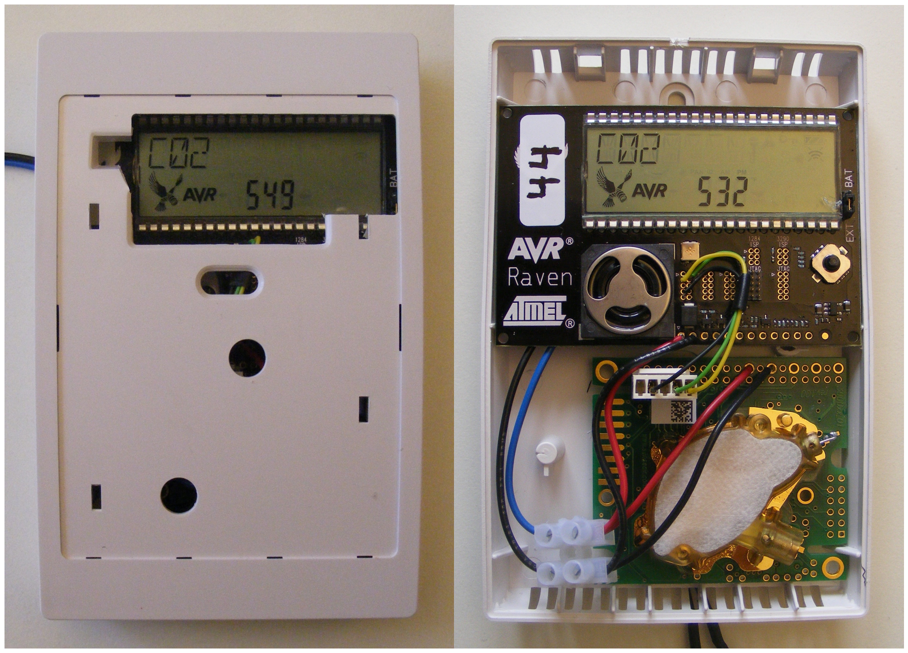
CHALMERS



Wireless Sensor Networks
an implementation using 6LoWPAN

Master of Science thesis in the Integrated Electronic System Design programme

NISVET JUSIC
THILAK RATHINAVELU

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, May 2011

Wireless Sensor Networks
an implementation using 6LoWPAN

Nisvet Jusic
Thilak Rathinavelu

Department of Microtechnology and Nanoscience
CHALMERS, University of Technology
Göteborg, Sweden, May 2011

Wireless Sensor Networks
an implementation using 6LoWPAN

NISVET JUSIC, nisvet.84@gmail.com
THILAK RATHINAVELU, thilrat@gmail.com

Examiner: KJELL JEPPSON

Chalmers University of Technology
University of Gothenburg
Department of Microtechnology and Nanoscience
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Microtechnology and Nanoscience
Göteborg, Sweden, May 2011

Abstract

The past decade has seen a lot of research in the area of Wireless Sensor Networks. The fruits of this research was a large number of proprietary network technologies, the most successful of these being ZigBee. However as the number of technologies have grown, so had the need for standardization. Added to this was the requirement of supporting large scale networks, most of the existing technologies did not scale well with size. The result was the IETF 6LoWPAN working group. Their standards allow IPv6 to be operated on low-power networks, IPv6 being both a proven standard and reliable in large scale networking.

A simple but complete network, where nodes deliver data to a central server, will be implemented and examined by this thesis, using Contiki. Developed by SICS, Contiki is one of the leading operating systems designed for low-power wireless devices. It carries SICSlowpan as its own implementation of the 6LoWPAN standard.

Contiki proved to be a very advanced and capable operating system. Developing in Contiki is significantly simplified by Cooja. Cooja is a network simulator where code can be tested before it is deployed to the target hardware, on which it is often much harder to debug. Another notable advantage is the programming language, code for Contiki is written in standard C. Some of the comparable operating system require code to be written in a custom dialect or language. Finally it is presently the only operating system supporting the TCP protocol, which is required by some implementations in this thesis.

Sammanfattning

Under det senaste årtiondet har det forskats mycket i trådlösa sensornätverk. Detta resulterade i en mängd proprietära nätverksteknologier, den mest framgångsrika av dessa är ZigBee. Men med ett ökande antal teknologier kom också ett behov av standardisering. Dessutom har på senare tid också kravet att stödja storskaliga nätverk tillkommit. Svaret på detta var IETF arbetsgruppen 6LoWPAN. Deras standarder tillåter IPv6 att användas på strömsnåla nätverk, IPv6 som är en välkänd standard för storskaliga nätverk.

Ett enkelt men fullständigt nätverk, där noderna levererar data till en central server, kommer implementeras i denna tes. Operativ systemet Contiki används som grundsten i nodmjukvaran. Utvecklat av SICS, Contiki är en av de ledande operativsystemen för strömsnåla enheter. Contiki skeppas med SICSlowpan, vilket är SICS implementering av 6LoWPAN.

Contiki visade sig vara ett avancerat och kapabelt operativ system. Att utveckla för Contiki förenklas dessutom genom användadet av Cooja. Cooja är en nätverkssimulator utvecklat av SICS. Kod kan simuleras och debuggas innan den laddas upp till hårdvara, på vilken det ofta är mycket svårare att felsöka. En annan fördel med Contiki är programmeringsspråket, applikationskod för Contiki är skriven i C. Detta till skillnad från vissa av konkurrenterna, där koden måste skrivas i en speciell dialekt eller språk. Slutligen är Contiki för tillfället det enda operativ systemet som stödjer TCP protokollet, vilket krävs för vissa av implementeringarna i denna tes.

Acknowledgements

We would like to thank SenseAir for the opportunity to conduct this thesis, especially Joakim Enerud and Henrik Rödjegård for their support and guidance. We would also like to thank Kjell Jeppsson at Chalmers University of Technology for his help in writing this report.

Finally we would like to thank Carl Bengtsson for putting us in touch with SenseAir and making this thesis possible.

Table of Content

List of Figures

List of Tables

Abbreviations

6LoWPAN	IPv6 over Low Power Area Networks
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BLIP	Berkeley Low-power IP
BMAC	Berkeley MAC
CRC	Cyclic Redundancy Check
CRUD	Create Read Update Delete
CSMA	Carrier Sense Multiple Access
CSS	Cascading Style Sheets
DC	Direct Current
EEPROM	Electrically Erasable Programmable Read-Only Memory
EJB	Enterprise JavaBeans
EL	Expression Language
FIFO	First In, First Out
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISM	Industrial, Scientific and Medical
Java EE	Java Enterprise Edition
Java SE	Java Standard Edition
JDBC	Java Database Connectivity
JSF	Java Server Faces
JPA	Java Persistence API
JPQL	Java Persistence Query Language
LAN	Local Area network
LCD	Liquid Crystal Display
LPL	Low-Power Listening
LR-WPAN	Low-Rate Wireless Personal Area Network
MAC	Medium Access Control
MCU	Microcontroller Unit
MIPS	Million Instructions Per Second
MTU	Maximum Transmission Unit
NIC	Network Interface Card
OSI	Open System Interconnection
PC	Personal Computer
PCB	Printed Circuit Board
PDU	Protocol Data Unit
PHP	PHP: Hypertext Preprocessor
PLC	Programmable Logic Controller

POJO	Plain Old Java Object
ppmv	parts per million by volume
RADVD	Linux IPv6 Router Advertisement Daemon
RAM	Random-Access Memory
REST	Representational State Transfer
ROLL	Routing over Low-power and Lossy networks
ROM	Read Only Memory
RPL	IPv6 Routing Protocol for Low-Power and lossy networks
RS-232	Recommended Standard 232
RTOS	Real-Time Operating System
RTU	Remote Terminal Unit
SICS	Swedish Institute of Computer Science
SLAAC	Stateless Address Autoconfiguration
SRAM	Static Random-Access Memory
SQL	Standard Query Language
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
USART	Universal Synchronous-Asynchronous Receiver/Transmitter
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Network
WWW	World Wide Web

1 Introduction

SenseAir AB is a world leader in infra-red gas sensor manufacturing [1]. They mainly develop and produce reliable low-power carbon dioxide sensors. Currently there exists a wired network infrastructure, though it is very costly and rigid to setup. A wireless network would be much more flexible as well as cheaper to install. SenseAir would therefore like to produce a prototype wireless infrastructure.

A previous Master thesis at SenseAir studied different radio network technologies, which resulted in a suggestion to use ZigBee [2][3]. ZigBee being the predominant network technology for low-power wireless networks. Lately a new standard, 6LoWPAN, has started to gain momentum [4]. SenseAir would like to investigate the possibilities to use this network technology for networking their sensors.

1.1 Background

Sensors, much like most other electronic components, are becoming smaller and less power consuming with each year. In then makes more and more sense to allow these device to become wireless, about a decade ago this effort started to draw a lot of research. The vision then was s*mart dust*. Sensor nodes of such small size that they were comparable with dust particles.

The current trend is to integrate Wireless Sensor Networks (WSN) into the *Internet of Things* [5]. The Internet is usually divided into the core and the fringe Internet. The core Internet consists of the backbone routers and servers. The fringe Internet consists of regular Personal Computers (PCs). The number of hosts in the core Internet are counted in the millions, whereas the number of hosts in the fringe Internet are counted in the billions. The Internet of Things is a new emerging sector, which consists of *smart objects*. Smart objects are generally embedded devices, which require machine to machine communication. The number of hosts in the Internet of Things is expected to reach trillions. Here lies the main reason for choosing 6LoWPAN, to support the large number of hosts IPv6 is required, which is provided by 6LoWPAN [6].

1.2 Purpose

The purpose is to evaluate 6LoWPAN as a viable alternative for a SenseAir WSN implementation. The system has two major tasks. The first task, data collection, is to fetch the sensor data from each sensor and aggregate it at a specific location. The second task, data presentation, is to present the collected sensor data for the user in a user friendly form.

1.3 Delimitations

First delimitation is one-way communication between nodes and server. Only nodes will initiate communication, the server's task is to collect the data from each node. This decision was made to simplify the node functionality.

Second delimitation is power concerns. No power-saving features are implemented, the nodes are always powered. This delimitation was set by SenseAir. The main reason is that the sensor itself consumes much more energy than the Microcontroller Unit (MCU) and radio.

Third delimitation is security. There are no readily available technologies for securing low-power wireless networks. This effort might require a thesis of its own.

1.4 Related work

The research area of Wireless Sensor Networks have spawned a large number of projects, each emphasizing different functional aspects.

Some are designed specifically for long battery life, such as the Wireless Sensor Systems project at Mid Sweden University in Sundsvall [7]. The environmental monitoring subproject aims to create a WSN with extremely long battery lifetime. To achieve this everything from hardware to network protocols are custom made. In contrast to this thesis where the main focus is to use existing standards.

An other project more closely related to this thesis is the Bosmart project at Acreo [8]. It is suspected that during wintertime a lot of excessive venting is done by tenants, to improve air quality in the apartment. This increases the cost of heating for the landlords, to investigate and survey this behaviour the Bosmart project will create a WSN sensing CO_2 in each apartment. An rapid decrease in CO_2 concentration will indicate excessive venting. Data is gathered from all apartments and is presented through a web interface.

2 Technologies

2.1 Wireless Sensor Networks

What differentiates WSN from other wireless networks is the sensing and controlling capability [9]. Wireless sensor network are usually deployed in inaccessible environments to monitor conditions or gather sensor data, that can be used for further analysis. Sensor networks are built-up of several small devices, each device connected to a sensor. These devices are required to be very small, power-efficient and cost-effective, but at the same time have sensing, computation, storage and communication capabilities. This has a big impact on the resource constraints, which differs dramatically from a typical wired network.

Wireless sensor networks are commonly self-organized, self-configurable and should last for years without maintenance. The latter being a very difficult demand where power consumption is the critical factor. A lot of research is done regarding power-efficiency for small resource-constrained devices, of particular interest are improvements that reduce power consumptions in radio transmissions.

Memory on each device is not large enough to store all gathered sensor data. That means collected data has to be transported through the wireless network and be stored at a specified location. Therefore the communication capability is needed. Aside from a sensor each single device is required to contain its own microcontroller, memory, radio transceiver and power supply.

2.2 Network Protocols

2.2.1 IEEE 802.15.4

Institute of Electrical and Electronics Engineers (IEEE) has developed standards for different wireless technologies, such as IEEE 802.11 Wireless Local Area Network (WLAN also known as *WI-FI*), IEEE 802.15.1 Bluetooth and IEEE 802.15.4 Low-Rate Wireless Personal Area Network (LR-WPAN) [10]. The latter, IEEE 802.15.4, is the most interesting, where low-power and low-bandwidth is taken into account [11]. The standard is defined at the Media Access Control (MAC) layer and physical layer, specified in the seven-layer Open System Interconnection (OSI) model, see figure 1 [12]. MAC is a sublayer of the data link layer and provides addressing and channel access control mechanisms. These mechanisms makes it possible for communication between nodes within a network. IEEE 802.15.4 focuses on low-speed and low-cost communication between devices, which results in a largest supported frame size of 127 bytes. The standard provides a data transfer rate of up to 250 kb/s at 2.4 GHz frequency.

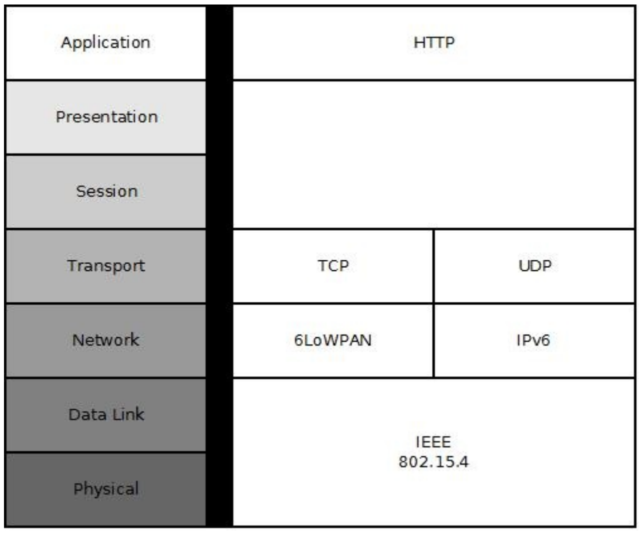


Figure 1: Network protocols

2.2.2 6LoWPAN

The IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) standard defines an adaption layer, which allows IPv6 packets to be transmitted over low-power networks [4][13]. The standard was developed in 2007 by the Internet Engineering Task Force (IETF) 6LoWPAN working group. The IETF is a organisation with the aim to improve the Internet by providing technical documents that will aid users and designers of Internet. The standard is positioned in the network layer of the OSI model.

