ Multitemp] Humidity Actuator Activity Monitor
[✓ Illuminance] Acceleration [✓ Carbon Dioxide] Pin monitor

Motion Devices:

PIR Piezo Acceleration Switch Capacitance Float sensor

Generic ADC: Off

Diagnostics:

[✓ Diagnostics] [✓ Path Update] [✓ Neighbours] [✓ Global Time]
Route test [✓ Bootloader advertising] [✓ Node Profile]

```
Wrote:
GatewayMessage: AUTOCONFIG_LIST_REQ (21), version 2, network 2, gateway ID 245,
hash null
PDU was: a1 02 02 00 00 00 31 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 1
Received list of nodes:
from network 2
Wrote:
GatewayMessage: AUTOCONFIG_LIST_REQ (21), version 2, network 2, gateway ID 245,
hash null
Connected to gateway
```

Figure 6.2: A prototype of the firmware management user interface that can be used for manually creating and transferring firmware parameters.

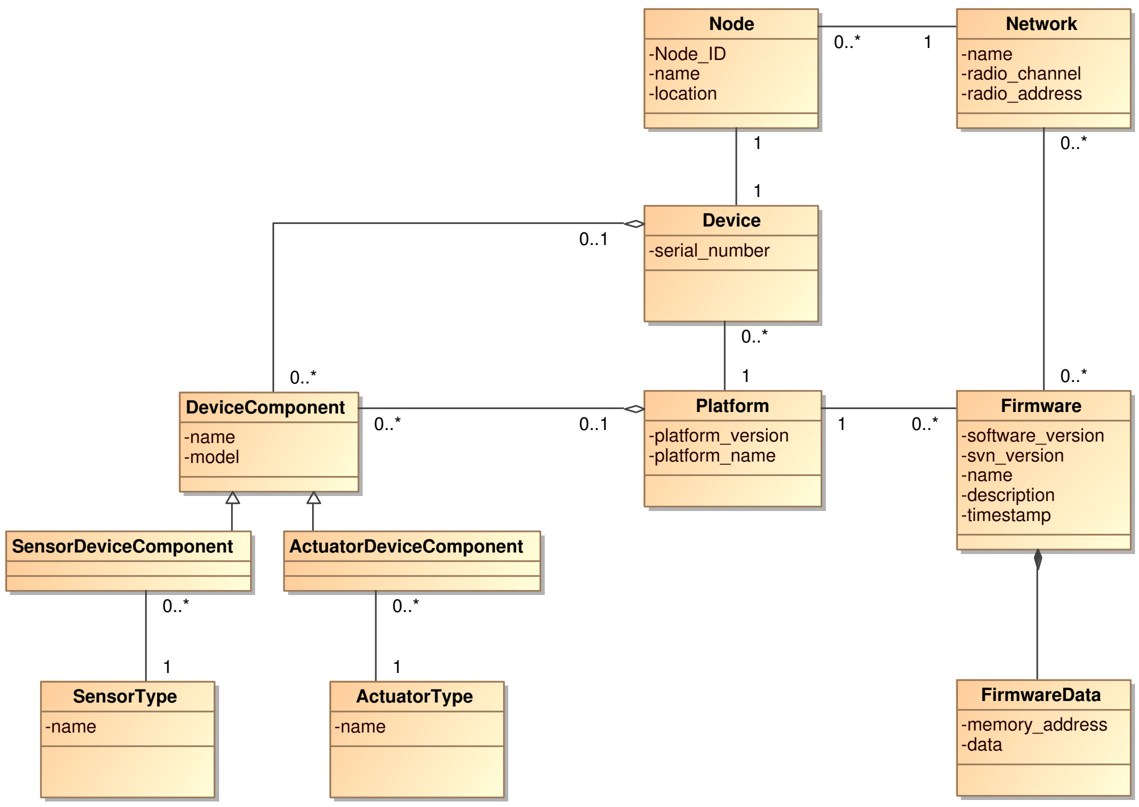


Figure 6.3: Firmware management information in the SNAP configuration database.

contains multiple nodes that are stored in *Node* objects with other node-specific information. These logical nodes are mapped to physical node devices with the *Device* objects. Each physical device is of a certain platform type and contains sensors and actuators. Platform information about a particular node, such as the platform version, is stored in the *Platform* object. Sensors and actuators are represented with *SensorDeviceComponent* and *ActuatorDeviceComponent* objects that are inherited from the *DeviceComponent* object. Each Platform object is linked to multiple DeviceComponents to indicate what sensors and actuators the platform supports. Each Device object is also linked to those DeviceComponents that are equipped to that physical sensor.

Adding support for storing firmware images requires defining two new tables. The *Firmware* object contains information about a particular firmware image: the automatically generated version number, the version control system version number, the name, the description, and a timestamp of when the firmware image was created. The actual binary data of the firmware image is divided into multiple *FirmwareData* objects. Each FirmwareData object contains a section of the firmware image and a memory address that indicates where that section of the firmware image belongs in the nodes program memory.

Firmware and Platform objects are linked in order to indicate what Platforms the Firmware object supports. Each Firmware can be mapped to multiple Network. This allows operators to use an identical firmware image in multiple WSNs, which makes it faster to move node from one network to another if they use the same firmware. Mapping one firmware image to multiple WSNs also helps to keep the number of different firmware images to a minimum.

6.4 Disseminating Firmware Images

This section describes how a new firmware image can be disseminated into a WSN using five steps. First, source code files are compiled to create a firmware image that supports the requirements of the WSN and is compatible with the deployed nodes. Second, the firmware image is uploaded to a database and information about the firmware image is transferred to the gateway nodes of the WSN. Third, gateway nodes advertise the firmware to the nodes inside the WSN. Fourth, gateway nodes fetch the firmware image from the database and transfer it to the neighboring nodes, which then disseminate the image further into the WSN. And finally, the bootloader is used to make sure that only successfully received firmware images are executed.

6.4.1 Creating a Firmware Image

Producing a working firmware image requires the source code files to be compiled and the resulting object files to be linked together to form a file that can be executed by the microcontroller of a node. Each firmware image also contains descriptive information: verification values for validating the integrity of the firmware image, software version numbering, platform compatibility information and descriptive text. This information is monitored during the lifetime of a node to keep track of the different firmware images and their features. In order to make uploading firmware images in the database faster for the developer, the TUTWSN compilation process automatically creates an XML file that describes the generated firmware image, as shown in Program 6.1.

6.4.2 Injecting Firmware Images with Autoconfigurator

There are three ways of injecting a new firmware into TUTWSN nodes. First, a WSN operator may manually update the firmware image of a single node and then use this node to disseminate the new firmware image to the rest of the network. This is not always applicable, as it requires the operator to have access to the WSN in order to update one of the nodes.

Second, a WSN operator may use a special *cloner* node that is programmed with the new firmware image which the cloner can wirelessly transfer to a receiving node. The cloner node is useful in situations, where physical access to the WSN installation

```
<?xml version="1.0" encoding="UTF-8" ?>
<firmware date="2010-01-13 16:50">

  <!-- Network information -->
  <network radio_address="ADDDCCBBAA"
           radio_channel="10" />

  <!-- The actual file containing the firmware image -->
  <image filename="firmware.hex"
         md5_checksum="c753977bca6f76a8c6fe503f38ce4eb2" />

  <!-- Software information -->
  <software sw_version="33000"
            svn_version="5402M"
            bootloader="yes"
            autoconfig="yes" />

  <!-- Hardware/platform information -->
  <hardware target="headnode"
            platform="simple"
            platform_version="v2c"
            hw_version="25856" />

</firmware>
```

Program 6.1: An XML document for a firmware image that has been automatically created during compilation

is possible but moving installed nodes or bringing programming equipment is difficult or expensive.

Third, a WSN operator may use one or more gateway nodes to inject the firmware image into the WSN, as shown in Figure 6.4. This is done by using the firmware management user interface to send a firmware image and the firmware description as an XML document to the Autoconfigurator service, which parses the XML document and stores the image in the firmware management database. The operator can then ask the Autoconfigurator service to fetch a list of the stored firmware images and transmit this list to the gateway nodes, which will advertise the listed firmware images into the WSN. Using gateway nodes to disseminate firmware images is more suitable for real-world deployments, as the firmware image can be disseminated without manual programming or physical access to the WSN. Furthermore, this will significantly speed up the dissemination speed of the firmware images as the firmware image can be injected into the WSN from multiple points at once.

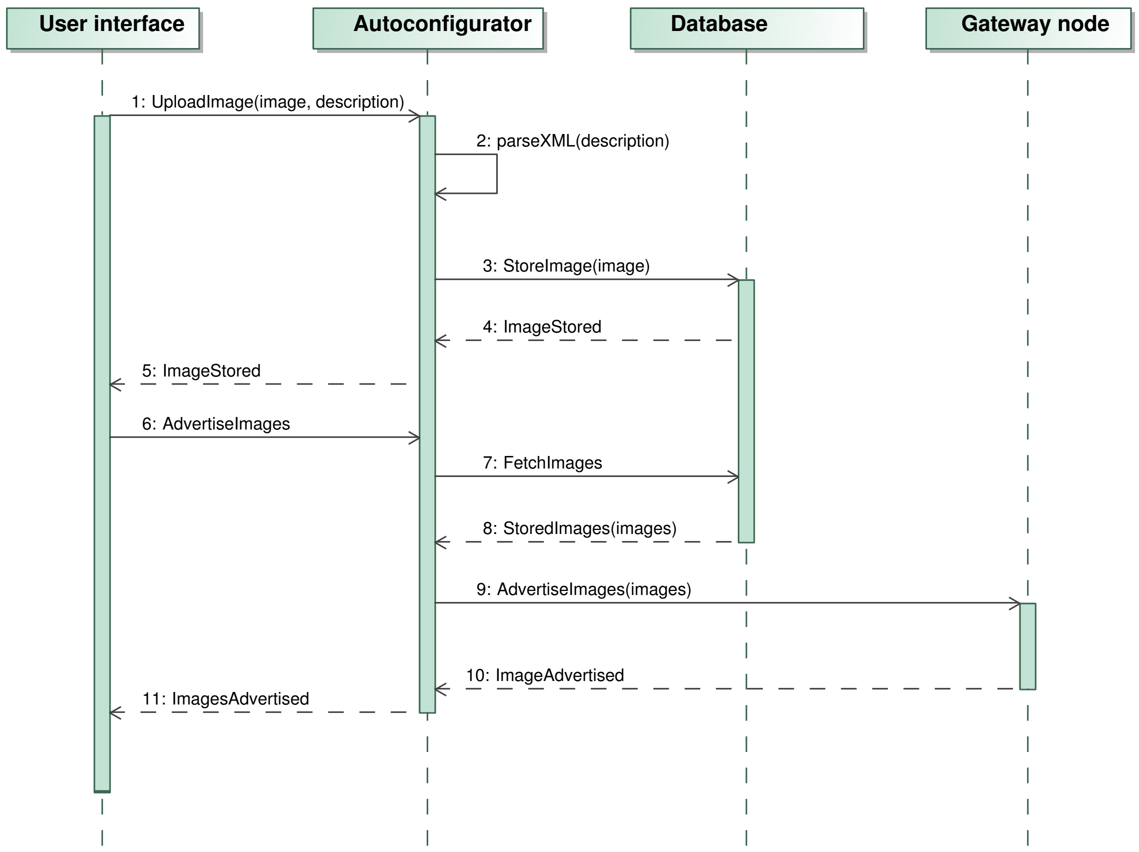


Figure 6.4: Storing and advertising a firmware image.