The Maximum Transmission Unit (MTU) of IPv6 packets. The MTU refers to the size of the largest Protocol Data Unit (PDU) that a network protocol can transmit. The default MTU of an IPv6 packet is 1280 bytes, which means an IPv6 packet can be as large as 1280 bytes. However, most of the low-power networks have a much smaller frame size, 127 bytes for the IEEE 802.15.4, see section 2.2.1, and thus can not support such a large packet. To cope with these large IPv6 packets 6LoWPAN offers fragmentation. The large IPv6 packet will be split up into smaller frames that can be transmitted over low-power networks. This is the main reason why 6LoWPAN is required.

The second functionality of 6LoWPAN is header compression. Low-power networks often have very low bandwidth, thus sending the entire IPv6-address is very wasteful. In a single network with the same prefix, only the last part of the IPv6 address is needed to identify a host. One feature of header compression is to remove the unnecessary part of the IPv6 address that is not needed and make the header smaller.

2.2.3 ZigBee

ZigBee was created in 2003 by a group of companies, known as ZigBee Alliance [3]. ZigBee Alliance work together to make low-cost and low-power wireless technology available in different markets. ZigBee is the dominant low-power network technology.

ZigBee is a protocol specification that builds upon the IEEE 802.15.4 standard. ZigBee adds some extra features, such as network and security layers and application protocol profiles. The standard is designed for LR-WPANs and operates on the worldwide Industrial, Scientific and Medical (ISM) 2.4 GHz radio band.

A ZigBee network is composed of three different type of devices; ZigBee Coordinator, ZigBee Router and ZigBee End Device. The coordinator is the most intelligence device in the network and might also bridge to other networks. There is only one coordinator in each ZigBee network. The coordinator is tasked to control the security and formation of the network. The Router allows the network to scale in size by acting as a traditional router and forward data from other devices. But the router also contains the application code which performs the specific sensing or controlling tasks within a network. The end device contains only the application code and can not route any data. This means the end devices have less responsibility and thereby can be asleep a significant amount of the time, thus increasing the battery life time for end devices. Though, coordinator and router devices has to be powered all the time to fulfill their functionality.

2.2.4 Internet Protocol

The original version of IP was created in the early 1970s by Vint Cerf and Bob Kahn [14]. It was designed in conjunction with the Transmission Control Protocol (TCP), see section 2.2.5 [15] [16]. Together they form what is now known as the TCP/IP stack, see figure 1.

IP is a so called connection less protocol, it purpose is to provide a *best effort* transmission service. Connection less means there is no setup time before transmitting data. IP is best effort in the sense that there is no guarantee that data will arrive safely at the destination. The result being hardware requirements for IP are quite low. Routers supporting IP only need to forward incoming packets to a matching outgoing link. The intelligence instead is placed in the end nodes, where the upper layers of the stack exist. These layers have the responsibility of dealing with the inherent unreliability issues of IP.

The majority of computers connected to the Internet today use Internet Protocol version 4 (IPv4). IPv4 was first drafted in 1981, and is the first and only IP version to enjoy widespread adoption. Unfortunately the requirements of the current Internet have started to outgrow the specifications of IPv4, namely the address space. IPv4 features a 32-bit address which limits the number of addresses to roughly 4.3 billion. With the rapid growth of the Internet the IPv4 address space is estimated to be depleted by march 2012 [5]. Although Network Address Translation (NAT) has alleviated the situation it does not solve the problem. In addition, devices providing NAT must hold network states, breaking the end-to-end principle in the process[6].

Internet Protocol version 6 (IPv6) is next version of the Internet Protocol (IP) [6]. With an address length of 128-bits its address space covers 3.4 * 10^38 entries, it effectively solves the address space issue for the foreseeable future. IPv6 not only brings a larger address space, but also a variety of new technologies.

Stateless Address Autoconfiguration (SLAAC) is one new technology used in this project. SLAAC allows nodes to automatically configure themselves using router discovery messages. A typical IPv6 address consists of a 64-bit network prefix and a 64-bit host address. The network prefix is supplied by the router, the host address is generated from the MAC address of the host. A sample IPv6 address generated from SLAAC is shown in table X.

	Network prefix	Host address
IPv6 address	aaab:0:0:0:	1122:3344:5566:7788
MAC address		11-22-33-44-55-66-77-88

Table 1: IPv6 address

2.2.5 Transmission Control Protocol

Transmission Control Protocol (TCP) is a transport layer connection oriented protocol designed to address the shortcomings of IP [6]. Connection oriented means a connection must first be established by the protocol. A three-way handshake is used to set up the connection, after which data can be transmitted. TCP offers reliability through retransmissions and acknowledgments. Each TCP segment sent must be acknowledged by the receiving party. Should a segment fail to be acknowledged within a certain time frame, it will be retransmitted.

2.2.6 User Datagram Protocol

User Datagram Protocol (UDP) is a transport layer connection less protocol. UDP, like IP, is a best effort protocol, as such it offers no guarantee of packet delivery [17]. This allows it to be even more lightweight than TCP. UDP adds checksumming and multiplexing by port over the functionality IP provides.

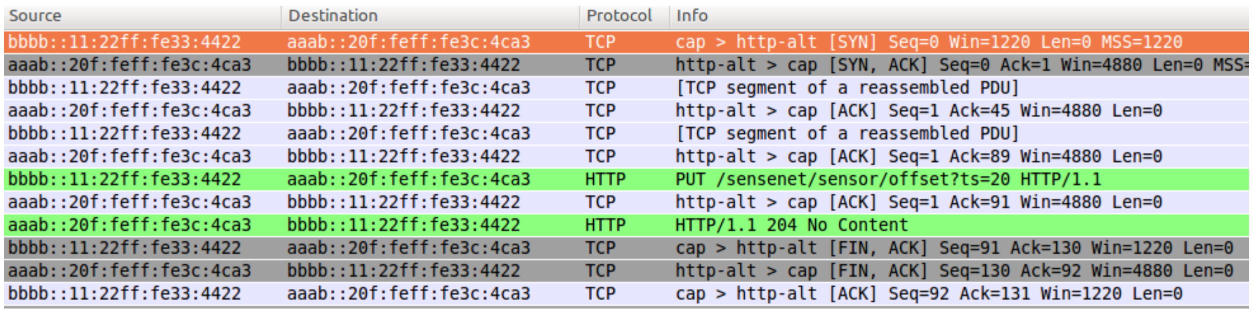
2.2.7 HTTP & RESTful services

The Hypertext Transfer Protocol (HTTP) is an application layer protocol designed to transfer Hypertext Markup Language (HTML) pages from a web server to a client [18]. It lays the foundation for the World Wide Web (WWW).

HTTP defines a number of action clients can take, and the respective actions servers should take. Client actions are called requests and server actions responses. Requests are made on resources on the server identified by an Uniform Resource Identifier (URI). URI addresses always start with *http://* or *https://*, the latter indicating a secure connection. The format of a request message is as follows:

- Request: indicating type of request and location of resource by URI
- Headers: meta information regarding the message
- Empty line
- Message content

Table 2 lists the most common requests a client can perform. Headers can contain information about the message and the communicating parties. Common headers include date, server or client names. Only one header field is mandatory in HTTP 1.1, the *host* field, which is the latest version of HTTP. The host field contains the domain name of the server, it is used to differentiate between multiple host names on a single IP address. A sample HTTP communication is shown in figure 2.



```
Source                   Destination              Protocol  Info
bbbb::11:22ff:fe33:4422  aaab::20f:feff:fe3c:4ca3 TCP       cap > http-alt [SYN] Seq=0 Win=1220 Len=0 MSS=1220
aaab::20f:feff:fe3c:4ca3 bbbb::11:22ff:fe33:4422  TCP       http-alt > cap [SYN, ACK] Seq=0 Ack=1 Win=4880 Len=0 MSS=
bbbb::11:22ff:fe33:4422  aaab::20f:feff:fe3c:4ca3 TCP       [TCP segment of a reassembled PDU]
aaab::20f:feff:fe3c:4ca3 bbbb::11:22ff:fe33:4422  TCP       http-alt > cap [ACK] Seq=1 Ack=45 Win=4880 Len=0
bbbb::11:22ff:fe33:4422  aaab::20f:feff:fe3c:4ca3 TCP       [TCP segment of a reassembled PDU]
aaab::20f:feff:fe3c:4ca3 bbbb::11:22ff:fe33:4422  TCP       http-alt > cap [ACK] Seq=1 Ack=89 Win=4880 Len=0
bbbb::11:22ff:fe33:4422  aaab::20f:feff:fe3c:4ca3 HTTP      PUT /sensenet/sensor/offset?ts=20 HTTP/1.1
aaab::20f:feff:fe3c:4ca3 bbbb::11:22ff:fe33:4422  TCP       http-alt > cap [ACK] Seq=1 Ack=91 Win=4880 Len=0
aaab::20f:feff:fe3c:4ca3 bbbb::11:22ff:fe33:4422  HTTP      HTTP/1.1 204 No Content
bbbb::11:22ff:fe33:4422  aaab::20f:feff:fe3c:4ca3 TCP       cap > http-alt [FIN, ACK] Seq=91 Ack=130 Win=1220 Len=0
aaab::20f:feff:fe3c:4ca3 bbbb::11:22ff:fe33:4422  TCP       http-alt > cap [FIN, ACK] Seq=130 Ack=92 Win=4880 Len=0
bbbb::11:22ff:fe33:4422  aaab::20f:feff:fe3c:4ca3 TCP       cap > http-alt [ACK] Seq=92 Ack=131 Win=1220 Len=0
```

Figure 2: HTTP traffic

Representational State Transfer (REST) is an design pattern for machine to machine communication [19].

A HTTP implemented RESTful service commonly uses the first four requests listed in table 2. These four requests are usually mapped to a Create Read Update Delete (CRUD) pattern, also indicated in table 2. The CRUD pattern refers to operations on the database. A GET request would fetch a row from the database and present it to the client, a POST request would typically create a new row in the database.

GET(READ)	Fetches the content of a resource
POST(CREATE)	Posts a message to the resource
PUT(UPDATE)	Puts a message to the resource, functionality similar to POST
DELETE(DELETE)	Deletes a resource
HEAD	Fetches only the headers of a resource

Table 2: Common HTTP requests

2.2.8 Modbus

The Modbus protocol was developed by Modicon in 1979 [20]. Modbus is a de facto industry standard serial communication protocols for devices connected on different type of buses.

The Modbus Application protocol is positioned at the application layer of the OSI model [21]. The application protocol defines a Protocol Data Unit (PDU) which allows client/server communication between devices. A request is performed by building a PDU, which consists of function code and additional information regarding the request, *data field*, see figure 3. The reply will also be a PDU containing function code and additional data response information.

Modbus Serial Line protocol is a master/slave protocol positioned at the data link layer of the OSI model [22]. The device requesting the information is called the master and devices responding to the request are called slaves. At any time only one master device may be connected to the Modbus serial line, while up to 247 slaves are supported. A request is always initiated by the master device, which can not handle more than one request at a time. Slave devices can not initiate requests and they will never transmit data without first receiving a request from the master device. Furthermore, the slaves can not communicate with each other.

Two transmission modes exist for serial lines, Remote Terminal Unit (RTU) mode and American Standard Code for Information Interchange (ASCII) mode. These defines the bit contents of message fields and how the information is decoded. The transmission mode should correspond for all devices on a Modbus serial line.

Modbus Serial Line protocol adds additional fields to the Modbus PDU, *address* and *error checking* field, see figure 3. The address filed specifies the slave address. In RTU mode error checking is performed by Cyclic Redundancy Check (CRC). CRC is an error detection code designed to detect accidental changes to raw data and is performed on every byte of the message content. The receiving device also calculates the CRC on the same message content. Should the calculated CRC not match against the acquired CRC from the sending device, an error will occur.

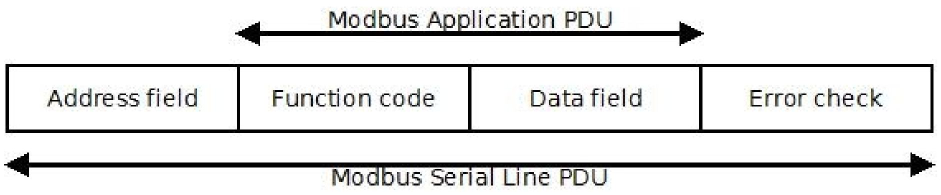


Figure 3: Modbus Application and Serial Line PDUs

3 System Overview

This section will give an brief introduction to the system, see figure 4 for an overview. Three base components can be found in the system: the node, the edge router and the server.

The node is the 6LoWPAN equivalent of a router. This means all nodes in the system are routers, hence all nodes will forward data within the wireless network. This is the foundation for the mesh network topology. Each MCU is connected to a sensor and will govern the data collection from the sensor and transmit the data to the data collection server.

The edge router is the up-link device. Through this the 6LoWPAN network is connected to the outside. Its only task is to receive data from the 6LoWPAN network and forward it to the Local Area Network (LAN), and in extension the Internet.

The server has two main tasks, data collection and data presentation. Data collection is performed through a web service, the server will receive data transmitted by all the nodes and store them in a database. The data presentation is performed through a website, where a line graph will show transient data.

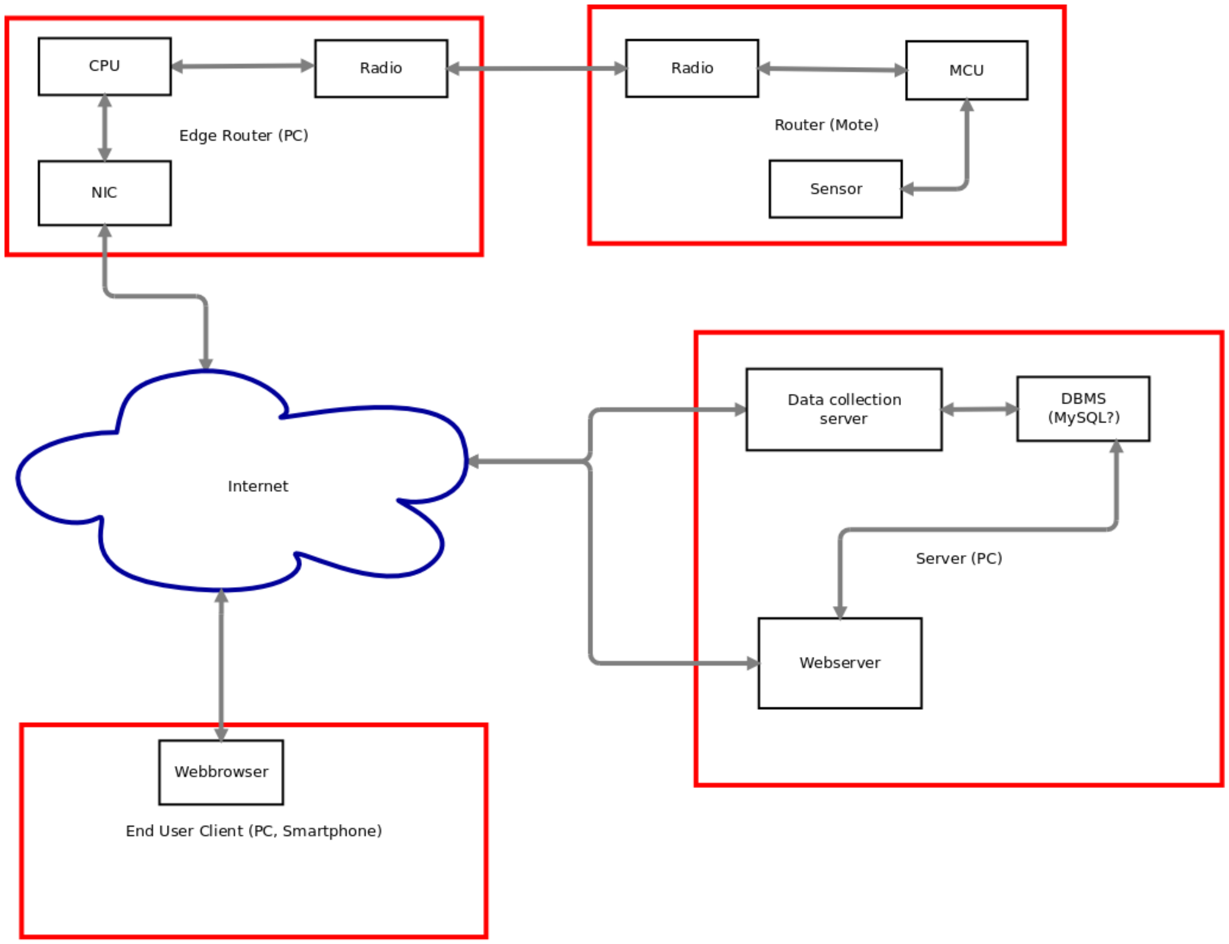


Figure 4: Overview of the system

4 Node

Nodes are sensor devices that build up the wireless sensor network. These collect sensor data and send it to the server. All nodes have router functionality, which allows them to pass on data from other nodes within the network. A node without enclosure is shown in figure 5. A node consists of an AVR Raven connected to a K30 sensor, provided by SenseAir. The Raven boards are running Contiki operating system.

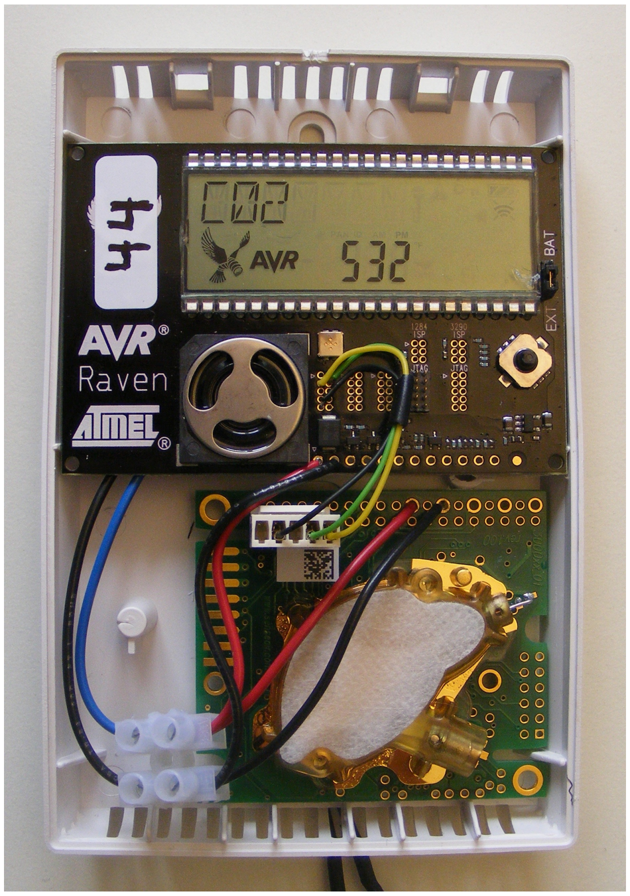


Figure 5: Node without enclosure

4.1 Node Operating System

For this project the task was to implement a 6LoWPAN wireless sensor network. With that in mind, abstracting away most of the low-level aspects proved to be a necessity. These low-level aspects can be provided by an operating system. Furthermore an operating system can offer features like power management, memory management, scheduling polices, multitasking and networking capability. However, traditional operating systems are not designed for devices with stringent resource-constraints such as power consumption, available bandwidth and memory.

There are a number of operating systems that are specifically designed for low-power and resource-constrained devices. They are less complex than general operating systems and most of them require merely a few kilobytes of Read Only Memory (ROM) and a few hundred bytes of Random Access Memory (RAM). The lightweight footprint make these operating systems suitable for memory-constrained devices. However not all operating systems contain an IP communication stack, especially a 6LoWPAN implementation which is the primary functionality needed for this project. The requirement of 6LoWPAN made it possible to narrow down the collection of operating systems to three, Contiki, TinyOS and FreeRTOS.

4.1.1 Contiki

Contiki is a small, open-source, event-driven operating system designed for low-power and memory-constrained devices [23]. The operating system was initially designed by Adam Dunkles in the Networked Embedded Systems group at the Swedish Institute of Computer Science (SICS), but has been further developed by a collaboration between developers from industry and academia. The memory allocated by the system can be configured to be as small as 40 kB of ROM and 2 kB of RAM.

A Contiki process consists of a single protothread running an infinite loop. Protothreads are lightweight stackless threads intended for very memory-constrained systems. They provide conditional blocking and thus also allow linear code execution. All protothreads in the system run on the same stack in contrast to traditional threads where each thread requires an own stack, hence the very small amount of memory allocation for each protothread. The stack is rewound every time a protothread blocks, which means the information contained by a protothread will be overwritten by other protothreads. That is why all variables used within a protothread has to be globally declared in order to be preserved.

One of the many features in Contiki is provided by the event-driven kernel. The kernel is non-preemptive, which means it will execute a process until that process blocks. Thus it is up to the programmer to schedule the processes within an application. Though, all events will be queued and handled by the kernel in a First In, First Out (FIFO) order. A blocking process will wait for an event to happen, meanwhile the kernel continues by executing other processes. When an event is triggered the kernel executes the corresponding process and passes associated event information.

The event-based kernel, protothreads and the blocking macros gives the user possibility to write programs with a sequential control flow.

Part of the Contiki operating system is a communication stack, uIPv6. The uIPv6 communication stack is also partly developed by SICS and is one of the world's smallest open-source TCP/IP stacks. SICSlowpan, which is Contikis implementation of 6LoWPAN, is included in uIPv6.

Forwarding and routing can be done at either the data link layer *mesh-under* or network layer *route-over* of the OSI model. Contiki implements the route-over routing techniques. This is enabled by ContikiRPL, which is an implementation of the RPL routing protocol developed by the Routing over Low-power and Lossy networks (ROLL) working group at IETF [24]. The protocol is specifically designed for IPv6 routing in low-power wireless sensor networks.

By implementing these different network standards and protocols designed for low-power wireless networks the power consumption is decreased in comparison to original networking protocols. To further decrease power consumption, Contiki implements an adaption MAC layer protocol, called X-MAC [25]. X-MAC is an adaptive short preamble, low-power MAC layer protocol designed for duty-cycled WSNs. Preamble is a signal used in WSNs to synchronize transmission timing between sender and receiver. Duty-cycle referrers to periodically cycling between awake and sleep mode. X-MAC uses an asynchronous Low-Power Listening (LPL) approach, also called preamble sampling, to link together a sender with a duty-cycling receiver. The sender transmits a preamble, which consists of many small packets, called preamble frames. These preamble frames has to be sent over a period that is at least as long as receiver's sleep period, in order to wake the receiver up. When the receiver wakes up, the preamble will be detected and the receiver will stay awake to perform the actual transmission. The short preamble and asynchronous LPL duty-cycling makes X-MAC more energy-efficient compared to other similar techniques that uses longer preamble and synchronous LPL.

Contiki also offers the programmer some type of power management. This feature is not provided implicitly, instead it is up to the programmer to code such behavior that will result in lower power consumption. One approach is to use less radio transmissions, which are very power inefficient. The data could instead be stored temporary within the node using Coffee, an flash-based file system provided by Contiki, and transmitted less often in larger chunks.

The Contiki and all the implementation is written in standard C, which makes it easy to understand and modify.

4.1.2 TinyOS

TinyOS is maybe one of the most known operating systems today designed for small embedded resource-constrained devices [26]. TinyOS started initially as a research project at the University of California Berkeley but has since been developed by a large group of academic and commercial developers and users, also called TinyOS Alliance. TinyOS is open-source and supports the 6LoWPAN standard. TinyOS's 6LoWPAN implementation is part of the Berkeley Low-power IP (BLIP) stack. BLIP is equivalent to Contiki's uIPv6 stack.

There are some similarities between Contiki and TinyOS, like the event-driven programming and RPL support. The TinyOS kernel works in a similar way as Contiki's event-based kernel. Tasks are run until completion and appropriate event handlers are signaled when an event happens. Tasks can be posted and will be scheduled by the kernel. Some of the RPL developers are also TinyOS developers and the RPL implementation for TinyOS is currently in progress.

TinyOS uses a combination of Berkeley MAC (BMAC) and X-MAC to achieve low-power communication. BMAC is another low-power MAC layer protocol designed for duty-cycled, low traffic WSN. BMAC is a Carrier Sense Multiple Access (CSMA) based technique, developed at the University of California at Berkeley. The adaption protocol exploit LPL to lower the power consumption, but in contrast to X-MAC uses an extended preamble.

TinyOS offers an Application Programming Interface (API) for power management, which can be used to manage both radio and processor. The programmer can allow the processor to sleep when there are no tasks to be run. But it is still up to the programmer to implement this feature in the application.

Application code is written in nesC programming language, which is a dialect of the standard C.

4.1.3 FreeRTOS

FreeRTOS is an open source Real-Time Operating System (RTOS) [27]. The operating system was originally designed by Richard Berry at Wittenstein High Integrity Systems in Bristol but has further been developed by the FreeRTOS team, a group of developers and users worldwide. FreeRTOS is based on a mini real-time kernel, which means that tasks are scheduled in real-time in contrast to Contiki and TinyOS.

FreeRTOS does not have a native 6LoWPAN implementation, in contrast to Contiki and TinyOS. However it does support NanoStack, which is a 6LoWPAN implementation developed by Sensinode [28]. The footprint of FreeRTOS is slightly larger than for Contiki and TinyOS.

FreeRTOS contains an implementation of the X-MAC.

The application code is written in C.

4.1.4 Operating system comparison

As shown in the table 3, all three operating systems include or supports an implementation of the 6LoWPAN standard. The communication stack implementations differentiates from each other, but this was not a critical issue. Table 4 shows a brief comparison of the three IP stacks, any deeper comparison between the stacks was not desirable. All three stacks supports UDP protocol, though Contiki's IP stack is the only one supporting TCP. In order to properly examine 6LoWPAN different implementations were created in this thesis. Due to the application protocols requiring TCP, this feature was important.

Operating system	**Contiki**	**TinyOS**	**FreeRTOS**
6LoWPAN implementation	SICSlowpan	BLIP	NanoStack
Programming language	C	nesC	C
Hardware platforms supported	20	12	26
Kernel (Scheduler) properties	Run to completion (event-based), threads FIFO, non-preemptive	Run to completion (event-based), threads, priority-based, preemptive	Real-Time, threads, priority-based, preemptive or cooperative

Table 3: Operating system comparisons

TinyOS is written in nesC while Contiki and FreeRTOS was written in pure C language. The time to learn a new language weighed against TinyOS. However, the later versions of TinyOS allows the programmer to write applications in C.

The operating system chosen for this project was Contiki, the main reason being the TCP support.

IP stack	**uIPv6**	**BLIP**	**NanoStack**
Supported OS	Contiki	TinyOS	FreeRTOS
UDP	Yes	Yes	Yes
TCP	Yes	No	No
Route-over	Yes	No	No
Mesh-under	No	Yes	Yes
Neighbor discovery	Yes	Yes	No

Table 4: Comparison of IP stacks

4.2 Node Hardware

With 6LoWPAN as starting-point the Contiki operating system was chosen, and based on Contiki's platform support the hardware could be selected. Two hardware platforms were considered for the project, AVR Raven and Zolertia Z1 [29][30]. Raven is an Atmel product with an 8-bit microcontroller and Z1 is a design from Zolertia with a 16-bit microcontroller.

Z1 is based on an older platform called Telosb/Tmote. Contiki has been ported to and well tested on the Telosb platform, however the platform was no longer available on the market. A Contiki port for Z1 platform was planned to be released at the start of this project but was postponed. Contiki port for AVR Raven already existed. A comparison of the two platforms can be seen in table 5.

Platform	**Raven**	**Z1**	**ZigBit**
MCU	ATmega 1284P 8-bit	MSP430F2617 16-bit	Atmega1281V 8-bit
Operation Frequency	20 MHz	16 MHz	4 MHz
Flash program memory	128 kB	92 kB	128 kB
EEPROM	4 kB	32 B	4 kB
SRAM	16 kB	10 kB	8 kB
Radio Transceiver	AT86RF230 2.4GHz at 250kbps	CC2420 2.4GHz at 250kbps	AT86RF230 RF Transceiver at250kbps

Table 5: Hardware platform characteristics

Platform	**Dimensions (Height x Width x Length)**
ZigBit:	2.0mm x 13.5mm x 18.8mm
S8:	8mm x 33mm x 20mm
K30:	14mm x 57mm x 51mm

Table 6: Platform dimensions

The characteristics for both platforms were similar, the main difference being the microcontroller. Z1 had a 16-bit microcontroller compared to AVR Raven's 8-bit.

However, due to the delayed porting of Contiki for Z1 platform, the Raven platform felt like a safer option.

Another advantage with Raven was the architecture similarities to ZigBit. ZigBit is a very compact 802.15.4/ZigBee Atmel module with almost the same features as the Raven platform, see table 5 [31]. This would give the possibility to integrate ZigBit modules with SenseAir's S8 sensors [1]. S8 sensor is the newest sensor platform developed by SenseAir, figure 6. The main characteristics for both the S8 sensor and the ZigBit module is their compact size, which can be seen in table 6.