6.4.3 Software Advertisements in the Firmware Image Transfer Protocol

The firmware image transfer protocol uses software advertisements as a way for nodes to share information about their firmware image. Software advertisements can have a significant effect on the performance of a dissemination protocol. Too frequent advertisements create unnecessary overhead and increase energy consumption. Conversely, infrequent advertisements can limit the propagation speed of a firmware image. There are two mechanisms for performing software advertisements with TUTWSN.

First, nodes advertise their firmware images periodically on a predefined advertisement channel, as shown in Figure 6.5. After each advertisement the source listens for a reply. The parameter T_a adjusts the interval between these *idle advertisements*. Increasing the frequency of idle advertisements also increases the energy consumption in the source but decreases the average listening time of the receivers.

Idle advertisements allow nodes to perform code acquisition even if their protocol stacks might otherwise be incompatible. Furthermore, if a node encounters a fatal problem while reprogramming, it may use the idle advertisements to find a new

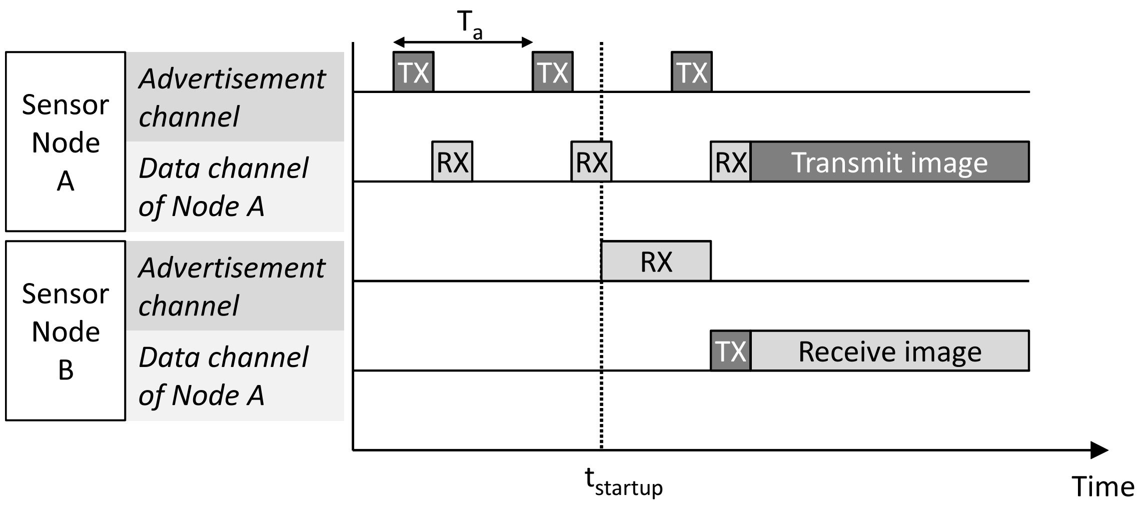


Figure 6.5: Example of idle advertisements. Node A periodically transmits advertisements with interval T_a. At $t_{startup}$, node B powers up, scans the advertisement channel and receives an advertisement from node A. Node B then fetches the new firmware image from node A using the channel of Node A.

source for image acquisition.

When integrating firmware management into TUTWSN, the TUTWSN network channel is used as the advertisement channel, so that nodes can receive idle advertisements while they are scanning for nearby clusters.

Second, software versions are compared when a node connects with its neighbors, as shown in Figure 6.6. Depending on the WSN architecture, this exchange of version information can be performed either on the MAC or the routing layer. With TUTWSN, it is performed when nodes associate on the MAC layer. Adding the version information into the association protocol adds a small overhead, but significantly improves the propagation speed of new firmware images without requiring major changes.

Code acquisition begins automatically when a node detects that one of its neighbors has a new version of a compatible firmware image. The node with a lower version number sends a *software request* and the other node responds with a *software confirmation*. After this, the nodes invoke the firmware transfer protocol. It is important to note that the software advertisements are exchanged on a hop-by-hop basis between neighbors without flooding the advertisements further into the network. If a network contains multiple nodes with different platform identification numbers, it may cause network segmentation.

6.4.4 Firmware Transfer Protocol

The firmware transfer protocol is responsible for reliably transferring a complete firmware image from one node to another. Typical firmware transfer protocol store

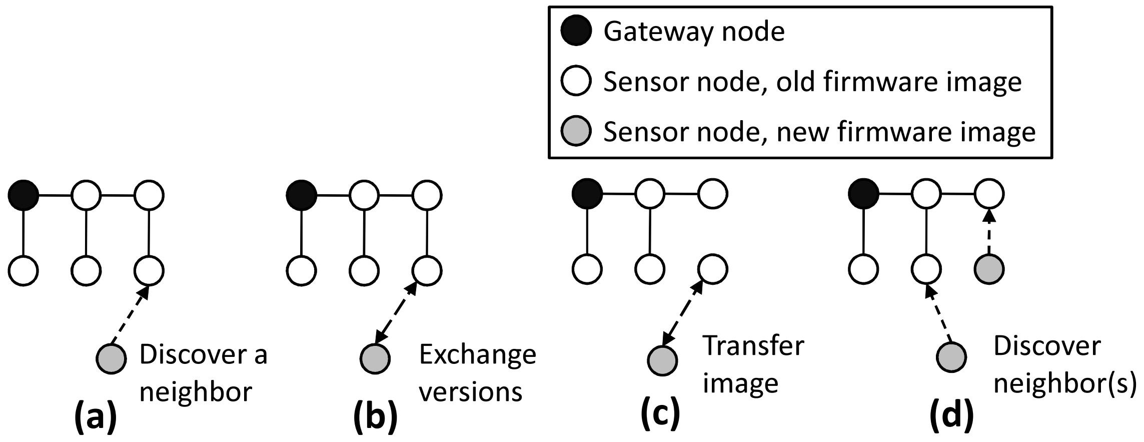


Figure 6.6: a) A node enters the network and discovers a neighbor. b) Nodes use software advertisements to exchange information about their firmware images and decide if they want to perform a transfer. c) Nodes use the firmware transfer protocol to update the node that has the old version. d) Nodes reboot, discover new neighbors and disseminate the new firmware image further.

firmware images in a temporary memory device [48]. But as TUTWSN nodes do not have an external memory for storing firmware images, the firmware transfer protocol must overwrite the main WSN stack as soon as the firmware transfer begins. Thus, the protocol stack of the firmware transfer protocol and the main TUTWSN stack are separated, as shown in Figure 6.7.

The separation of the memory sections and the lack of the external memory mean that the firmware transfer protocol cannot update itself. Therefore, all services used by the firmware transfer protocol are stored within this memory segment. On the other hand, the number of included services must be kept to a minimum as they increase the size of the memory segment and potentially introduce program errors. Consequently, the firmware transfer stack includes only the necessary modules to perform the firmware image transfer and the program memory rewrite. The firmware transfer protocol and the main program are never executed concurrently. The firmware transfer protocol can therefore utilize data memory segments during transfer that are normally reserved for the main program. Overlaying the data memory significantly reduces the data memory requirements of the firmware transfer protocol. Despite the overlaying, a small amount of dedicated memory is needed for passing information between the main program and the firmware transfer protocol.

The firmware transfer protocol follows the general client-server architecture. The receiver of the firmware image acts as a client, while the sender acts as a server. Communication between the client and the server is performed at a channel selected by the server, which is transmitted within the software advertisements. Servers may choose to use a single network-wide dedicated channel for the image transfers or they may use an appropriate channel selection algorithm to choose channels that

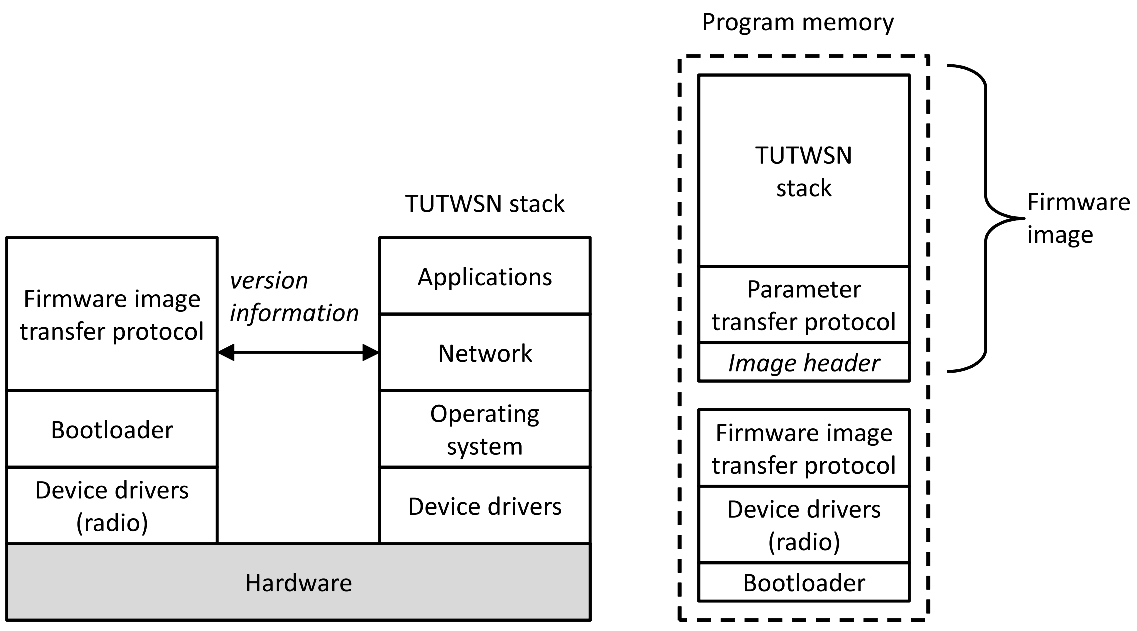


Figure 6.7: The logical structure and the memory layout of the firmware image transfer protocol and the TUTWSN stack. The firmware transfer protocol is a separate protocol stack that shares version information with the network layer in the WSN stack. The program memory contains a separate section for firmware transfer protocol and the main firmware image. The firmware parameter transfer protocol is also stored within the firmware image.

are not being used. Choosing different channels is preferred, as this lowers the change of collisions with nearby image transfers or normal WSN communications. With TUTWSN, the software advertisements use the cluster channel of the MAC layer as the server channel to minimize the number of collisions between concurrent program image transfers.

The client begins by requesting the header of the firmware image, as shown in Figure 6.8. Once the header is received, the client marks the current header invalid and requests the contents of the firmware image in segments. After each segment the client immediately writes the data to the program memory, thus invalidating the previous firmware image. After the whole firmware image is received, the client validates the image by calculating the message authentication code of the received firmware image and comparing it to the one in the header. If the calculated code matches the one received in the header, the header is marked valid. Otherwise, the header remains marked invalid. After the transfer the client and the server reboot.