Figure 6: S8 sensor

4.2.1 AVR Raven board

The AVR Raven board consist of two AVR microcontrollers (1284P and 3290P) and one radio transceiver (AT86RF230) [32]. An Universal Synchronous and Asynchronous serial Receiver and Transceiver (USART) is used for internal communication between the two microcontrollers. The board also contains a Liquid Crystal Display (LCD).

Aside from theses hardware parts the AVR Raven board includes some features, like the 32 kHz clock crystal that is connected to both microcontroller's asynchronous timer interfaces. Another feature is the voltage regulator that allows the board to be powered by either two 1,5V LR44 battery cells or by an external 5-12V DC power supply.

Atmega 3290P

The secondary microcontroller, ATmega 3290P is used to handle the LCD display on the board. ATmega 3290P is an 8-bit AVR microcontroller with 32 kB flash memory, 1 kB EEPROM and 2 kB SRAM [34].

AT86RF230

AT86RF230 is a low-power 2.4 GHz transceiver designed for low-cost IEEE 802.15.4 [35]. The transceiver is designed for low power consumption and is connected to the Atmega 1284P.

4.2.2 K30 Sensor

The K30 is a sensor platform developed by SenseAir and is used for CO_2 measurement [1]. Its measurement range is between 0 and 5000 parts per million by volume (ppmv). The expression ppm is used to describe the concentration of one substance in another, in this case it refers to

CO_2 concentration. The sensor can be powered by a 4,5 to 14V Direct Current (DC) source and has a baud rate of 9600 bps. For external communication there is an Universal Asynchronous Receiver/Transmitter (UART) serial interface with TxD and RxD lines, where Modbus communication protocol is used. Though this sensor platform does not support measurement of temperature and humidity, there are other SenseAir's sensor models that include these features. Because the same Modbus communication protocol is used for all sensors developed by SenseAir the sensor model can be replaced without any modification to the application software.

The K30 sensor platform was provided by SenseAir. The sensor is connected to the AVR Raven board with 3 wire RS-232.

4.2.3 JTAG ICE mkII

JTAG ICE mkII is a development tool for Atmel 8-bit and 32-bit devices [36]. JTAG ICE is used to program and debug the AVR boards.

4.2.4 Power supply

As the power was not an issue in this project an external 9V power supply is used to power both the Raven board and the sensor. The sensor is not powered via the Raven board nor the other way around, both are connected directly to the external power source.

4.3 Node Application

All AVR Raven boards are running Contiki operating system. The application was developed as a Contiki program, written in standard C with Contiki specific coding definitions.

4.3.1 Setup environment

Cooja is an advanced Java-based simulation environment for Contiki operating system [23]. The tool allows the programmer to simulate a wireless sensor network with user specified Contiki application. The advantage of using a controlled environment is the possibility to observe and verify the application code before deploying it into hardware. Cooja was used extensively during this project, especially in the beginning when no hardware was available.

AVR Studio 4 software was provided with the RZRAVEN kit [29]. This software is an Integrated Development Environment (IDE) that is used to build and debug applications for 8-bit AVR microcontrollers. However the software could only be run under Windows operating systems while Contiki software tools were tested for Linux, which is why the AVR Studio 4 was never fully explored. Though, it was useful for debugging code deployed on the hardware.

A C compiler is used to compile Contiki together with the node application code.

4.3.2 Application states

Contiki is a process-based, event-driven operating system which means the node application is implemented with events and blocking macros. This also means that the node application functionality can be illustrated as a sequential state-machine. The application consists of two processes, network process and main process. Two state diagrams of the processes are shown in figure 7 and 8. These figures are good to have in mind when reading the remaining of this section where the basic node application behavior will be described.

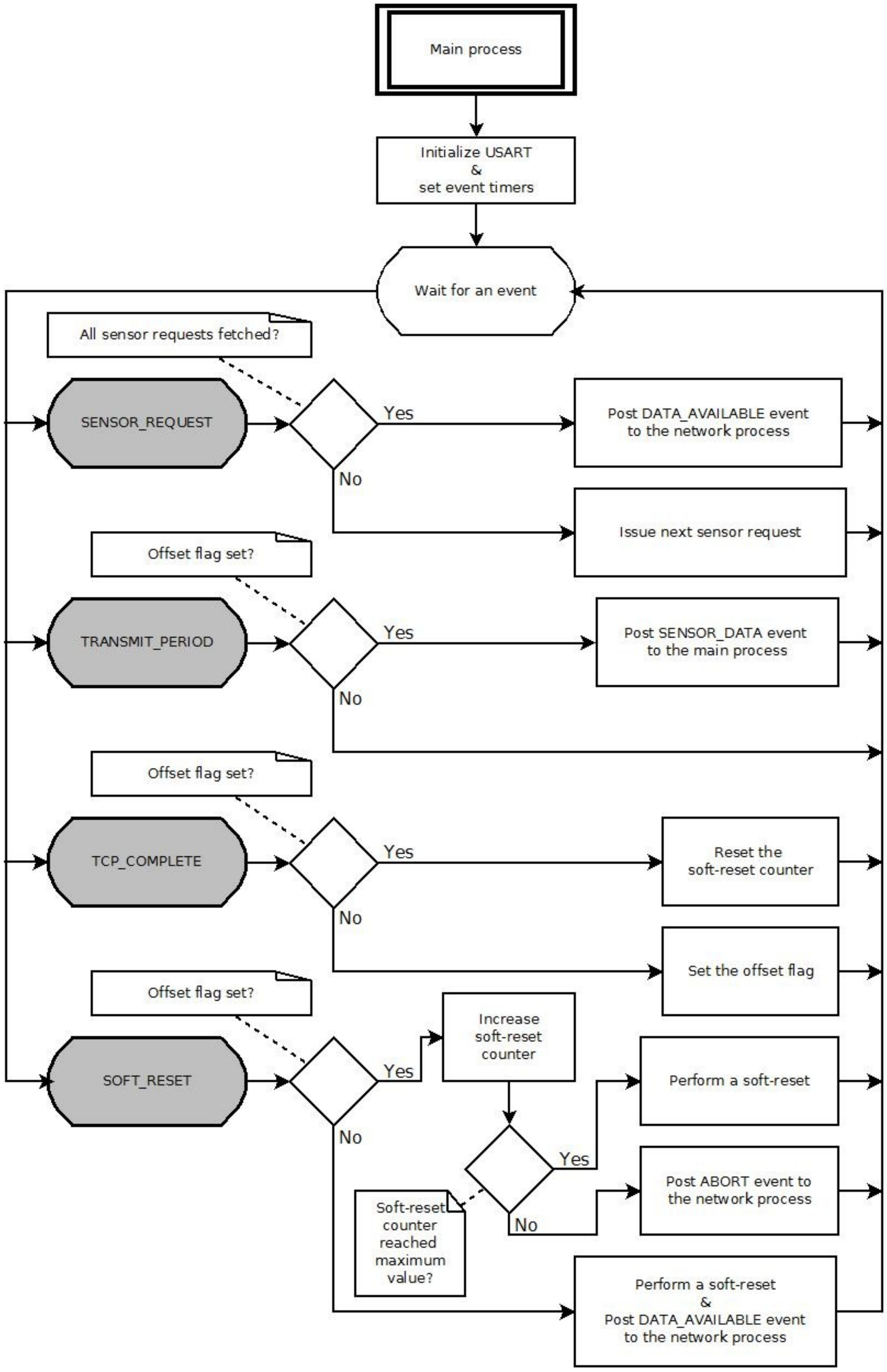

Main process
Initialize USART & set event timers
Wait for an event
All sensor requests fetched?
SENSOR_REQUEST
Yes
No
Post DATA_AVAILABLE event to the network process
Issue next sensor request
Offset flag set?
TRANSMIT_PERIOD
Yes
No
Post SENSOR_DATA event to the main process
Offset flag set?
TCP_COMPLETE
Yes
No
Reset the soft-reset counter
Set the offset flag
Offset flag set?
SOFT_RESET
Yes
No
Increase soft-reset counter
Soft-reset counter reached maximum value?
Yes
No
Perform a soft-reset
Post ABORT event to the network process
Perform a soft-reset & Post DATA_AVAILABLE event to the network process

Figure 7: Main process state diagram

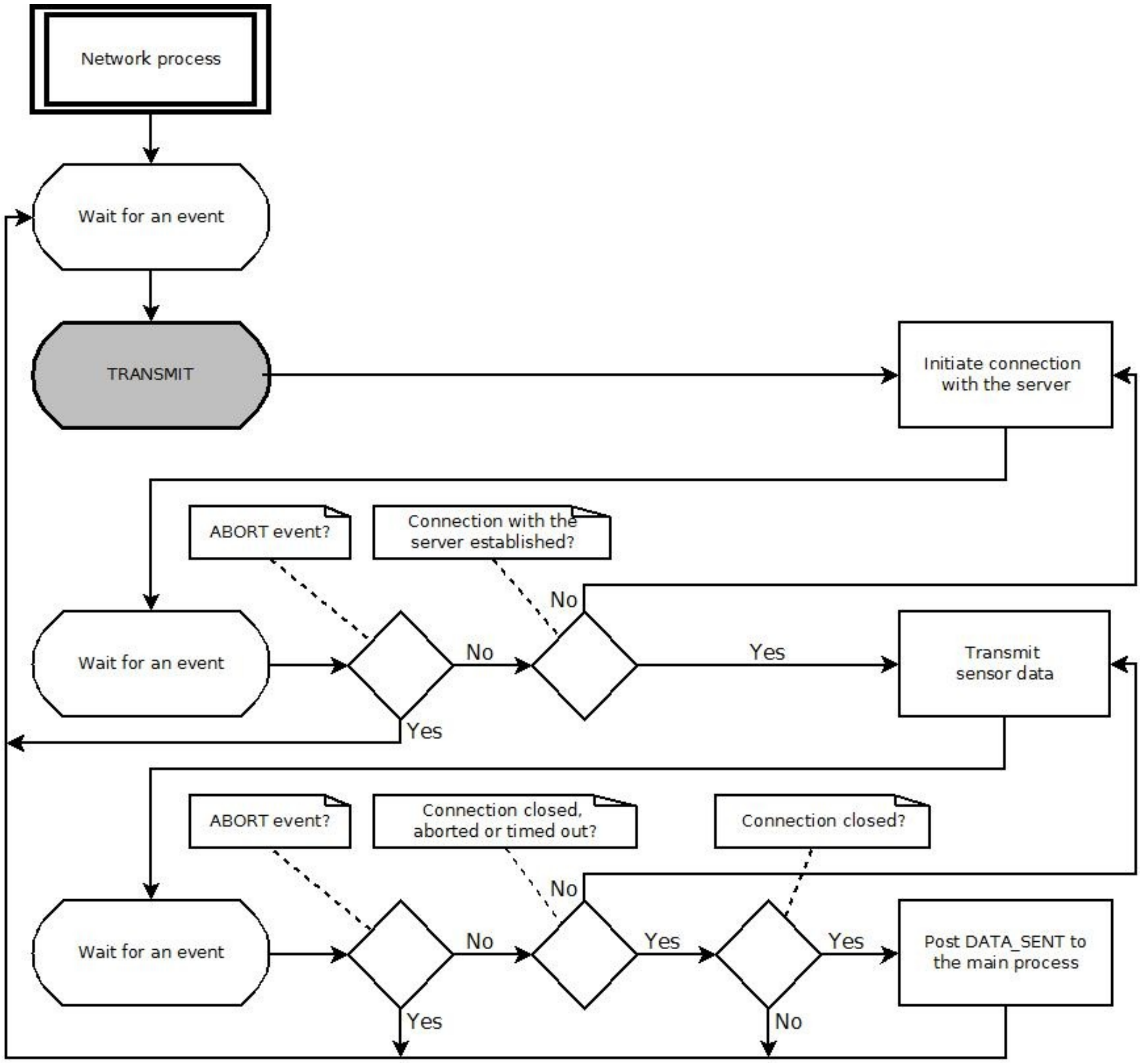


Figure 8: Network process state diagram

Main process

The purpose of the main process is to initiate transmission of sensor data at specified time intervals. This is achieved by event timers provided by Contiki, which are used to generate timed events. When an event timer expires an event will be posted to the process which set the timer, in this case the main process. The main process controls the application flow based on what events are triggered. When an event is triggered, for example a timer expires or an event is posted from another thread, the Contiki kernel will activate the main process and pass the information regarding the event. Depending on the event type, the main process will enter different states. All the actions within the state will be executed after which the main process will block and wait for another event.

The main process consists of four states which are activated by four different events, two user defined event types and two timer events. The four states are; SENSOR_REQUEST, TRANSMIT_PERIOD, TCP_COMPLETE and SOFT_RESET.

SENSOR_REQUEST

This state's main task is to initiate *sensor requests* to the sensor. A sensor request is a hard-coded frame that is sent over serial line to the sensor, requesting specific sensor data. Sensor data could for example be CO_2, temperature or humidity value. Each sensor data can be fetched with separate sensor request implementations. Therefore all sensor requests has to be processed within this state before transmitting to data collection server. All the different sensor requests that are issued in this state are called a *request chain*. Presently CO_2 and temperature value are fetched from the sensor. The byte values representing CO_2 and temperature are placed consecutively in the RAM, which allows them to be fetched by a single request. Therefore the request chain contains a single sensor request, although more requests can be implemented.

The state is triggered by the SENSOR_DATA event, which is posted either from the main process itself or by an input handler function which processes the fetched sensor data. The main process triggers this event when a new transmission is scheduled to take place. From the input handler function the event is posted every time a sensor response has been properly processed, but can also be posted if something went wrong with the sensor communication.

Based on the *request pointer*, the state will issue either the next sensor request in the request-chain, or a transmission of the fetched sensor data. The request pointer points at the sensor request that is currently handled. However, would something be wrong with the sensor communication the same sensor request will be issued again. After each sensor request is sent to the sensor, this state is exited and the main process will block awaiting another event to happen.

When all sensor requests within the request chain are issued and all sensor data is successfully fetched, an event will be posted to the network process, clearing it to transmit the received sensor data to the data collection server. The data will also be sent to the 3290P MCU using a custom serial line protocol, which will display the value on the LCD.

TRANSMIT_PERIOD

The TRANSMIT_PERIOD state keeps track of when to initiate a request chain, that is when to transmit new sensor data to the data collection server. This state is triggered by an event timer, *transmit_timer*, every 30 seconds. Each time transmit_timer expires an event will be posted to the main process by the kernel. When the main process reaches this state the transmit_timer will be reset with the same time interval.

It is important to distinguish the two terms *reset* and *restart* when talking about event timers in

Contiki. Resetting an event timer in Contiki will set the starting point of the new interval to be the same as when the timer last expired. For example if a timer expired five seconds ago and reset is used for that timer, the new starting point will be five seconds ago. Exactly the same time as it last expired, even though that time is in the past. The time interval of the event timer will be unchanged. This feature enables event timers to be stable over time, which is exactly what is needed in this case. On the other hand when an event timer is restarted the starting point will be at that specific time the restart function was used.

In this state the transmit_timer is reset, reset_timer is restarted and the request pointer is set to point at the first sensor request. Then a request chain is initiated, which means the SENSOR_DATA event is posted to the main process itself. The event will be placed in an event queue and will shortly after trigger the main process again.

TCP_COMPLETE

The TCP_COMPLETE state is triggered by the DATA_SENT event. The DATA_SENT sent is posted from the network process every time a TCP connection with the server has been closed. Reaching this state means that everything went according to the plan and it is now up to the transmit_timer to trigger another request when it expires.

Should it be the first time the main process enters this state, it means the offset mark has just been successfully sent to the server. The offset mark is the system clock of the node in seconds which is used on the server side to adjust the time, see section 6.3.3. At this point an offset flag, *offset_sent,* will be set that implies the offset mark has been transmitted and the reset_timer is set to 25 seconds. This part of the state is only executed when an offset mark has been transmitted to the server.

All the other times this state is reached it indicates that a transmission with the server has been finalized and therefore the soft-reset variable is zeroed. The soft-reset variable is used within the SOFT_RESET state to determine if a soft-reset should be performed or not. Soft-reset is performed by shutting down the network process and start it up afresh. This way the system clock and time synchronization are still intact. A hard-reset on the other hand refers to rebooting the node, which will restart everything from scratch including the system clock.

SOFT_RESET

The SOFT_RESET state is triggered when the reset_timer expires. This state is mainly a safe-state that will prevent deadlock and restart the network process if no successful transmission have been reported for a while.

This state also handles offset transmission, with the reset_timer reduced to five seconds. The reason for this is to have a small error margin for the offset mark and still keep the transmit_timer synchronization. Reaching the SOFT_RESET state during offset mark transmission means the offset mark has not been successfully transmitted within the five second period. The network process will be restarted and another attempt to transmit the offset mark is performed. This part of the state will be executed until the offset mark has been successfully transmitted to the data collection server. When that happens the offset_sent flag will be set, which will change the functionality of this state and also set the reset_timer to 25 seconds.

During normal transmissions this state functions as a soft-reset. Each time the reset_timer expires, there is still 5 seconds before the transmit_timer triggers another sensor request. The soft-reset counter is increased, which counts amount of unsuccessful transmissions. Soft-reset counter is set to allow three consecutive unsuccessful transmission before a soft-reset is performed. Should the soft-reset counter reach the maximum value it will indicate that the network process is stuck somewhere or the server is unreachable. The network process is then killed and restarted again. By shutting down the network process the connection is also aborted which allows for a new connection to be established. If the counter has not reached the soft-reset value an ABORT event is posted to the network process. This event will instruct the network process to abort the ongoing connection and reach its initial blocking state before another request is initiated by the transmit_timer.

Network process

The functionality of the network process is to establish a stable connection with the data collection server and transmit the sampled sensor data. All processes in Contiki contain some kind of blocking macros, and the network process is no exception.

TRANSMIT

The network process has only one main state, TRANSMIT, which will be triggered by the DATA_AVAILABLE event. The process will block until the event is posted, instructing the process to transmit new sensor data to the server. Inside this main state there are also two other necessary TCP events used to perform the transmission to the server. These two TCP states both block until they receive a TCP event from the uIPv6 stack or the ABORT event from the main process. When the ABORT event is received the network process simply returns back to its main blocking state awaiting another DATA_AVAILABLE event.

The first TCP state will block until a connection with the server is established, which will be posted by the uIPv6 stack. Should the connection fail, the process will try again until it succeeds or receives an ABORT event.

Should a successful connection with the server be established the network process will initiate a protosocket and enter a loop to complete the transmission. Within this loop a protosocket function is called to send the data through the initiated protosocket. However each time a data segment is transmitted the protosocket will block within the function and await permission from uIPv6 stack to continue, which is the reason for the loop in network process. When a TCP event is received by the network process the protosocket function will be called again. This loop will continue until the function runs to completion, which will close the connection at node side and end the protosocket. There are also two more TCP events that can end the loop, those are if the connection was aborted or timed out. Aside from these three TCP events the ABORT event is also acceptable, which will abort the ongoing connection.

4.3.3 Application functionality

Three different implementations of the data collection part have been tested, both on the node and the server side. Those are RESTful, TCP and UDP collection, see section 6.3.3. RESTful collection performs transmission over the HTTP application layer protocol, while TCP and UDP collection uses a custom made protocol. In this section the application functionality and the differences between the three implementations on the node side will be described.

RESTful collection

When a node is powered on both main process and network process are started simultaneously. The network process creates an uIPv6 address from the hard-coded server-address, before reaching its initial blocking state, TRANSMIT.

The main process calls the function *usart_initialize* at start-up. This function initializes the USART that is used to communicate with the K30 sensor. A Contiki defined function is used to configure the USART with correct settings. There are two USARTs for the ATmega 1284P; USART0 is used internally between the two MCUs and USART1 can be used for external communication. In this case USART1 is used to communicate with K30 at a baud rate of 9600 bps. This baud rate is default for most of the sensor models at SenseAir. The USART configuration is set to correspond to the K30 Modbus communication protocol settings. Also during the initialization an input handler, *sensor_input_handler*, is set for USART1 that will manage the incoming sensor data.

After initialization of the USART is complete, two event timer are set, transmit_timer and reset_timer. The transmit_timer is used to keep the transmitting interval synchronized. The reset_timer will be used for soft-reset functionality, and as a transmitting interval timer for the offset mark. The transmit_timer is set to 30 seconds and the reset_timer to 5 seconds. The first attempt to communicate with the server is made now by posting the DATA_AVAILABLE event to the network process. This is done just before the main process reaches its default blocking state.

The network process receives the DATA_AVAILABE event, which is the only event that triggers its initial blocking state. The network process then tries to establish a connection with the server by calling the Contiki function *tcp_connect*, attaching the server IP address and port number

8080. Then the network process blocks again until a tcpip_event is posted to the process from the uIPv6 stack. If the received event indicates the connection was accepted the network process will take one step further to complete the transmission. Should the uIPv6 stack post any other event at this state the network process will start over again requesting a new TCP connection. When an connection is established the network process will initiate a protosocket, which handles the TCP connection. By calling the protosocket function send_data an attempt is made to transmit the fetched sensor data. Even here the network process blocks until it receives a releasing event, either from the uIPv6 stack regarding the connection or from the main process aborting the connection.

The function send_data is called within the network process and is used to transmit data to the server. This is a protosocket function, which will send the sensor data over TCP, using HTTP as application level protocol. The data is sent as HTTP requests, in which corresponding sensor values are inserted.

If an offset mark is to be sent, no sensor values are needed. The current system clock time in seconds is transmitted to the data collection server within an HTTP request, see table 7. During this transmission the send_data function will block until an acknowledgment is received from the server. When a confirmation is received the send_data function will post the DATA_SENT event to the main process and then close the current connection on the node side.

Should sensor data be transmitted an HTTP request containing the sensor data values is transmitted to the server, see table 7. During this transmission no confirmation is needed, instead the function closes the connection and let's the network process send the DATA_SENT event when the whole connection is closed.

Packet type	***Packet layout***
Offset	*"PUT /sensenet/sensor/offset?ts=50 HTTP/1.1\r\n"*
Data	*"PUT /sensenet/sensor/?ts=120&co2=550&te=26&hu=0 HTTP/1.1\r\n"*

Table 7: HTTP offset and sensor data request layout

The first time node contacts the server the DATA_SENT event will be posted within the send_data function. Because it is the first time the main process enters this state the offset_sent flag will be set and the reset_timer is set to 25 seconds.

The next event that triggers the main process will be the transmit_timer. When the transmit_timer puts the main process into the TRANSMIT_PERIOD state the timer will be reset as always. Because the offset_sent flag has been set at this point the reset_timer will be restarted, the request pointer reset and SENSOR_DATA event posted to the main process itself. The event will trigger the SENSOR_REQUEST, which will initiate the sensor request to the K30 sensor by calling the sendframe function.

The sendframe function takes an array containing a sensor value request. The sensor requests are hard-coded arrays of raw hex values. Currently one request is implement, which fetches both CO_2 and temperature values. More requests can be implemented, though the SENSOR_REQUEST state need to be slightly modified to handle those requests. The current CO_2 and temperature request is implemented as an 8-bit array. The Modbus communication protocol is used between the sensor and the Raven board and hence the requests are implemented as frames with Modbus specific function codes. A detailed description of the CO_2 and temperature request frame can be seen in table 8. This request is gathered from sensor's RAM with the user defined function code 0x44, *read from RAM*.

Address field	Function code	Data			CRC	
		Starting address		# of Bytes		
0xFE	0x44	0x00	0x08	0x04	Low-byte	High-byte

Table 8: CO2 and temperature request

The Modbus protocol also requires a 16-bit CRC value to be calculated on the payload contents. The CRC has to be appended as the last filed to the request as two 8-bit values, with the low-order byte first. The CRC calculation is performed by the *calc_crc* function which will return a 16-bit calculated CRC of the sensor request payload. The frame is then sent one byte at a time using the Contiki function *rs232_send*, which prints one byte to the RS232 module. No events are posted from here, because the application now awaits response from the sensor, which will be posted to the input handler (sensor_input_handler) initialized at main process start-up.

The sensor_input_handler function will receive one byte at a time from the sensor as a response to the CO_2 request. The first two bytes, sensor address and function code, will be checked to match the expected response, which should be the same as in the request. Any unexpected byte will post an SENSOR_DATA event to the main process, which in return will issue the same request again, because the request pointer has not yet been changed. The third byte received will indicate payload data length, in bytes. This value will be stored in an array. The last two bytes will contain a CRC of the whole response frame calculated by the sensor. A new CRC calculation is performed on the incoming response, which is why the first three bytes are also stored in the array. The newly calculated CRC is compared against the received CRC. Should there be a mismatch, the main process will be notified just as before and same request will be issued again. Should the CRC's match, current system clock will be saved. The saved system clock is the time the sensor data was acquired. The received CO_2 and temperature values will be stored as a 16-bit values.

Before posting the SENSOR_DATA event to the main process the request pointer is increased to point at the next request, which is needed within the SENSOR_DATA state. The second request is temperature and the third request is humidity. All sensor values will be stored at specified position in the array. Those values will later be accessed when HTTP transmission is performed.

Presently only CO_2 and temperature are fetched. A reset value used for debugging, which counts the number of soft-resets performed by a node, is stored instead of the humidity value.

On receiving the SENSOR_DATA event the main process enters the SENSOR_REQUEST and sends the next sensor request until the whole request-chain has been successfully received. When all sensor data is acquired, the main process will post the DATA_AVAILAIBLE event to the network process.

The network process will go through the same procedure as before. Though this time the DATA_SENT event will not be posted by the send_data function, instead the network process will post the event when a connection at the server side is closed.

When the main process receives the DATA_SENT event it will reset the soft-reset counter that counts the number of consecutively aborts. When the transmit_timer expires next time a new request chain will be started.

SOFT_RESET state is reached 25 seconds after each request chain is issued by the TRANSMIT_TIME state, which is 5 seconds before the transmit_timer expires. Within this state the network process is either forced to its initial blocking state or completely restarted.

TCP collection

To meet the server requirements for this approach minor changes were made within the node application. The functionality of the application is the same as described for RESTful. The differences were in packaging of the collected sensor data and the transmission protocol used. RESTful collection stored sensor data as separate 16-bit values, which can easily be inserted into the application level HTTP transmissions. With TCP collection, a custom protocol was designed to handle the transmission of sensor data between nodes and data collection server. The collected sensor values are stored in an 8-bit array, *sensor_data*. When a transmission is made to the server, the sensor_data array is transmitted over TCP as raw data bytes.

For an offset mark the sensor_data array will be empty, because no sensor values has been fetched. The status code will be stored at position 0, assigned the value 0 which indicates an offset mark packet. The current system clock will be stored as a 32-bit value at position 1-4, with the high-order bytes first. The length of the TCP segment will be 5 bytes, see table 9.

Status	Offset mark
1 byte	4 bytes

Table 9: A five byte packet containing the offset mark

For a sensor data packet the status bit will be set to 1, indicating sensor data. The length of the TCP packet will be 11, see table 10. The sensor_data array will contain the system clock of the last time a sensor value was acquired from the sensor. This information will be stored at same position as for the offset mark packet (1-4). The remaining 6 bytes are used for CO_2 value, temperature, and humidity, two bytes for each sensor value. CO_2 is stored at position 5-6, temperature at 7-8 and humidity at 9-10. The present application functionality uses soft-reset variable at position 9-10.

Status	Offset mark	CO_2 value	Temperature value	Humidity value
1 byte	4 bytes	2 bytes	2 bytes	2 bytes

Table 10: An 11 byte packet containing sensor data

UDP collection

For the UDP collection slightly more modification was necessary. Because UDP gives no assurance that the packets will reach the server, the node does not receive any confirmation from the server and thus can not know if the packet has been delivered successfully. Hence no need for the TCP_COMPLETE and SOFT_RESET events within the main process. Besides from this the functionality of the main process is the same as before.

The state diagram for the network process is shown in figure 9. The network process uses UDP protocol to transmit the sensor data. The sensor data packaging is the same as for TCP implementation, where the size of the packets were five bytes for offset and 11 bytes for sensor data. Data collection server will respond on the offset mark, letting the network process know that the offset mark has been delivered. Upon receiving confirmation the network process enters its simple state where it only can be triggered by the DATA_AVAILABLE event. Each time the network process is triggered an UDP segment will be sent, which contains the sensor data, after which the network process returns to its blocking state again to await another DATA_AVAILABLE event.