6.4.5 Bootloader

Information about a firmware image is stored in a header, which is placed at the beginning of the firmware image in the program memory of the node, as shown in Table 6.1. First, firmware images are identified by a combination of a 16-bit platform identification number and a 16-bit software version number. The platform

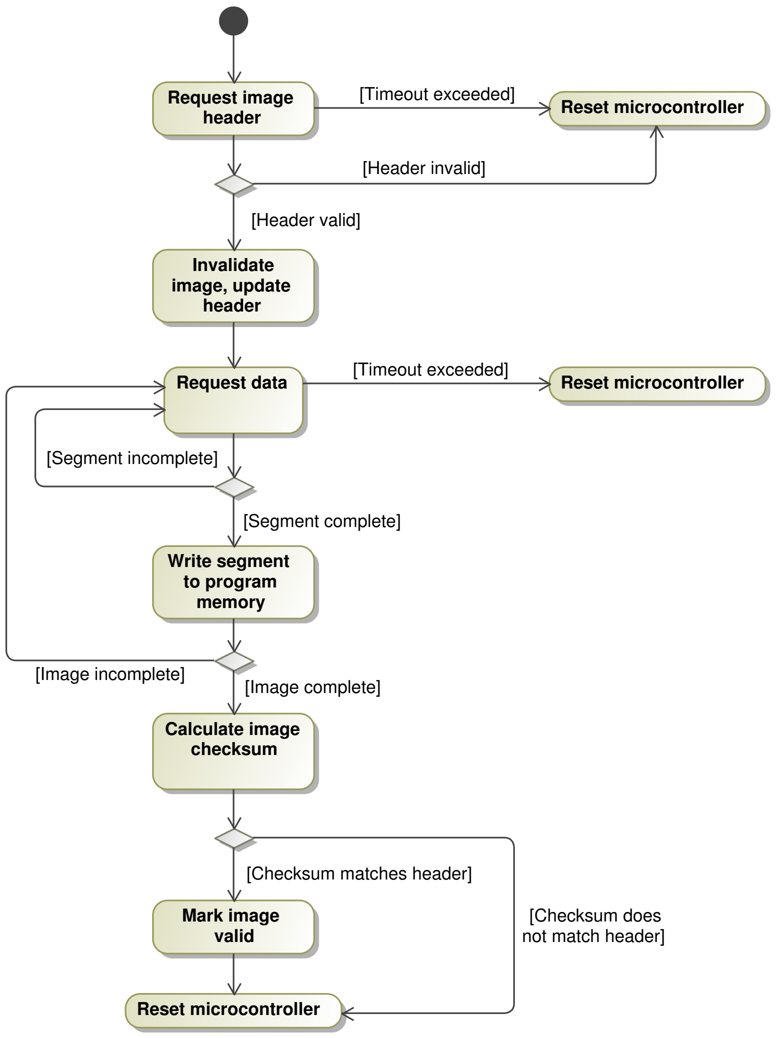


Figure 6.8: The firmware transfer protocol first requests and validates the header, then requests the firmware image in segments and finally validates the complete image.

identification number is used to limit the transfer of firmware images between incompatible nodes, while the version number is used to identify different versions of the same firmware. An 8-bit hash value is calculated from the identifiers in order to save bandwidth during image transfers. Firmware images are signed with a Rivest Cipher 4 message authentication code, which allows the receiver to detect errors in the received firmware image and validate that the image is from an authorized source, as the authentication key is kept secred. The distribution enabled bit controls whether or not the node is allowed to advertise its firmware image. This bit is always enabled in end-user deployments but during testing it can be disabled to easily limit the dissemination of firmware images that contain untested code. The header contains a valid bit that indicates whether or not the firmware image has been successfully received and validated and can be safely executed.

When a TUTWSN node powers up and begins executing code, it first executes a

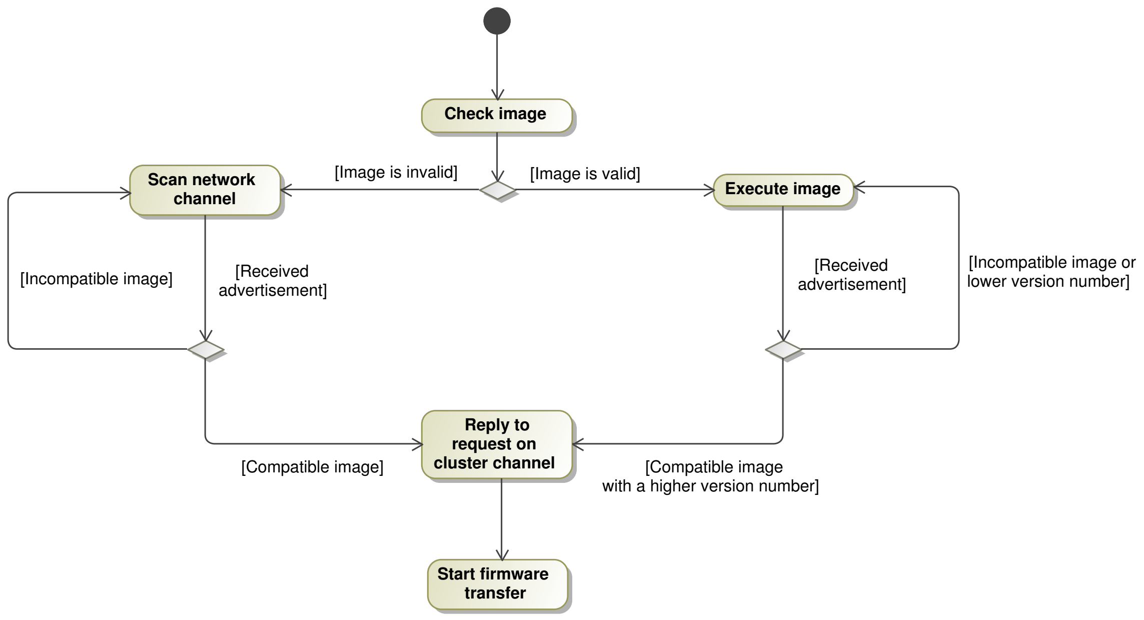


Figure 6.9: During startup the bootloader program checks that the firmware image is valid. If the image is valid, it is then executed. The firmware transfer protocol is invoked if the firmware image is invalid or if a software advertisement is received.

program called the *bootloader*, as shown in Figure 6.9. The bootloader is responsible for verifying that the main firmware image is a valid executable. This is done by checking the state of the valid bit in the header of the firmware image. If the image is marked valid, the bootloader immediately begins executing the WSN stack inside the firmware image from a predefined memory location. When software advertisements for a new firmware image are received, the stack may then invoke the firmware transfer protocol to update itself. If the valid bit is set to an invalid state, the bootloader begins scanning the TUTWSN network channel to detect idle advertisements. Once an advertisement for a compatible firmware image is received, the bootloader begins executing the firmware transfer protocol.

Table 6.1: Contents of the header in a firmware image.

Field	Length (bits)
Hardware identifier	16
Software version	16
Hash	8
Message authentication code	32
Distribution enabled	1
Image valid	1
Total	*74*

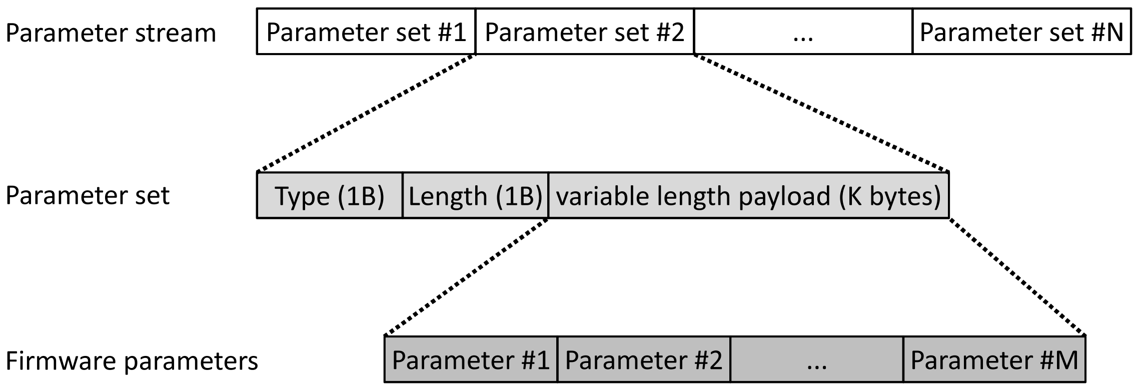


Figure 6.10: Firmware parameters from different modules are combined to form a parameter stream.

6.5 Transferring Firmware Parameters

In addition to transferring firmware images, TUTWSN supports configuring them during runtime. Configuring three steps.

First, the Autoconfigurator service fetches the parameters of a particular node from the firmware management database and compresses them into a parameter stream.

Second, this parameter stream transferred to the target node using either the single-hop or the multihop parameter transfer protocol. If the node has not been configured at all, the parameter stream is transferred by the single-hop parameter transfer protocol. On the other hand, if the node is already deployed and a member of a WSN, the parameter stream is transferred over the existing TUTWSN Node API using the multi-hop parameter transfer protocol. Once the node receives the parameters, it uses the parameter handler interface to store the parameters in persistent memory and reconfigure itself accordingly.

6.5.1 Firmware Parameters

Each firmware parameter of a node belongs to a particular protocol layer, device drive, library or application module. These modules can define *parameter sets* that contain the firmware parameters that module requires, as shown in Figure 6.10. Multiple parameter sets from different modules are combined to form a *parameter stream* which contains all the firmware parameters the node requires.

Each parameter set is identified with a type field that indicates which module owns the module parameters inside the set. Each set also contains a length field that indicates the size of the payload section of the parameter set. Using a variable length payload section and an explicit length field allows modules to encapsulate the firmware parameters. Thus, parameter sets can be stored, reordered and transferred without knowing the exact contents of the payload section. Furthermore, new parameter sets and firmware parameters can easily be added without requiring changes in other modules.

6.5.2 Single-hop Parameter Transfer Protocol

When a node powers up for the first time, it is in an uninitialized state because it has not yet received its firmware parameters. The node cannot execute the TUTWSN protocol stack as it does not know its node ID or the network parameters. A single-hop parameter transfer protocol was designed so that these uninitialized nodes could receive their initial firmware parameters during the manufacturing or the deployment phase. The single-hop transfer protocol uses the same serial number that is used to identify the physical hardware to address the nodes. Nodes can only be identified if they already know their serial number: nodes can use a hardware serial number memory chip or each microcontroller can be factory programmed with firmware that contains the serial number.

If a node does not know its serial number, it must be identified manually by an operator using a user interface, as shown in Figure 6.11. After powering up, an uninitialized node begins periodically broadcasting requests on a predefined radio channel, which is called the *configuration channel*, until it receives a reply.

As broadcasts from different nodes cannot be uniquely identified, the operator must make sure that only one node is powered up at the same time. The operator then commands a gateway node to listen to the configuration channel. Once the gateway node hears a request, it forwards it to the Autoconfigurator service, which shows a dialog to the operator on the firmware management user interface. The operator then types in the serial number of the node. The operator may now either manually type in the firmware parameters of the node or the Autoconfigurator may automatically connect to the SNAP configuration database and fetch the predefined firmware parameters. Once the firmware parameters are fetched, the Autoconfigurator transforms them into a parameter stream and transfers them to the node. Upon receiving and verifying the parameters, the node acknowledges the received parameter stream.

If each node knows its serial number, transferring the firmware parameters can be automated, as shown in Figure 6.12. This allows preprogrammed nodes to be directly shipped from the manufacturer to the WSN end-user and configured on-site. Instead of manually configuring each node individually, the nodes automatically begin broadcasting requests.

As the nodes know their serial number, the single-hop parameter transfer protocol can now identify individual nodes. The Autoconfigurator service periodically asks each gateway node in every WSN to listen for the configuration channel and notify

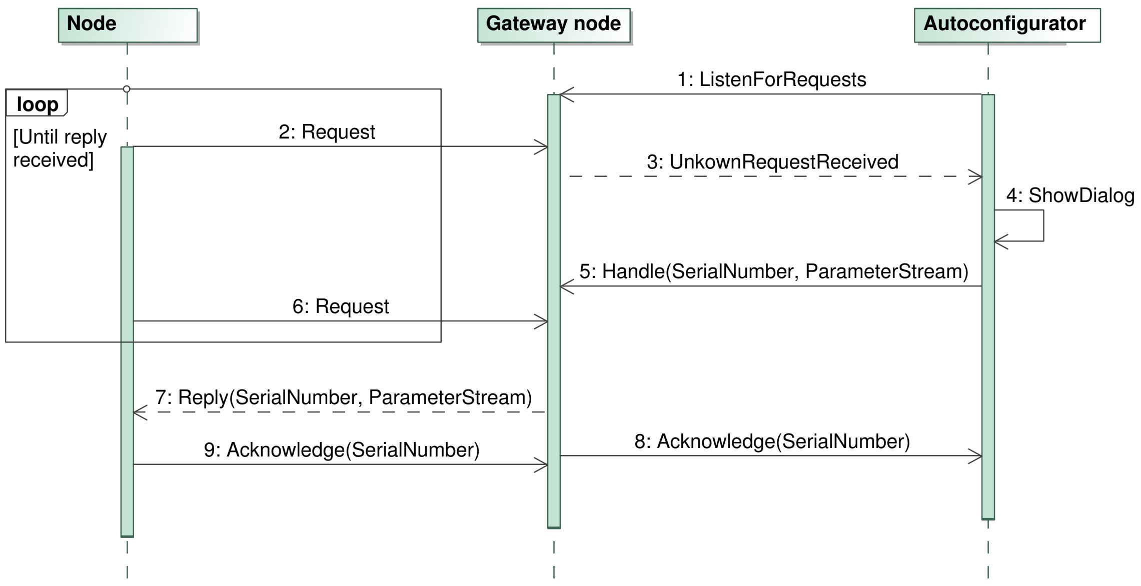

Node
Gateway node
Autoconfigurator
loop
[Until reply received]
1: ListenForRequests
2: Request
3: UnkownRequestReceived
4: ShowDialog
5: Handle(SerialNumber, ParameterStream)
6: Request
7: Reply(SerialNumber, ParameterStream)
8: Acknowledge(SerialNumber)
9: Acknowledge(SerialNumber)

Figure 6.11: Transferring parameters to a node without a serial number requires that an operator identifies the node.

it if any node is sending requests for firmware parameters. If a request is received, the Autoconfigurator service reads the serial number of the request and fetches the firmware parameters from the firmware management database. The firmware parameters are then transmitted to the node.

Whether or not a node knows its serial number, the communication protocol for transferring parameter streams between a node and a gateway node remains the same. The node begins the transfer by sending a request PDU, as shown in Table 6.2.

The source field indicates if the packet is from a node or from a gateway node. In this case, the source field is set to zero to indicate that the PDU was sent by a node. An alternating bit protocol is used as the sequence number. Therefore, the node chooses an initial value for the sequence bit, which is then inverted every time a reply has been successfully received. The PDU type field indicates if the PDU is a new request or a reply to a PDU.

The source address is the serial number of the node. If the node does not know its serial number, the address field is set to zero. Finally, the packet contains a version number to indicate what version of the simple parameter transfer protocol the node supports.

The gateway node responds with a parameter stream reply PDU, as shown in Table 6.3. In this case, the source field indicates that the packet originates from a gateway node. The gateway node replies to the request using the same sequence number it received in the request PDU. The reply PDU has a maximum payload length of 20 bytes. If the transferred parameter stream is longer, it must be sent in

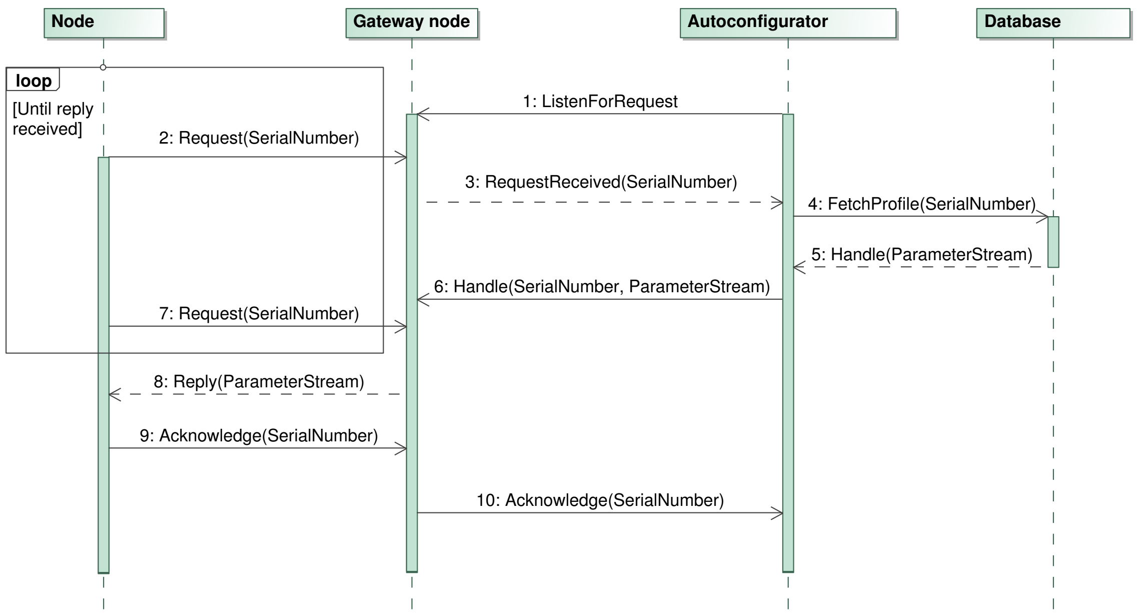

Node
Gateway node
Autoconfigurator
Database
loop
[Until reply received]
1: ListenForRequest
2: Request(SerialNumber)
3: RequestReceived(SerialNumber)
4: FetchProfile(SerialNumber)
5: Handle(ParameterStream)
6: Handle(SerialNumber, ParameterStream)
7: Request(SerialNumber)
8: Reply(ParameterStream)
9: Acknowledge(SerialNumber)
10: Acknowledge(SerialNumber)

Figure 6.12: Transferring parameters to a node with a serial number can be automated with SNAP and the Autoconfigurator service.

multiple reply PDUs. The end of stream indicator is used to mark whether or not the parameter stream ends in this PDU.

As the parameter stream might not need the whole payload section, the length of the payload is explicitly marked in the PDU. The target address field is taken from the request PDU to make sure that only the intended recipient of the parameter stream will receive the payload. When filling up the payload section with parameter sets, the gateway node only stores full parameter sets without splitting sets into smaller parts. Even though this often leaves parts of the payload section unused, it makes it easier for the node to process the payload. Finally, the payload is validated with a 16-bit CRC checksum.

An example transfer of a 25 byte long parameter stream is shown in Figure 6.13. The node begins by sending a request packet with a sequence number k. The

Table 6.2: Contents of a parameter stream request PDU.

Field	Length (bits)
Source (Node:0, Gateway:1)	1
Sequence	1
PDU Type (Request:0, Reply:1)	1
Source address	32
Protocol version	8
Total	*48*

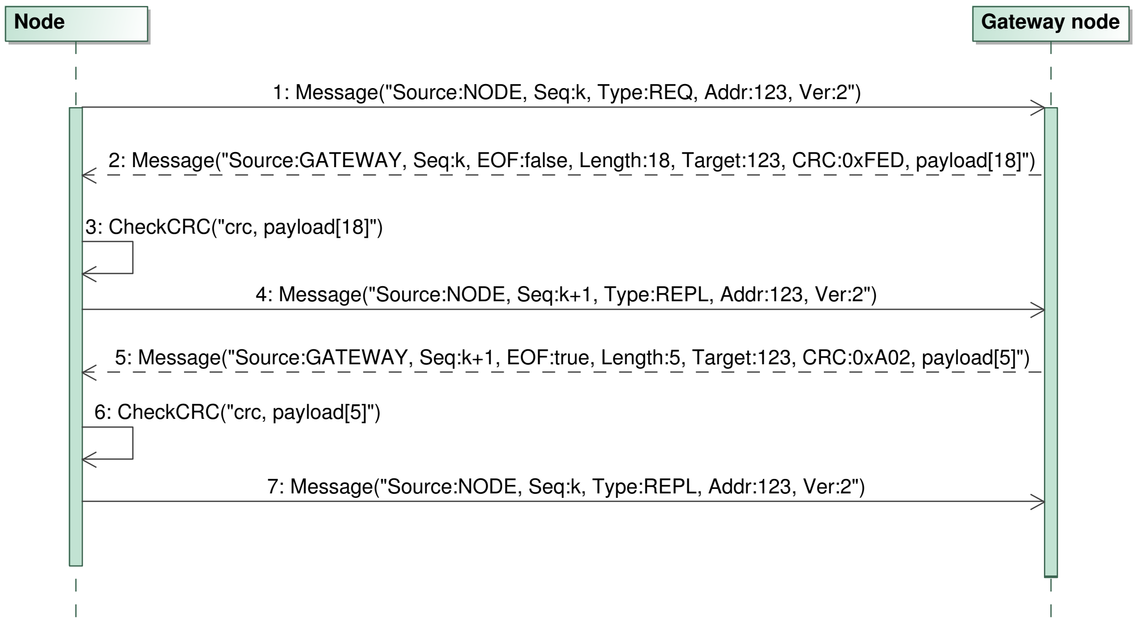

Node
Gateway node
1: Message("Source:NODE, Seq:k, Type:REQ, Addr:123, Ver:2")
2: Message("Source:GATEWAY, Seq:k, EOF:false, Length:18, Target:123, CRC:0xFED, payload[18]")
3: CheckCRC("crc, payload[18]")
4: Message("Source:NODE, Seq:k+1, Type:REPL, Addr:123, Ver:2")
5: Message("Source:GATEWAY, Seq:k+1, EOF:true, Length:5, Target:123, CRC:0xA02, payload[5]")
6: CheckCRC("crc, payload[5]")
7: Message("Source:NODE, Seq:k, Type:REPL, Addr:123, Ver:2")

Figure 6.13: An example transfer of a parameter stream. The parameter stream does not fit into a single PDU and, therefore, it is sent in two parts.

gateway node receives the request and replies with the same sequence number. As the whole parameter stream does not fit into a single PDU, the gateway node splits it into two parts and marks the first PDU with the end of stream bit set to false. The node receives the reply, verifies the CRC checksum of the payload section and processes the payload. Then, the node sends a reply with a sequence number $k + 1$ to indicate that the previous PDU was successfully received. As the end of stream bit was set to false, the node knows to wait for additional PDUs. The gateway node then sends the rest of the parameter stream and sets the end of stream bit to true. Finally, the node receives the final reply, verifies it and sends the acknowledgement to the gateway node with the sequence number k.