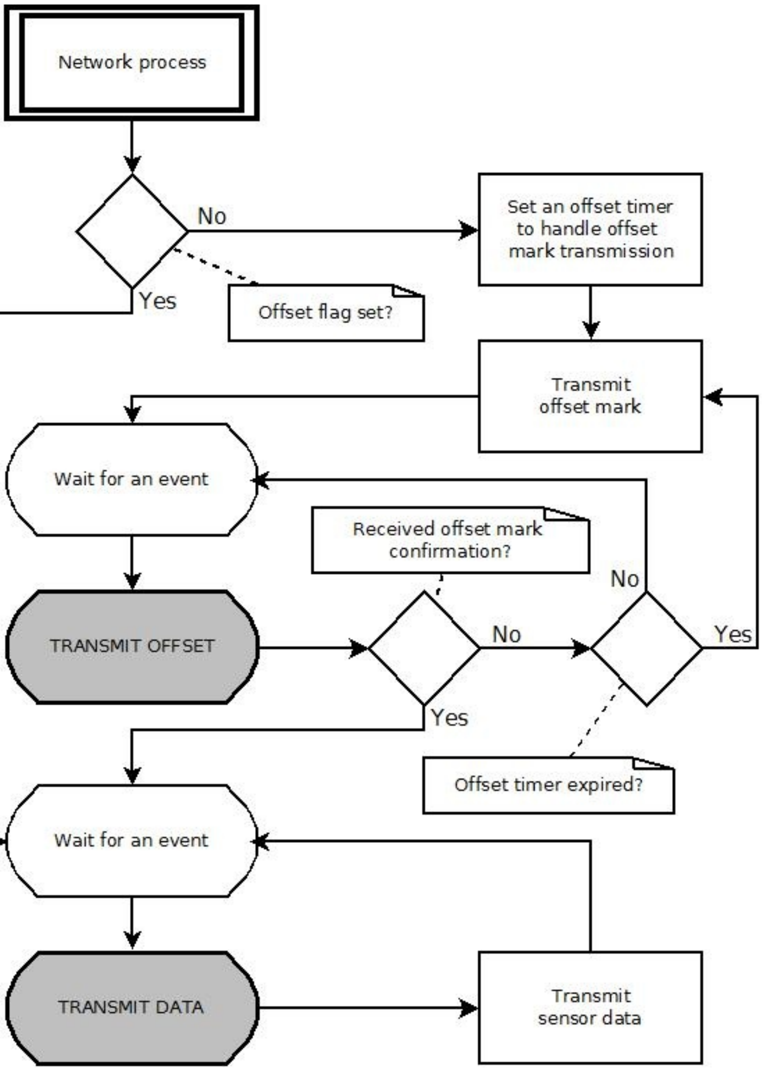


Figure 9: UDP network process state diagram

5 Edge Router

The edge router's main task is receiving packets on the 6LoWPAN network and forward them onto the LAN. This device should be kept as simple as possible, ideally it should not perform any task other than above described. The reason for this is to uphold the end-to-end principle of the TCP/IP stack.

5.1 PC

The basis of the edge router consisted of a regular PC fitted with a Network Interface Card (NIC) and a 802.4.15 wireless NIC. The PC was supplied by SenseAir, as such no major design decisions were made on this hardware. The only consideration was that the PC should be able to run a modern operating system. The specifications for the PC were as shown by table 11. The wireless NIC however had to be researched and procured, see section 5.2.

Processor	Intel Celeron 2.80 GHz
RAM	Generic RAM (2 GB)
Harddisk	Samsung SP0411N (40 GB)
NIC	Intel 82562 EZ 10/100 Ethernet Controller

Table 11: PC specification

The operating system used was Ubuntu [37]. There were several reasons for this choice, the main one being Contiki was well tested with Ubuntu. Windows was considered in the early stages of the project as some of the supporting software only was available for Windows, for example Atmel Studio, a programming environment for Atmel hardware. Similar software was available for Linux, though the Linux equivalent often proved more difficult to use as well as having less functionality. However it was decided that the advantages of using Linux would outweigh the initial convenience of using Windows. One significant advantage is the greater flexibility in system configuration which Linux offers.

5.2 Jackdaw

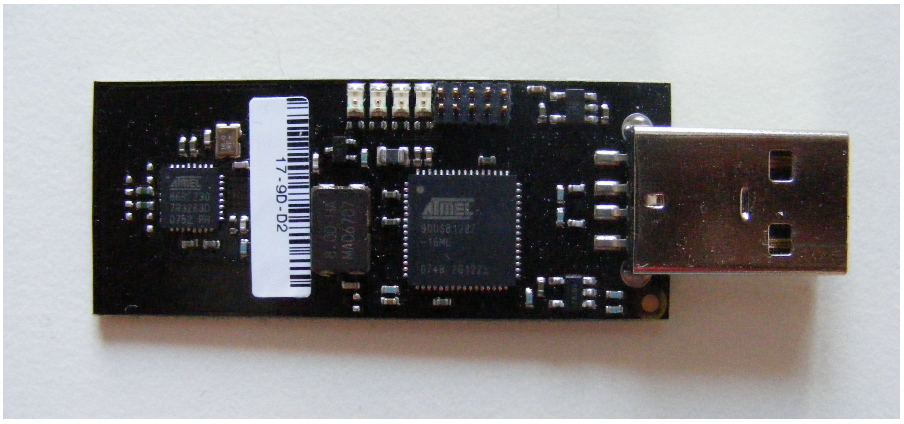
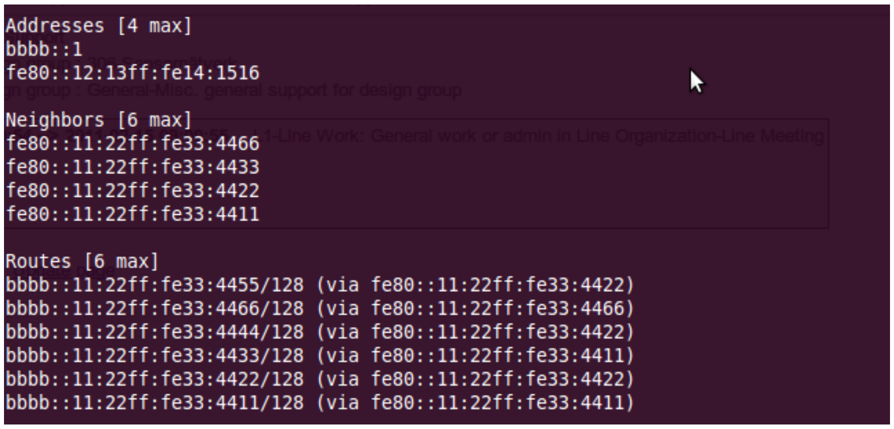


Figure 10: RZUSBSTICK

Jackdaw was chosen as the 802.4.15 wireless NIC [38]. Jackdaw is the codename for a hardware/software combination developed by the Contiki team.

The hardware part is an Atmel RZUSBSTICK, see figure 10 [32]. The RZUSBSTICK is equipped with an Atmel AT90USB1287 microcontroller. The main task of the MCU is to manage usb communication and the onboard AT86RF230 radio transceiver. The AT90USB1287 has 8KB of internal ROM, which can be extended to 32KB. The RAM size is the main constraint for amount of routes that can be mapped. With the default 8KB RAM, the Jackdaw can map up to six routes, hence it can support a network with up to six nodes. This restriction arises from the fact that there is no 6LoWPAN stack available for Linux, as a temporary solution the Jackdaw translates IEEE 802.15.4 frames to Ethernet frames before routing them. Additionally this also places constraints on what MAC addresses can be used for the nodes [23].

The software part consists of a Contiki installation configuring the usbstick as an edge router. When connected to the PC the usbstick registered as usb0. The Jackdaw was recognized as an Ethernet device, as such all the usual network configuration commands are usable on it.


```
Addresses [4 max]
bbbb::1
fe80::12:13ff:fe14:1516

Neighbors [6 max]
fe80::11:22ff:fe33:4466
fe80::11:22ff:fe33:4433
fe80::11:22ff:fe33:4422
fe80::11:22ff:fe33:4411

Routes [6 max]
bbbb::11:22ff:fe33:4455/128 (via fe80::11:22ff:fe33:4422)
bbbb::11:22ff:fe33:4466/128 (via fe80::11:22ff:fe33:4466)
bbbb::11:22ff:fe33:4444/128 (via fe80::11:22ff:fe33:4422)
bbbb::11:22ff:fe33:4433/128 (via fe80::11:22ff:fe33:4411)
bbbb::11:22ff:fe33:4422/128 (via fe80::11:22ff:fe33:4422)
bbbb::11:22ff:fe33:4411/128 (via fe80::11:22ff:fe33:4411)
```
Figure 11: Jackdaw network information

The Jackdaw has a serial interface, accessible through a terminal emulator. Figure 11 shows a screenshot of the main menu. A number of settings can be made and inspected through this interface. The neighbours menu is particularly interesting as it lists all neighbours and routes currently registered. Figure 11 shows a typical setup with six nodes.

5.3 Configuration

With the hardware properly installed the PC has access to the LAN by default. Initially the Ethernet NIC is configured only with an link-local address, for routing purposes it must be assigned a global address. The convention used for assigning global addresses was to use the link-local address as template, and replace the prefix *ff02*, which is the link-local prefix, with a suitable global prefix. The chosen prefix was *aaab*. The global address derived from this is used by the web server as its interface.

To bring the wireless network online some configuration is necessary. The first task is to configure the PC as a router on the wireless network. For this the application radvd is used, radvd broadcasts router advertisements onto a network [39]. The broadcasted prefix is used in the stateless auto configuration process of IPv6. A sample configuration file for radvd is available on the Contiki website. Settings in this file govern aspects such as when and with which interval router advertisements should be broadcasted, and which prefix should be used on the network. The modified configuration file used in this project uses the prefix *bbbb*. Initially the Jackdaw is only configured with a link-local IPv6 address. For routing purposes the Jackdaw must be assigned a global address, which matches the network prefix. By default the address is *bbbb::1*.

In addition to configuring the PC, certain modifications were also made to the code running on the Jackdaw. By default IPv6 is turned off on the Jackdaw, this option is of course required. Also by default a web server is compiled with Jackdaw, where the user can examine various network information, such as routing tables and neighbours. This restricts the number of routes supported by the Jackdaw to two, and therefore also the number of nodes supported in the network to two. In order to increase the number of routes the web server was disabled.

A general configuration guide for the edge router can be found on the Contiki web page for further reference [23].

6 Server

The server has two main tasks or modules, data collection and data presentation.

6.1 Hardware

To simplify matters the server runs on the same hardware as the edge router. Though, in concept the server is separated from the edge router and is designed to run on any computer connected to the edge router.

6.2 Software Technologies

A number technologies were considered for the server software, the two main candidates were PHP and Java Enterprise Edition (Java EE) [40][41].

PHP is a scripting language used for building websites and systems. Its strength is simplicity, it offers rapid development and a low learning curve. However as it is a scripting language it is not suited for performing operations on large amounts of data. Hence PHP would be excellent for the data collection module. However as the presentation module would need to perform some kind of calculation after fetching data, most often calculating averages, it would suffer in performance.

Java EE on the other hand is well suited for this task. Java EE is the specification for a web development framework built on top of Java SE, Java SE being the *regular* Java flavour. Though programming in EE is quite a bit different than SE, and the learning curve is steep compared to that of PHP. Microsoft's .NET framework offers much of the same benefits as Java EE, but was not considered due to the licensing costs of their servers, where as there are free open source Java EE implementations. Mono which is an open source version of .NET is quite a bit slower than Java EE, and was therefore dismissed [42]. Though it should be noted that the increased performance came at the cost of increased memory usage.

In an effort to use an unified platform for both of the servers main tasks Java EE was chosen. This decision was made to simplify the development of the server software. This allows all server software to be written in the same language. Thus not only code can be shared between the two modules, but run-time resources can also be shared more easily. An example is the database layer, which both tasks utilize. The data collection module uses the layer to store incoming data, the presentation module uses it to fetch data.

6.2.1 Application Server

There are a number of application servers available for Java EE, but at the time of this project only two were compatible with Java EE 6, Glassfish 3 and JBoss AS6 [43][44]. Glassfish is developed by Oracle and JBoss AS6 by JBoss(supported by Redhat). The two servers offer the same functionality, but JBoss was chosen as it had better development tools, such as better integration with Eclipse IDE[45]. Though in a production environment Glassfish might prove the better choice as it is considered to be more lightweight, a quality much appreciated when dealing with constrained hardware resources.

6.2.2 Database server

MySQL was chosen as the database server. It is the prevailing open source SQL server, widely used in web applications and has a reputation for being stable and fast [46]. It is very easy to install and has some excellent tools, such as phpMyAdmin, for setting up and administrating the database [47]. MySQL also has a stable Java Database Connectivity (JDBC) driver, which is required for connecting to the database from Java.

6.3 Software

6.3.1 Database

The database schema consists of two tables, *sensors* and *datas*, shown in figure 12.

The sensors table holds information about all the sensors in the system. The id column is an auto incrementing primary key column. Primary key means this column uniquely identifies each row in the table, auto incrementing means this number will increment by one for each new row in the table. The name column holds an optional string which describes the sensor. The IP address holds the complete IPv6 address of the sensor. This column has the constraint *unique*, that is this column also uniquely identifies each row. Hence the ip address could be used as primary key, in which case the id column is redundant. However as the primary key in the sensor table is also a foreign key to the datas table, *sensorid*, a primary key of type int is preferable to one of type varchar(255), due to performance reasons. The offset marks in server time when the sensor started transmitting data, see section 6.3.3.

The datas table holds the data for all sensors. The sensorid column indicates to which sensor the data belongs. As stated the sensorid is a foreign key, as such the sensorid value must exist in the sensors table. Timestamp indicates at which time the data points in this row were collected on the node. Due to Java measuring time in milliseconds since the epoch, this is of type bigint. The sensorid and timestamp columns together form a compound primary key for the datas table. That is each row is uniquely identified by each unique combination of sensorid and timestamp. The remaining three columns are data points, co2concentration is measured in ppm, temperature in ºC and humidity in %. When the sensor does not have the capability to measure humidity, this field is used for debugging the network and will then measure number of soft resets, see section 4.3.3.