Table 6.3: Contents of a parameter stream reply PDU.

Field	Length (bits)
Source (Node:0, Gateway:1)	1
Sequence	1
End of stream indicator	1
Payload length	5
Target address	32
CRC checksum of the payload	16
Parameter stream payload	160
Total	*216*

```
/*
 * Set default values for parameter set
 * and clear memory.
 */
void init_parameters(void);

/*
 * Take the firmware parameters from the argument
 * param_set and store them in the EEPROM memory.
 */
void store_parameters( ParameterSet_t* param_set );

/*
 * Read the firmware parameters from EEPROM and
 * process them.
 */
void enable_parameters(void);

/*
 * Calculate the CRC checksum over the firmware
 * parameters in the EEPROM.
 */
void calculate_crc(uint16_t* crc);
```

Program 6.2: Interface functions for the firmware parameter handlers

6.5.3 Multihop Parameter Transfer Protocol

As the single-hop parameter transfer protocol is designed for uninitialized nodes and does not support routing, it does not meet the requirements for transferring updated firmware parameters to nodes that have been already deployed. Instead, the TUTWSN Node API was used to implement a multihop version of the parameter transfer protocol. Compared to the single-hop protocol, the downside of the multihop version is that it offers a smaller payload size of 16 bytes and that it only supports updating existing firmware parameters. Otherwise, it functions in a similar manner to the single-hop parameter transfer protocol.

6.5.4 Firmware Parameter Handler Interface

Once a node has received one or more parameter sets, it must have methods for processing, storing and using the received firmware parameters. Program 6.2 presents a *parameter handler interface*, which program modules may implement if they wish to receive firmware parameters.

When a node powers up, as shown in Figure 6.14, it calls the `calculate_crc` -method for each module, which causes the modules to read their stored firmware parameters from the EEPROM memory and to calculate a CRC checksum from them. This value is compared to a previously stored CRC checksum. If the CRC cal-

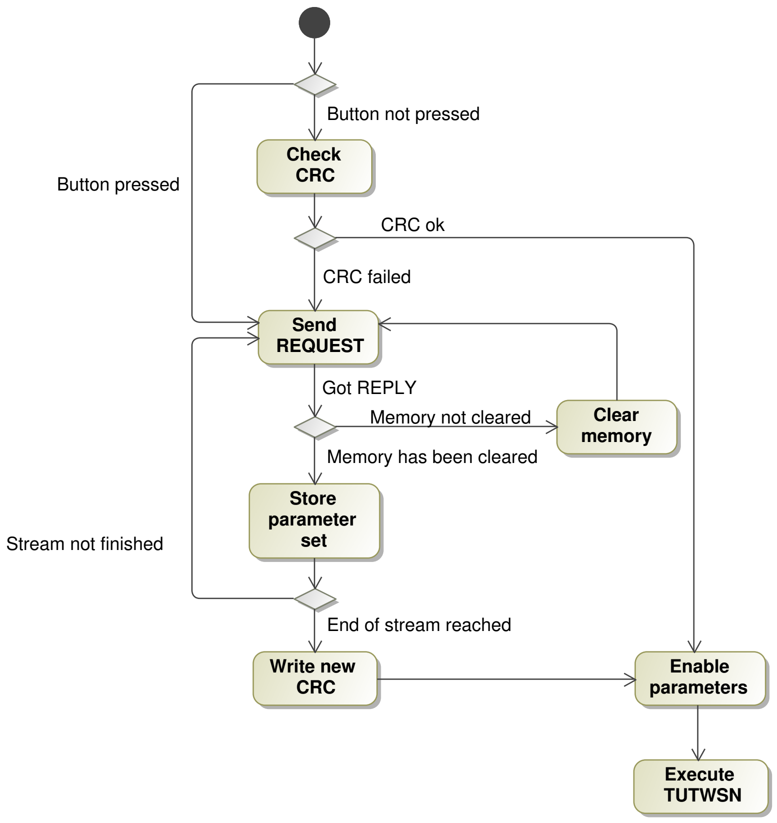


Figure 6.14: Processing parameters in a node

culations match, the `enable_parameters`-method is called for each module, which causes them to read their firmware parameters and stores them in local variables. Then, the execution of the TUTWSN protocol stack is started. Conversely, if the CRC calculation results do not match due to memory corruption or if the push button of the node is pressed during startup, the node enters the single-hop parameter transfer protocol loop.

There may be a large number of modules that implement the parameter handler interface. Not all of these modules are always active nor do they always need their parameters. A motion detection application may implement the parameter handler interface and define many firmware parameters that control its function but only a few nodes in a WSN might use this application. In order to minimize the amount of parameters transferred, the handler interface has a method for disabling modules. When a node receives parameters, it first checks whether or not the firmware parameters have been initialized. If not, the `init_parameters`-method is called. This causes each module to revert back to default values for each firmware parameter. Then, only those parameter sets that that particular node requires are transferred while the rest of the firmware parameters stay in their default or disabled state.

During the parameter transfer, a group of parameter sets is received at once. As each parameter set contains the type field, it can be dispatched to the correct module in a straightforward way using the `store_parameter`-method. The receiving module will parse the firmware parameters from the set and store them in the EEPROM memory. After the whole parameter stream has been received, the CRC calculation is performed and the new checksum value of the firmware parameters is stored in the EEPROM memory. Finally, the `enable_parameters`-method is called again and the execution of the TUTWSN protocol stack is started.

7. EVALUATION OF FIRMWARE MANAGEMENT

This chapter presents the evaluation of firmware management with TUTWSN. First, the implementation details, such as memory usage and the implemented parameter sets, are presented. Second, the firmware transfer protocol is evaluated in four different test scenarios ranging from link reliability measurements to updating a deployed WSN. Third, the use of firmware management ing managing a large scale WSN is evaluated by using a large campus WSN, where firmware management has been extensively used to both update and configure the nodes. The use and benefits of firmware management in the manufacturing of TUTWSN nodes is evaluated.

7.1 Implementing Firmware Management

The implementation of firmware management does not yet cover the whole TUTWSN infrastructure from end to end. Instead, the implementation effort has been directed at the firmware transfer protocol and the single-hop parameter transfer protocol. Implementing these two firmware management protocols allows verifying and testing the core features and evaluating their usability in different scenarios. Once the evaluation phase is complete, the implementation will focus on adding support for firmware management in SNAP and implementing the multi-hop parameter transfer protocol over the Node API layer.

The firmware transfer protocol and the single-hop parameter transfer protocol have been integrated into the main TUTWSN protocol stack. In total, the implementation took a little over four thousand lines of C source code files, as shown in Table 7.1. The program memory consumption of the firmware transfer protocol and the single-hop parameter transfer protocol are 4130 and 2330 bytes, respectively. The implementation of firmware management corresponds to a total of 6460 bytes or 4.9% of the 128 kilobytes of program memory, as illustrated in Figure 7.1.

Table 7.1: Lines of code and comments in the implemented protocols. The single-hop protocol is implemented separately for gateway nodes and nodes while the firmware transfer stack implementation is identical for both node types.

Component	Lines of code	Lines of comments
Single-hop protocol, node	402	193
Single-hop protocol, gateway node	445	130
Single-hop protocol, common code	151	121
Firmware transfer stack	2321	783
Total	*3319*	*1227*

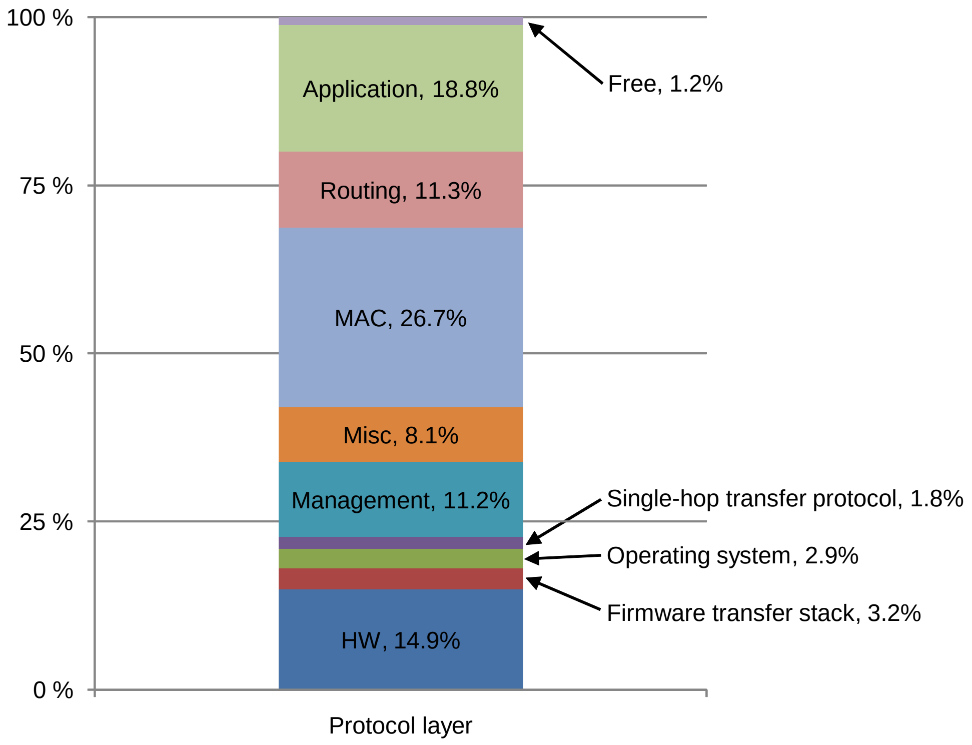


Figure 7.1: ROM consumption. 128 kilobytes of program memory is divided between the protocol layers. The significant portion of the MAC layer is due to the complexity of the synchronized FTDMA algorithm.

Table 7.2: Implemented parameter sets and firmware parameters.

Parameter set	Firmware parameter	Length (bytes)	Description
Network	`channel`	1	Network channel of the MAC layer.
	`phy_address`	5	Radio address of the physical layer.
Node Management	`role`	1	Node role.
	`node_id`	3	Node ID of node.
Node API	`applications`	8	64 bits long bit array of the enabled applications.
Motion application	`mot_devices`	1	8 bits long bit array of enabled motion detection devices.
Analog application	`adc_devices`	1	8 bits long bit array of enabled analog input devices.
Hardware	`serial_number`	4	Serial number of the device.

The implemented firmware transfer protocol supports software advertisements and disseminating firmware images within a multihop WSN. As the database support through SNAP has not yet been implemented, the firmware transfer protocol cannot be used remotely through gateway nodes. Instead, the current supported method for injecting a new firmware image consists of manually programming a single node with a new image or using a cloner node to transfer a firmware image from one node to another.

Support for the single-hop parameter transfer protocol was implemented in both gateway and nodes and in the TUTWSN Gateway. Several firmware parameters and parameter sets were implemented for the protocol layers of the TUTWSN stack, as shown in Table 7.2.

The `channel` and `phy_address` parameters allows nodes to be moved from one WSN to another, while the `node_id` parameter allows changing the Node ID of a node and replacing a broken node by reusing the same Node ID.

Higher level parameters `applications`, `mot_devices` and `adc_devices` control what device drivers and applications are enabled in a node. A `serial_number` parameter was created so that a serial number could be easily given to uninitialized nodes.

Table 7.3: Firmware transfer test cases.

Test case	Number of nodes	Purpose
#1	2	Measure the the firmware image transfer time between two nodes.
#2	2	Measure how the distance between nodes and the antenna orientation affects the reliability of the radio link.
#3	25	Measure how effectively a firmware image spreads in a dense WSN.
#4	25	Measure how effectively a firmware image spreads in a sparse WSN.

7.2 Firmware Transfer Protocol Performance Evaluation

The performance evaluation of the firmware transfer protocol was done in four test cases using 2.4 GHz TUTWSN nodes, as shown in Table 7.3. First, the transfer time of a firmware image over a single hop was measured in test case #1. Second, the reliability of the firmware transfer as a function of the radio link length was measured in test case #2. Third, the update time for a WSN of 25 closely placed nodes was measured in test case #3. In test case #4, the same 25 node WSN was deployed in an office environment and the update time was measured.

The results of the test case #1 are shown in Table 7.4. The time to manually program a node with a Microchip ICD2 device, or T_{prog}, is included for reference. The firmware image was transferred in 4880 packets which each contained 27 bytes of payload. By calculating a sum of the individual times

$$T_{update} = T_{advertise} + T_{transfer} + T_{verify} = 88.0s,$$

we can estimate the minimum time it will take for a node to notice a new firmware image, transfer the image and verify it.

In test case #2, a firmware image was transferred between two nodes. The tests were done using channels 41 and 101, which correspond to frequencies of 2.441 GHz and 2.501 GHz, respectively. These frequencies were chosen so that the interference caused by WLAN access points that operate near channel 41 could be observed. Channel 101 is on a higher frequency that did not contain interference. Both nodes were within a line of sight of each other and rotated so that they either faced each other straight on or were side to side, as shown in Figure 7.2. This was done in order to see how the orientation of the node and the antenna affects the radio link reliability and what was the worst case performance, as nodes in a typical deployment

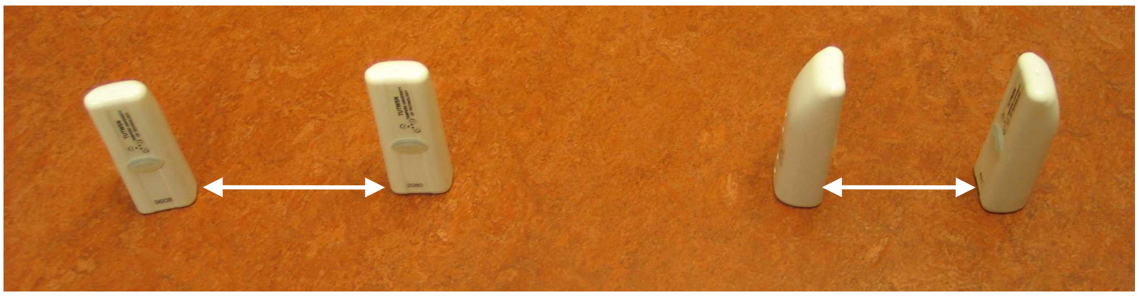


Figure 7.2: Test setup for test case #2.

are installed in an ad hoc manner without considering the antenna orientation. The reliability of the radio link was calculated as the ratio between the number of packets received by the number of packets sent.

Results of test case #2 are shown in Figure 7.3. When the nodes were placed head on, the reliability of the radio link was high and only a few packets had to be resent. Reliability of channel 41 dropped when the distance approached 15 meters. This was due to a nearby WLAN access point, that was active near the same channel. When the nodes were placed side to side, the reliability of the radio link suffered and the maximum reliable transfer distance dropped to around 20 meters on both channel.

For test case #3, a small, high density WSN of 25 nodes was deployed in an office environment. The nodes were placed in a five times five matrix where the distance between nodes was approximately 20 centimeters. Each node executed a standard TUTWSN protocol stack and took temperature and luminance measurements every few minutes. The test case was started by updating one of the nodes with a cloner node. The purpose of this test case was to see how the software advertisement protocol handled a dense active WSN, where each node could hear many neighboring nodes. The test was repeated twice.

Table 7.4: Results of test case #1.

Symbol	Time (s)	Description
T_{prog}	69.0	Manually programming a node.
$T_{transfer}$	51.0	Transfer time for a firmware image between two nodes in optimal conditions.
T_{verify}	24.3	Calculating the RC4 verification checksum for a received firmware image.
$T_{advertise}$	12.7	Average time to receive an advertisement from a updated node.

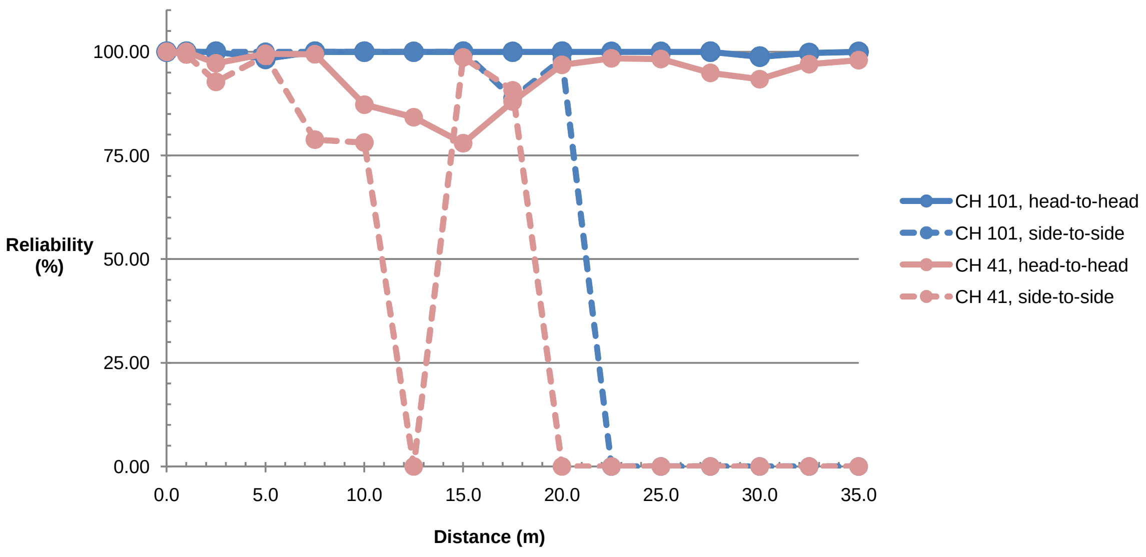


Figure 7.3: Reliability of individual packets in a firmware transfer as a function of distance in test case #2. Transfers on channel 41 were disturbed by interference from a nearby WLAN access point.

Results of test case #3 are shown in Table 7.5. During both measurements, over 7 concurrent updates were observed in the WSN. An exact reason for the few failed transfers was not discovered, but the high amount of firmware transfer activity might have caused interference. When a transfer failed, the bootloader correctly identified that the firmware image was corrupted and it restarted the firmware transfer. In the end, all the nodes were successfully updated.

When a node is updated, there is a small delay before the updated node begins spreading its firmware image further. This *software advertisement latency* contains the time it takes for the node to restart after receiving a firmware image, search and associate with its neighbors and exchange version information about its new firmware image.

In order to evaluate the latency of the software advertisements, the ideal update procedure was simulated, as shown in Figure 7.4. The results show that an ideal update process, where the advertisement latency is zero and the nodes immediately know when an update is available, takes approximately 340 seconds. When the ideal

Table 7.5: Results of test case #3.

Measurement type	Measurement #1	Measurement #2
Maximum simultaneous updates	8	7
Failed and restarted updates	4	1
Total time for update	611 seconds	849 seconds

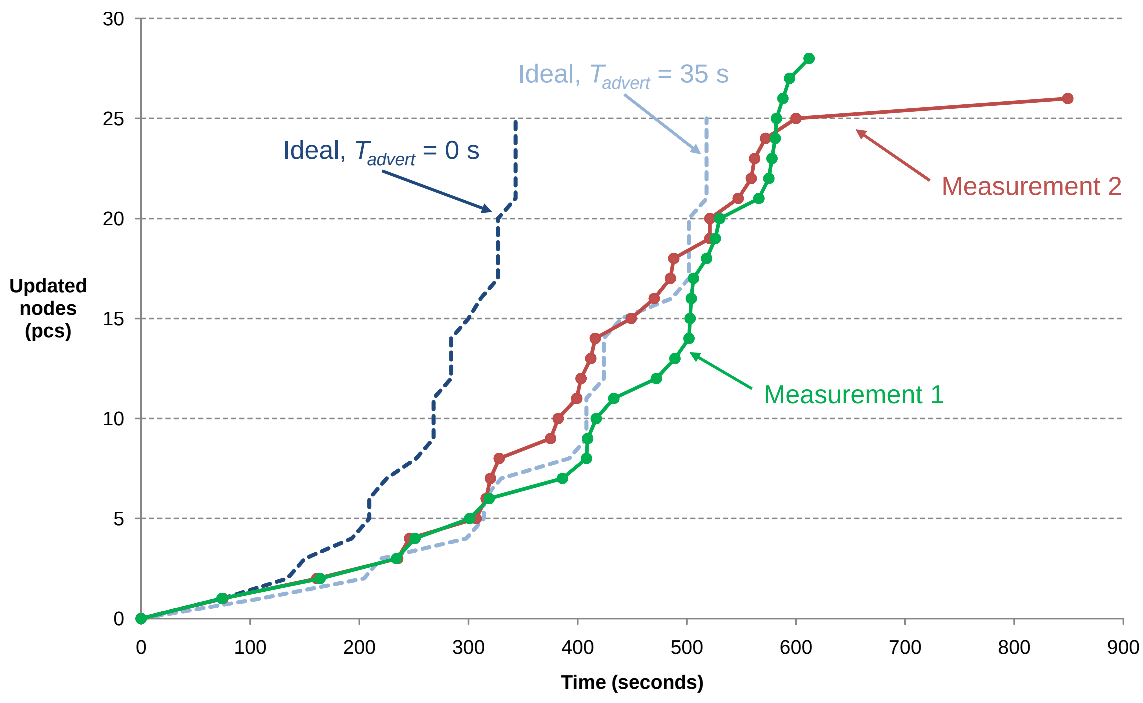


Figure 7.4: Updating a dense WSN of 25 nodes in test case #3. Time T_{advert} is the advertisement latency, which indicates how long it takes for an updated node to advertise its new firmware to neighboring nodes.

process is adjusted to include an advertisement latency of 35 seconds, the simulated performance is almost identical to the performance of the observed measurements.

In both measurements, the whole WSN was updated in about 10 minutes. In the second measurement one of the nodes took an unusual long time to update. This problem is inherent to the way the WSN was updated with a cloner node. Active software advertisements are sent when nodes associate with each other. Nodes always try to associate towards other nodes that are closer to the network sink. Therefore, if a node was associated directly with the network sink, it might not notice if an updated firmware image is spreading in the rest of the network. The node would only notice the new firmware image once it either performed a network scan to find new neighbors and received an idle advertisement, or if another, already updated, node associated with it. This delay would not occur, if the firmware image would be injected from the sink itself.