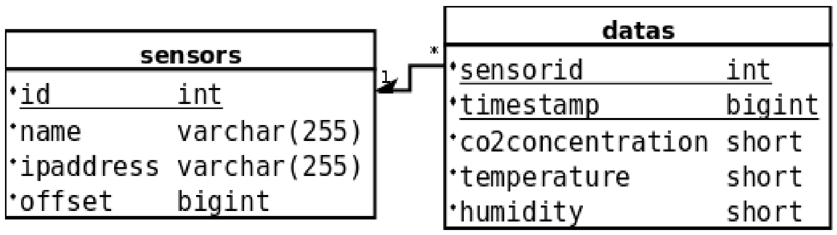


Figure 12: Database schema

6.3.2 Database layer

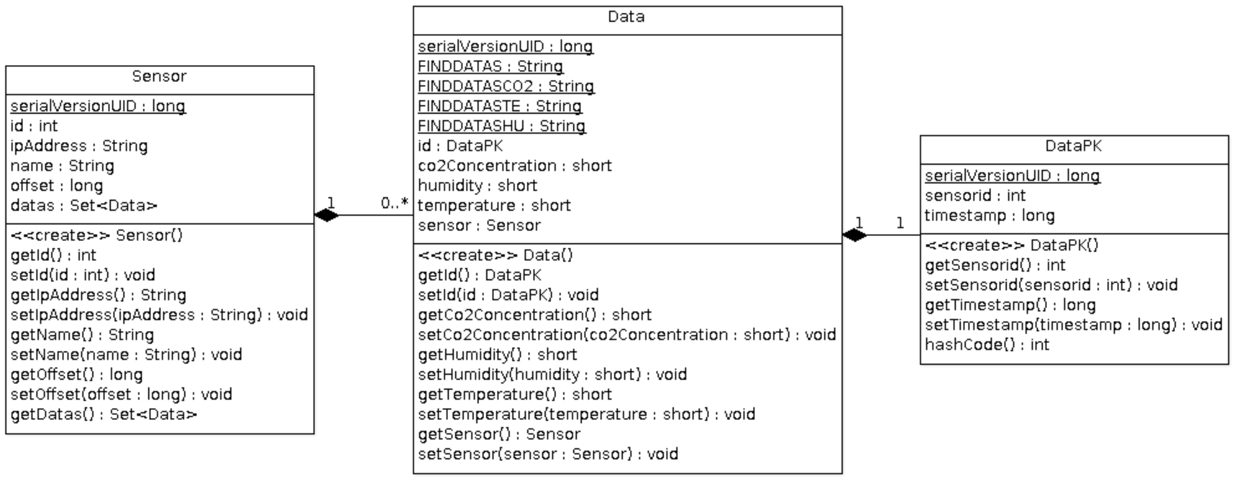


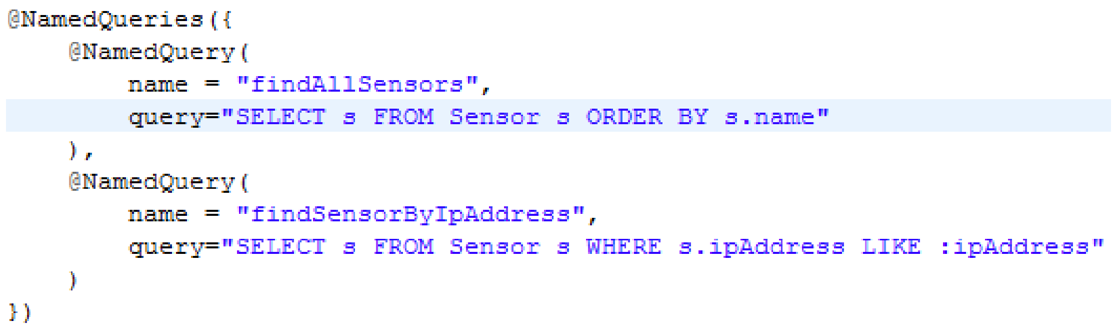
Figure 13: Database abstraction layer

The database layer provides access to the database through Java Persistence API (JPA). Source code for the database layer is found in the se.senseair.sensenet.base package, illustrated in figure 13. JPA allows the mapping of database tables to special Java classes called entities.

In general the instance variables of an entity map to a column in the table. The type of the variable is chosen to be as close to the MySQL counterpart as possible. A few of the instance variables listed might seem odd, as they do not have a counterpart in the table. The *id* variable in the Data class for example is of class DataPK, PK short for Primary Key. Since the datas table has a compound primary key it can not be represented by a simple variable. A custom class had to be created to model the key. In DataPK the two primary key columns for the datas table are found.

Associations are represented by additional variables with the class of the associated tables class. For example the foreign key sensorid in the datas table is represented by the sensor instance variable in the Data class, it has the type Sensor. So instead of getting the foreign key first and execute a separate query manually into the sensors table, one can immediately get the Sensor object. Respectively, in the Sensor class the datas variable represents the collection of datas which are associated with each sensor.

The three classes depicted in figure 13 offer a complete model of the database, and queries to the database can now return Java objects. Each object represents a row in the database, changing a value in the object will change the value in the database. JPA requires queries to be performed in a certain manner. First off queries are written in Java Persistence Query Language (JPQL). Though very similar to SQL in JPQL tables are not queried, but Java classes, as shown in figure 14.



```
@NamedQueries({
    @NamedQuery(
        name = "findAllSensors",
        query="SELECT s FROM Sensor s ORDER BY s.name"
    ),
    @NamedQuery(
        name = "findSensorByIpAddress",
        query="SELECT s FROM Sensor s WHERE s.ipAddress LIKE :ipAddress"
    )
})
```

Figure 14: JPQL sample code

Writing queries in JQPL allows the database layer to be database agnostic, that is the type of database (MySQL, MSSQL, Oracle etc.) can be exchanged without changing the code. Merely the connection configuration of the data-source need to be changed. A data-source is Java's representation of the database, its connection configuration holds information such as the URI of the database, the user and password with which to log onto the database and what type of database it is.

Each data-source is associated to a persistence unit, this links the models from figure 13 to the data-source. The persistence unit is managed by an entity manager, on which JPQL queries are performed. The entity manager is usually provided by the container, but can also be programmatically created. All operations on a database are handled through a single entity manager to allow for concurrency, else race conditions might arise on the entities. Finally operations on an entity manager must be enclosed in a transaction. Transactions provide atomicity at the entity level. All operations in a transaction are committed at the end of the transaction.

Enterprise JavaBeans (EJB) is a Java API for managing the business logic of a Java EE application. EJBs simplify database access by for example automatically initiating and closing transactions at the start and end of a java function. If a Plain Old Java Object (POJO) is used transactions would have to be manually marked in code. The interface to the database layer is DBBean, which is an EJB bean. The operations found in this bean are shown in figure 15. Figure 15 also shows that DBBean has an interface, DBLocal. Generally an EJB will always have an local or remote interface. A local interface will allow the bean only to be accessed by local applications, where as a remote interface allows external applications to access this bean.

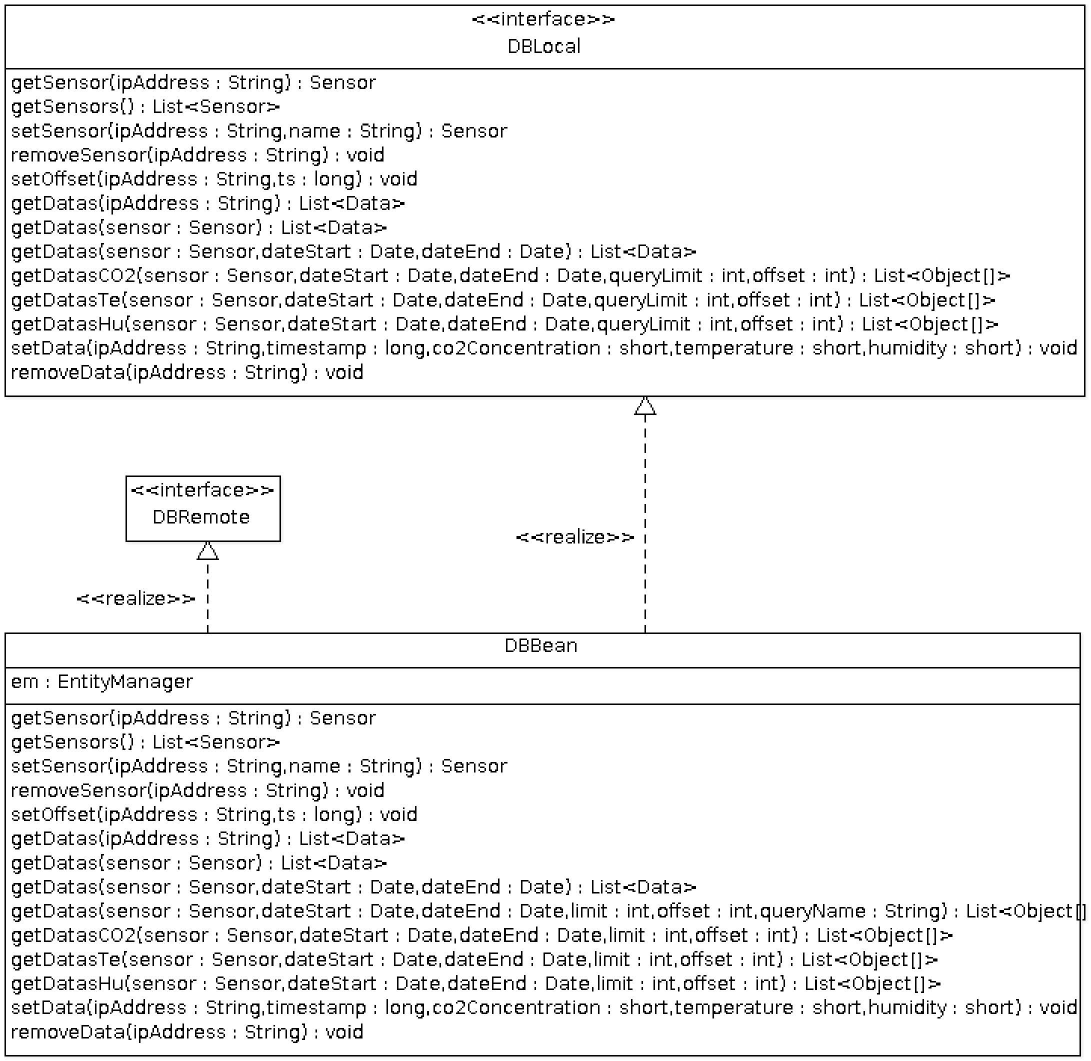

<<interface>>
DBLocal
getSensor(ipAddress : String) : Sensor
getSensors() : List<Sensor>
setSensor(ipAddress : String,name : String) : Sensor
removeSensor(ipAddress : String) : void
setOffset(ipAddress : String,ts : long) : void
getDatas(ipAddress : String) : List<Data>
getDatas(sensor : Sensor) : List<Data>
getDatas(sensor : Sensor,dateStart : Date,dateEnd : Date) : List<Data>
getDatasCO2(sensor : Sensor,dateStart : Date,dateEnd : Date,queryLimit : int,offset : int) : List<Object[]>
getDatasTe(sensor : Sensor,dateStart : Date,dateEnd : Date,queryLimit : int,offset : int) : List<Object[]>
getDatasHu(sensor : Sensor,dateStart : Date,dateEnd : Date,queryLimit : int,offset : int) : List<Object[]>
setData(ipAddress : String,timestamp : long,co2Concentration : short,temperature : short,humidity : short) : void
removeData(ipAddress : String) : void
<<interface>>
DBRemote
<<realize>>
<<realize>>
DBBean
em : EntityManager
getSensor(ipAddress : String) : Sensor
getSensors() : List<Sensor>
setSensor(ipAddress : String,name : String) : Sensor
removeSensor(ipAddress : String) : void
setOffset(ipAddress : String,ts : long) : void
getDatas(ipAddress : String) : List<Data>
getDatas(sensor : Sensor) : List<Data>
getDatas(sensor : Sensor,dateStart : Date,dateEnd : Date) : List<Data>
getDatas(sensor : Sensor,dateStart : Date,dateEnd : Date,limit : int,offset : int,queryName : String) : List<Object[]
getDatasCO2(sensor : Sensor,dateStart : Date,dateEnd : Date,limit : int,offset : int) : List<Object[]>
getDatasTe(sensor : Sensor,dateStart : Date,dateEnd : Date,limit : int,offset : int) : List<Object[]>
getDatasHu(sensor : Sensor,dateStart : Date,dateEnd : Date,limit : int,offset : int) : List<Object[]>
setData(ipAddress : String,timestamp : long,co2Concentration : short,temperature : short,humidity : short) : void
removeData(ipAddress : String) : void

Figure 15: Servicebean for database operations

6.3.3 Data collection

This section will describe the procedure used for data collection, and describe the three implementations included in the project.

Aside from collecting data, the system must also be able to indicate when the data was collected. One straightforward solution would be to register at what time the data arrives at the server and assume the transmission time to be negligible. Unfortunately this can incur a rather large error as TCP is used for transmission. TCP utilizes retransmission when a packet is lost, this means a packet may arrive at its destination much later than expected. A better solution is to transmit a timestamp along with the data, the system clock of the node is excellent for this purpose. The system clock measures time in seconds since boot up, hence if the node is rebooted the system clock is reset. The server and node clocks must therefore be synchronized.

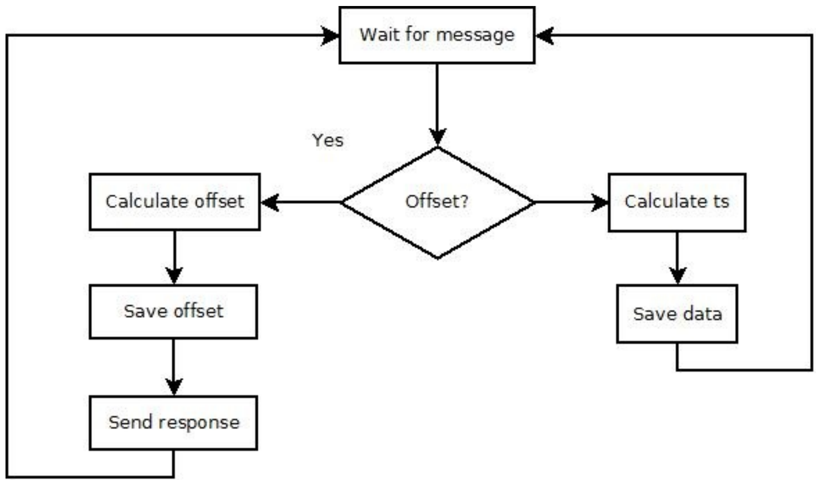


Figure 16: Server workflow

Synchronization is achieved through an offset message sent by the node, transmitted shortly after boot up. The offset message contains the current node clock, upon reception the server subtracts this value from the current server time and storing it to the database as the offset value, see section 6.3.1. Figure 16 show this work flow. This offset value will be added to the timestamps of all future incoming data transmissions, calculating the approximate time of data collection on the node.

With this solution there can still be a discrepancy if the offset message suffers retransmissions, which results in all future data transmissions having a delayed timestamp. To mitigate this the node will only allow a windows of five seconds for the offset message to successfully be acknowledged, else a new offset with an updated node time will be sent. This allows the regular data transmissions to have a window as large as the interval at which data is sent. That is, since data is sent every thirty seconds, the node may send retransmissions for up to thirty seconds before resigning.

In all three implementations the node is identified by its ipaddress. The server examines the HTTP or TCP segments and extracts the source ipaddress, simplifying traffic slightly by not requiring the transmission of an additional identifier.