In test case #4, a 25 node WSN was sparsely deployed in an office environment, as shown in Figure 7.5. The purpose of this test was to see how a firmware image would propagate through a sparse WSN. Most of the WSN was updated in 20 minutes while the last two nodes were updated at 25 and 26 minutes, respectively. While the firmware image was spreading through the WSN, some of the nodes attempted to fetch the firmware image from updated nodes that were far away even when other nearby nodes had already been updated. This caused the transfer time

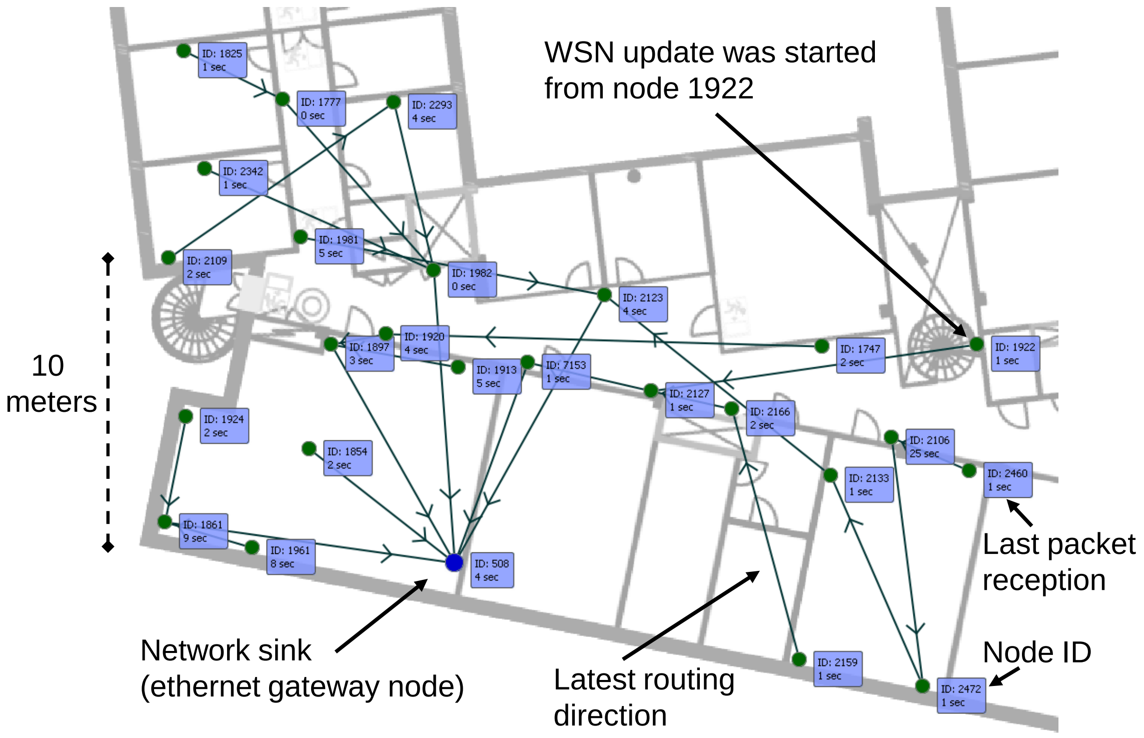


Figure 7.5: Updating a sparse WSN of 25 nodes in test cast #4.

of the firmware image to increase, as the reliability of the long radio link was low and packets had to be resent. This behavior is due to the aggressive nature of the current advertisement algorithm which causes nodes to fetch a firmware image immediately when they receive an advertisement for a new compatible image. The measurement data suggests that a more careful advertisement algorithm that considers link reliability might prove to be faster.

Based on these four tests, the implemented firmware management components met the stated requirements for firmware image transfers. Although a small number of firmware transfers failed at first, the bootloader made sure that the nodes restarted the transfers.

7.3 Usage of Firmware Management in a Campus-wide WSN

A large WSN of 268 nodes has been deployed at the Tampere University of Technology campus and it has been in active use since 2007. This campus WSN is used to both monitor the indoor climate at the university and also to provide a real WSN for students to use on WSN courses. The campus WSN contains several different types of nodes, including mobile nodes that students carry with them, as shown in Table 7.6. The stationary nodes are installed in public locations around six buildings on the campus grounds, as shown in Figure 7.6.

In addition to being one of the first WSN to be used for teaching, the campus

Table 7.6: Node types in the campus WSN.

Node type	Sensors	Number of nodes
Gateway node	temperature, luminance	7
Normal node	temperature, luminance	200
Carbon dioxide node	temperature, luminance, carbon dioxide	3
Surveillance node	temperature, luminance, infra-red	10
Humidity node	temperature, luminance, humidity	2
Humidity and surveillance	temperature, luminance, humidity, infra-red	7
Mobile node	temperature, luminance, acceleration	39
Total		*268*

WSN is also the first large scale deployment to use firmware management. Each node is programmed with an identical firmware image that supports updating and configuring the firmware image. Replacing broken nodes or moving sensors from one node to another only requires a course assistant to update the configuration of a node using the prototype firmware management user interface. As the multi-hop parameter transfer protocol has not yet been implemented, this has required the course assistant to physically move the target sensor near a gateway node to transfer the parameters. Regardless of this limitation, maintenance operations on the campus WSN have already been significantly faster than on previous, manually programmed, WSN deployments.

As the campus WSN supports updating firmware images, it is a suitable candidate for large scale testing of firmware image transfers. A firmware image was injected into the campus WSN from the fourth floor of the Tietotalo computer science building.

The updated firmware image first spread through the four floors of the Tietotalo building and then through the rest of the WSN in five hours, as shown in Figure 7.7. As the firmware image was injected from a single point using a cloner node, the propagation speed suffered from the same limitations that were observed in test case #3 in the previous section.

The propagation of the firmware image stopped for 30 minutes between the Rakennustalo civil engineering building and the Main building, where two adjacent nodes where routing data in opposite directions and did not associate with each other. Once the nodes exchanged advertisements, the update spread through

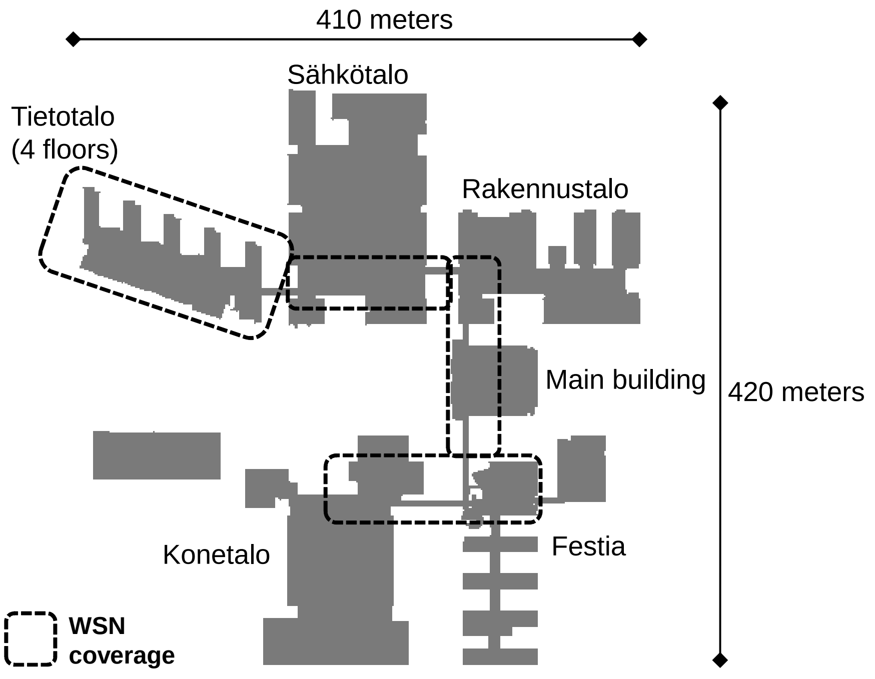


Figure 7.6: Map of the campus at TUT.

the rest of the WSN in little over an hour.

Regardless of the propagation delay, updating the campus WSN took significantly less time than manually updating the whole WSN, which can take several days. Furthermore, once firmware images can be injected through gateway nodes, updating a WSN like the campus WSN will happen in a fraction of the current update time.

7.4 Manufacturing Nodes with Firmware Management

In addition to updating and managing existing WSN, using firmware management in the manufacturing and testing of nodes was also evaluated. Previously, the reliability of assembled TUTWSN nodes has fluctuated from batch to batch. As the assembled nodes were shipped from the manufacturing site and stored before testing, problems in the soldering or the placing of components would surface weeks or months after the nodes had been assembled.

Since the fall of 2009, manufactured TUTWSN nodes have been built using pre-programmed microcontrollers. Each node contains an identical firmware image that can be configured at the manufacturing site using the single-hop parameter transfer protocol. The manufacturer has been supplied with the prototype firmware management user interface that can be used to quickly transfer predefined firmware parameters and a serial number to each node, as shown in Figure 7.8.

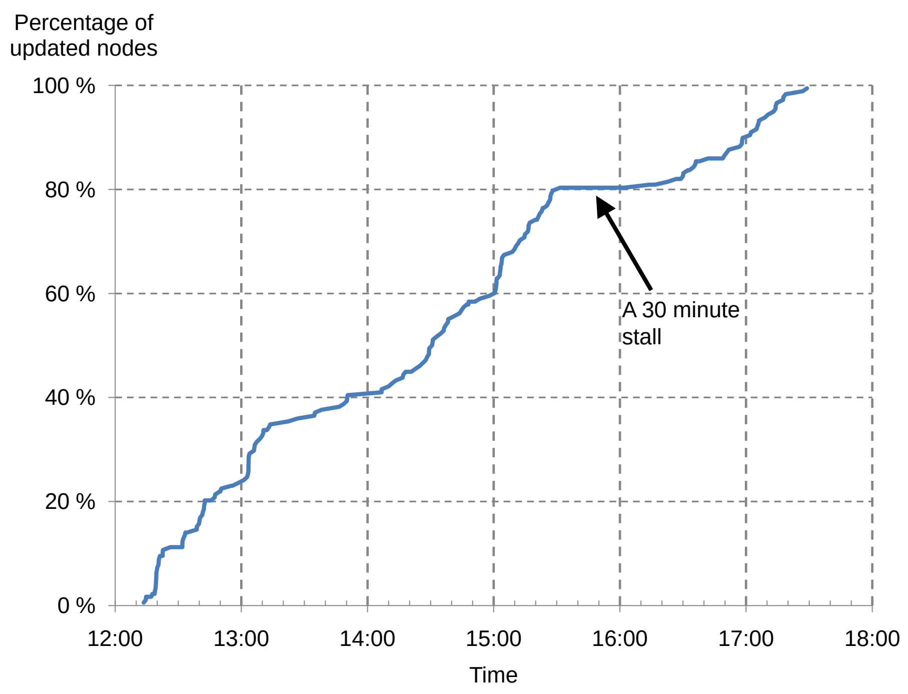


Figure 7.7: Percentage of updated nodes as a function of time.

Figure 7.8: Node being configured wirelessly at the manufacturing site with the prototype firmware management user interface and the single-hop parameter transfer protocol.

The user interface reports whether or not the node successfully received the parameters. Therefore, the person operating the user interface immediately knows if there has been a manufacturing fault in the node. Although this approach does not test the equipped sensors, it reliably tests the power generation subsystem, the microcontroller and the radio of the node. Since the wireless configuring of nodes was started, not a single faulty TUTWSN node has been shipped from the manufacturer.

8. CONCLUSIONS

This thesis presents the design, the implementation and the experimental measurements of firmware management for WSNs. It is designed according to the analyzed requirements of manufacturing and deployments of WSNs and to the inherent restrictions of low-energy, resource-constrained WSNs.

Firmware management was designed and implemented in a state of the art WSN architecture. Firmware management contains three server side components and three node side components. The firmware management user interface, the Autoconfigurator service and the database storage for firmware images and parameters form the server side components while the firmware parameter transfer protocols, the firmware image transfer protocol and the bootloader form the node side components. Firmware management allows remotely disseminating firmware images from a database into deployed WSNs without requiring physical access. Firmware management also allows dynamically altering the parameters of individual nodes, for example, for moving nodes from one WSN to another.

The node side firmware management components have been implemented and tested using TUTWSN node platforms, with an 8-bit 2 MIPS Microchip PIC18LF8722 microcontroller and a 2.4 GHz Nordic Semiconductors nRF24L01 radio. The server side components that are required for full end-to-end support for firmware management have not yet been implemented completely. Thus, the evaluation effort of firmware management has been directed at evaluating the performance of the node side firmware management components in real WSNs. The results have shown that the implemented components have fulfilled the requirements for firmware management.

Firmware management has shown that it can significantly ease the management of large scale WSNs and that it can be used effectively in manufacturing of nodes to increase production yields. Based on experimental measurements, updating a node using the designed protocols takes less than 90 seconds, while a large scale WSN of 268 nodes can be updated in five hours. Although firmware management was designed and implemented for TUTWSN, the design of firmware management is applicable as well to other modern WSN architectures, such as IEEE 802.15.4 based WSNs.

Future work on firmware management will concentrate primarily on two tasks.

The first task will be on firmware image transfers and finding a balance between increased propagation speed, higher reliability and lower energy usage. More specifically, the goal is to further explore the design of firmware image advertisement protocols and enhance the current advertisement protocol to account for radio link reliability when choosing a source for image acquisition. The second task is to further develop the use of firmware management in the manufacturing of nodes. The TUTWSN protocol stack will be modified to support more extensive self-testing and reporting on nodes and their external sensors.

REFERENCES

[1] I.F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Communications Magazine, IEEE*, 40(8):102–114, Aug 2002.

[2] Infrared Data Association. IrDA website. [Online]. Available: `http://www.irda.org/`, 2009. [Accessed: Jan. 19, 2010].

[3] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. Multi-level software reconfiguration for sensor networks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 112–121, New York, NY, USA, 2006. ACM.

[4] Yunxia Chen, Student Member, and Qing Zhao. On the lifetime of wireless sensor networks. *IEEE Commun. Lett*, 9:976–978, 2004.

[5] Crossbow Technology, Inc. MICA2 Wireless Measurement System. [Online]. Available: `http://www.xbow.com/`, March 2007. [Accessed: Apr. 7, 2010].

[6] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462, Nov. 2004.

[7] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[8] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[9] Ana-Belén García-Hernando, José-Fernán Martínez-Ortega, Juan-Manuel López-Navarro, Aggeliki Prayati, and Luis Redondo-López. *Problem Solving for Wireless Sensor Networks.* Springer Publishing Company, Incorporated, London, 2008.

[10] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on*

Programming language design and implementation, pages 1–11, New York, NY, USA, 2003. ACM.

[11] Bluetooth Special Interest Group. Bluetooth Specification Document. [Online]. Available: `http://www.bluetooth.com/`, 2009. [Accessed: Jan. 19, 2010].

[12] DACI Research Group. TUTWSN - Wireless Sensor Network. [Online]. Available: `http://www.tkt.cs.tut.fi/research/daci/ra_tutwsn_overview.html`, September 2009. [Accessed: Nov. 12, 2009].

[13] Jukka Haaramo. Sensoriverkon toimitusprosessi. Master's thesis, Tampere University of Technology, 2010.

[14] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[15] Jonathan W. Hui. Deluge 2.0 - tinyos network programming. [Online]. Available: `http://www.cs.berkeley.edu/~jwhui/deluge/deluge-manual.pdf`, July 2005. [Accessed: Nov. 13, 2009].

[16] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.

[17] International Organization for Standardization. Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. [Online]. Available: `http://www.iso.org/`, 2010. [Accessed: Feb. 13, 2010].

[18] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 96–107, New York, NY, USA, 2002. ACM.

[19] J.K. Juntunen, M. Kuorilehto, M. Kohvakka, V.A. Kaseva, M. Hännikäinen, and T.D. Hämäläinen. WSN API: Application programming interface for wireless sensor networks. In *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*, pages 1–5, Sept. 2006.

[20] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for “smart dust”. In *MobiCom ’99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278, New York, NY, USA, 1999. ACM.

[21] Mikko Kohvakka. *Medium Access Control and Hardware Prototype Designs for Low-Energy Wireless Sensor Networks*. PhD thesis, Tampere University of Technology (TUT) (Finland), 2009.

[22] Mikko Kohvakka, Marko Hännikäinen, and T.D. Hämäläinen. Wireless sensor prototype platform. In *Industrial Electronics Society, 2003. IECON ’03. The 29th Annual Conference of the IEEE*, volume 2, pages 1499–1504 Vol.2, Nov. 2003.

[23] Bilgin Kosucu, Kerem Irgan, Gurhan Kucuk, and Sebnem Baydere. FireSenseTB: A Wireless Sensor Networks Testbed for Forest Fire Detection. In *IWCMC ’09: Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing*, pages 1173–1177, New York, NY, USA, 2009. ACM.

[24] Patrick Kuckertz, Junaid Ansari, Janne Riihijarvi, and Petri Mahonen. Sniper fire localization using wireless sensor networks and genetic algorithm based data fusion. In *Military Communications Conference, 2007. MILCOM 2007. IEEE*, pages 1–8, Oct. 2007.

[25] Mauri Kuorilehto, Mikko Kohvakka, Jukka Suhonen, Panu Hämäläinen, Marko Hännikäinen, and Timo D. Hämäläinen. *Ultra-Low Energy Wireless Sensor Networks In Practice: Theory, realization and deployment*. Wiley Publishing, Chichester, 2007.

[26] Philip Levis and David Culler. Maté : A Tiny Virtual Machine for Sensor Networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, December 2002.

[27] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA ’02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[28] Kuorilehto Mauri, Hännikäinen Marko, and Hämäläinen Timo D. A survey of application distribution in wireless sensor networks. *EURASIP Journal on Wireless Communications and Networking*, 5:774–788, 2005.

[29] Microchip Technology, Inc. PIC18F8722 Product Page. [Online]. Available: `http://www.microchip.com/`, February 2008. [Accessed: Dec. 7, 2009].

[30] Microchip Technology, Inc. MPLAB C Compiler for PIC18 MCUs. [Online]. Available: http://www.microchip.com/, 2009. [Accessed: Dec. 13, 2009].

[31] Microchip Technology, Inc. PIC18F67J60 Product Page. [Online]. Available: http://www.microchip.com/, October 2009. [Accessed: Mar. 25, 2010].

[32] Nordic Semiconductors. nRF24L01 Product Specification. [Online]. Available: http://www.nordicsemi.com/, July 2007. [Accessed: Dec. 7, 2009].

[33] Nordic Semiconductors. nR905 Product Specification. [Online]. Available: http://www.nordicsemi.com/, April 2008. [Accessed: Feb. 18, 2010].

[34] The ZigBee Standards Organization. Zigbee specification, version r17. [Online]. Available: http://www.zigbee.org, January 2007. [Accessed: Feb. 16, 2010].

[35] F.J. Pierce and T.V. Elliott. Regional and on-farm wireless sensor networks for agricultural systems in eastern washington. *Computers and Electronics in Agriculture*, 61(1):32 – 43, 2008. Emerging Technologies For Real-time and Integrated Agriculture Decisions.

[36] Maxim Integrated Products. DS620 Low-Voltage, ±0.5°C Accuracy Digital Thermometer and Thermostat. [Online]. Available: http://www.maxim-ic.com, August 2004. [Accessed: Feb. 17, 2010].

[37] L. B. Ruiz, J. M. Nogueira, and A. A. F. Loureiro. MANNA: A Management Architecture for Wireless Sensor Networks. *Communications Magazine, IEEE*, 41(2):116–125, 2003.

[38] Jaspal S. Sandhu. Wireless sensor networks for commercial lighting control: Decision making with multi-agent systems. In *In AAAI Workshop on Sensor Networks*, pages 131–140, 2004.

[39] Silvia Santini, Benedikt Ostermaier, and Andrea Vitaletti. First experiences using wireless sensor networks for noise pollution monitoring. In *REALWSN '08: Proceedings of the workshop on Real-world wireless sensor networks*, pages 61–65, New York, NY, USA, 2008. ACM.

[40] Jukka Suhonen, Mauri Kuorilehto, Marko Hännikäinen, and T.D. Hämäläinen. Cost-aware dynamic routing protocol for wireless sensor networks - design and prototype experiments. In *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*, pages 1–5, Sept. 2006.

[41] Avago Technologies. APDS-9002 Miniature Surface-Mount Ambient Light Photo Sensor. [Online]. Available: http://www.avagotech.com, June 2006. [Accessed: Feb. 17, 2010].

[42] Texas Instruments. CC1101 . [Online]. Available: http://www.ti.com/, January 2010. [Accessed: Mar. 12, 2010].

[43] The Institute of Electrical and Electronics Engineers, Inc. IEEE Standard for a Smart Transducer Interface for Sensors and Actuators. [Online]. Available: http://ieee1451.nist.gov/, May 2002. [Accessed: Apr. 8, 2010].

[44] The Institute of Electrical and Electronics Engineers, Inc. IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs). [Online]. Available: http://standards.ieee.org/getieee802/download/802.15.4a-2007.pdf, August 2008. [Accessed: Nov. 12, 2009].

[45] The Institute of Electrical and Electronics Engineers, Inc. IEEE 802.15 Working Group for WPAN. [Online]. Available: http://www.ieee802.org/15/, 2009. [Accessed: Jan. 19, 2010].

[46] The Internet Engineering Task Force. A Simple Network Management Protocol (SNMP), RFC 1157. [Online]. Available: http://tools.ietf.org/html/rfc1157, May 1990. [Accessed: Apr. 12, 2010].

[47] VTI Technologies. SCA3000-E01 3-Axis ultra low power accelerometer with digital SPI interface. [Online]. Available: http://www.vti.fi, September 2009. [Accessed: Mar. 12, 2010].

[48] Qiang Wang, Yaoyao Zhu, and Liang Cheng. Reprogramming wireless sensor networks: challenges and approaches. *Network, IEEE*, 20(3):48–55, May-June 2006.

[49] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, September 1991.

[50] Geoff Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *EWSN'05: Proceedings of the Second European Workshop on Wireless Sensor Networks*, 2005.

[51] Mote in-network programming user reference. [Online]. Available: http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf, March 2003. [Accessed: Nov. 11, 2009].

[52] Haiqing Yang and Yong He. Wireless sensor network for orchard soil and climate monitoring. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 1, pages 58–62, 31 2009-April 2 2009.