RESTful collection

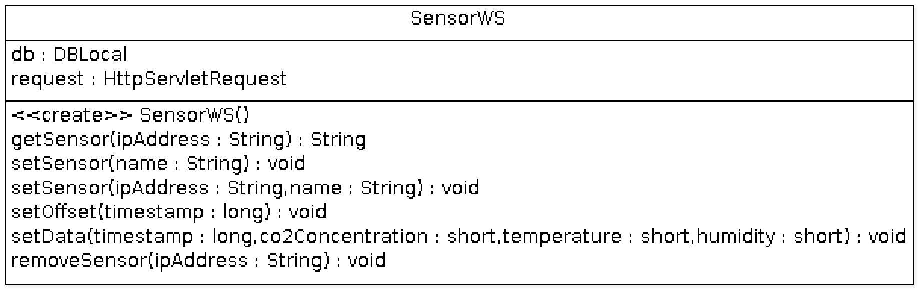

SensorWS
db : DBLocal
request : HttpServletRequest
<<create>> SensorWS()
getSensor(ipAddress : String) : String
setSensor(name : String) : void
setSensor(ipAddress : String,name : String) : void
setOffset(timestamp : long) : void
setData(timestamp : long,co2Concentration : short,temperature : short,humidity : short) : void
removeSensor(ipAddress : String) : void

Figure 17: Webservice provider

Data collection is performed through a RESTful web service. It is implemented with RESTEasy, which is the JBoss implementation of JAX-RS. JAX-RS is a Java API specification for RESTful web services [48]. It allows the mapping of Java functions to a specific combination of Uniform Resource Identifier (URI) and REST operation. Figure 17 shows the SensorWS class, which implements the web service. Each function in this class is mapped to its own combination of URI and REST operation. The base URI used is sensenet, the absolute location of the web service is "http://[aaab::20f:feff:fe3c:4ca3]:8080/sensenet/", all site URIs are relative this.

The SetOffset function is the first step in the time synchronization. When a node comes online, it will call this function first and supply it with the current node time. SetOffset is mapped to the a PUT on "sensor/offset".

The node then starts delivering data by sending the current node time and sensor data. This triggers the SetData function. SetData is mapped to a PUT on "sensor/".

The response on a successful REST request is always a HTTP 204 "No Content", indicating there is no message body [18].

TCP collection

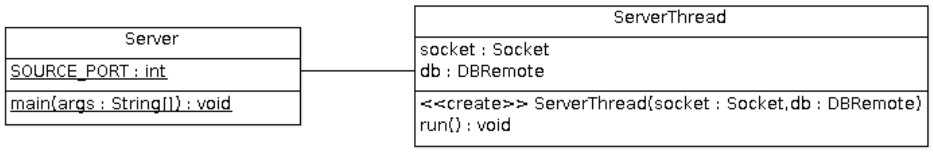

Server
SOURCE_PORT : int
main(args : String[]) : void
ServerThread
socket : Socket
db : DBRemote
<<create>> ServerThread(socket : Socket,db : DBRemote)
run() : void

Figure 18: TCP server classes

This implementation uses a TCP link for data collection. A stand alone TCP server was written in Java, figure 18 shows the class diagram. The Server class accepts incoming TCP connections and spawns a ServerThread for each connection. The ServerThread class performs the same duties as the SensorWS class does in the RESTful collection. It parses incoming messages and stores data to the database, the latter is made possible by DBRemote. DBRemote is the remote interface for DBBean, see figure 15.

Two types of messages are sent by the node, offset and data, see section 4.3.3. The first byte of a received message is examined to determine the type of message. In case an offset message was received, a one byte response is sent, to notify the node that the offset was successfully transmitted. No response is sent for a data message. Port 20000 is used both locally and remotely for the connection.

UDP Collection

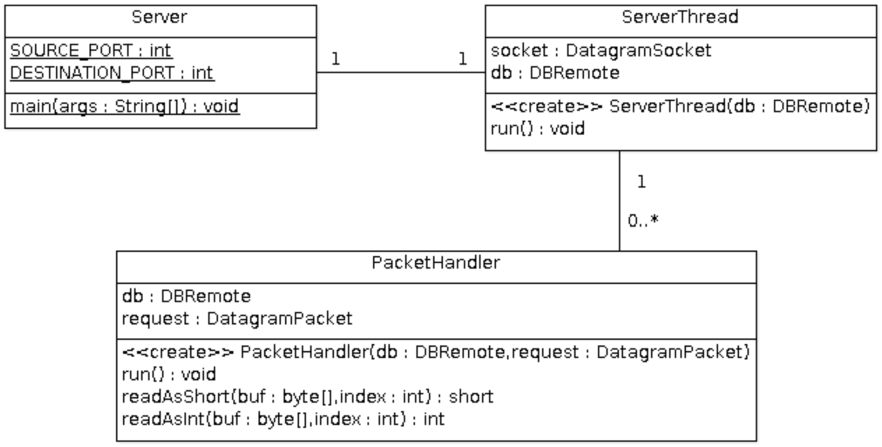

Server
SOURCE_PORT : int
DESTINATION_PORT : int
main(args : String[]) : void
1
1
ServerThread
socket : DatagramSocket
db : DBRemote
<<create>> ServerThread(db : DBRemote)
run() : void
1
0..*
PacketHandler
db : DBRemote
request : DatagramPacket
<<create>> PacketHandler(db : DBRemote,request : DatagramPacket)
run() : void
readAsShort(buf : byte[],index : int) : short
readAsInt(buf : byte[],index : int) : int

Figure 19: UDP server classes

UDP collection, like TCP collection, uses a stand alone Java server. The Server class, shown in figure 19, spawns a ServerThread which accepts incoming UDP segments. For each packet a PacketHandler thread is spawned to process it. The payload uses the same format as TCP collection. PacketHandler will therefore determine whether the packet is an offset or data packet in the same fashion as is done in TCP collection, and take the same actions as TCP collection. Incoming UDP packets are received on port 20001, response packets are sent to port 20002 on the node.

6.3.4 Data presentation

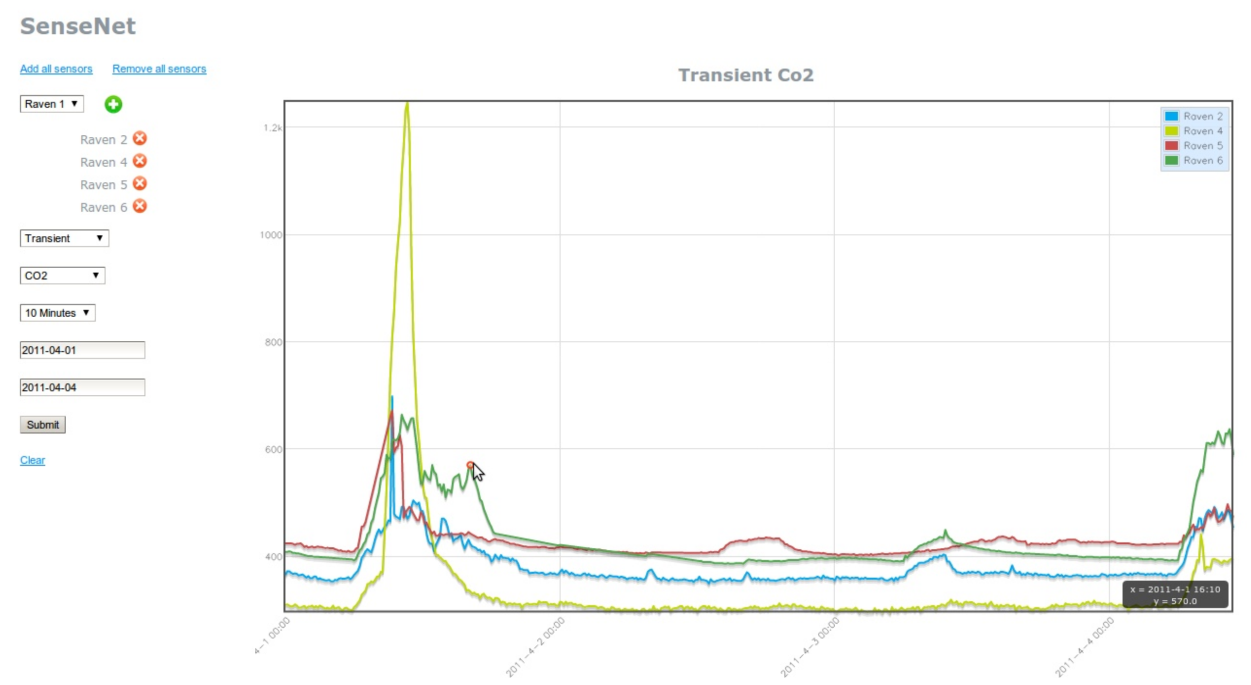


Figure 20: Web page screenshot

Data presentation is performed through a single web page, shown in figure 20. In the left pane there is a menu where the user can choose which sensor and what kind of data (CO_2, temperature or humidity) to show. The user can also set a date range. In the right pane there is a chart which presents the selected data.

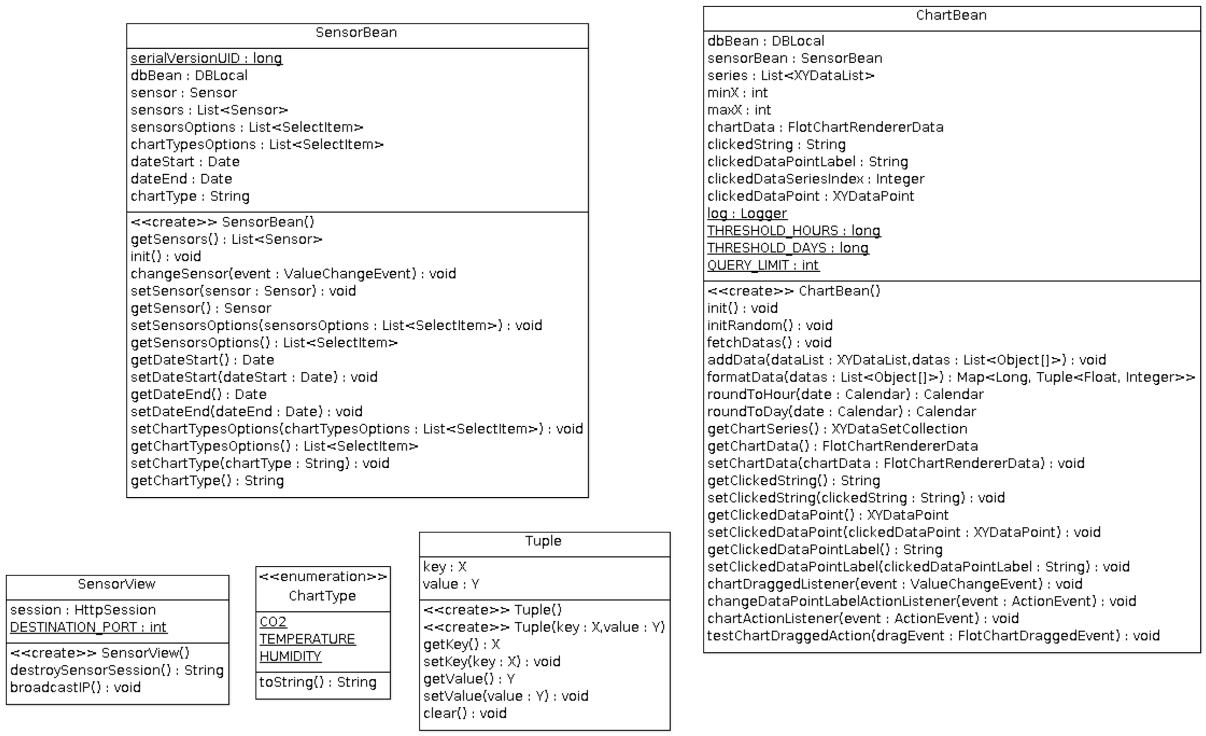

SensorBean
serialVersionUID : long
dbBean : DBLocal
sensor : Sensor
sensors : List<Sensor>
sensorsOptions : List<SelectItem>
chartTypesOptions : List<SelectItem>
dateStart : Date
dateEnd : Date
chartType : String
<<create>> SensorBean()
getSensors() : List<Sensor>
init() : void
changeSensor(event : ValueChangeEvent) : void
setSensor(sensor : Sensor) : void
getSensor() : Sensor
setSensorsOptions(sensorsOptions : List<SelectItem>) : void
getSensorsOptions() : List<SelectItem>
getDateStart() : Date
setDateStart(dateStart : Date) : void
getDateEnd() : Date
setDateEnd(dateEnd : Date) : void
setChartTypesOptions(chartTypesOptions : List<SelectItem>) : void
getChartTypesOptions() : List<SelectItem>
setChartType(chartType : String) : void
getChartType() : String
ChartBean
dbBean : DBLocal
sensorBean : SensorBean
series : List<XYDataList>
minX : int
maxX : int
chartData : FlotChartRendererData
clickedString : String
clickedDataPointLabel : String
clickedDataSeriesIndex : Integer
clickedDataPoint : XYDataPoint
log : Logger
THRESHOLD_HOURS : long
THRESHOLD_DAYS : long
QUERY_LIMIT : int
<<create>> ChartBean()
init() : void
initRandom() : void
fetchDatas() : void
addData(dataList : XYDataList,datas : List<Object[]>) : void
formatData(datas : List<Object[]>) : Map<Long, Tuple<Float, Integer>>
roundToHour(date : Calendar) : Calendar
roundToDay(date : Calendar) : Calendar
getChartSeries() : XYDataSetCollection
getChartData() : FlotChartRendererData
setChartData(chartData : FlotChartRendererData) : void
getClickedString() : String
setClickedString(clickedString : String) : void
getClickedDataPoint() : XYDataPoint
setClickedDataPoint(clickedDataPoint : XYDataPoint) : void
getClickedDataPointLabel() : String
setClickedDataPointLabel(clickedDataPointLabel : String) : void
chartDraggedListener(event : ValueChangeEvent) : void
changeDataPointLabelActionListener(event : ActionEvent) : void
chartActionListener(event : ActionEvent) : void
testChartDraggedAction(dragEvent : FlotChartDraggedEvent) : void
SensorView
session : HttpSession
DESTINATION_PORT : int
<<create>> SensorView()
destroySensorSession() : String
broadcastIP() : void
<<enumeration>>
ChartType
CO2
TEMPERATURE
HUMIDITY
toString() : String
Tuple
key : X
value : Y
<<create>> Tuple()
<<create>> Tuple(key : X,value : Y)
getKey() : X
setKey(key : X) : void
getValue() : Y
setValue(value : Y) : void
clear() : void

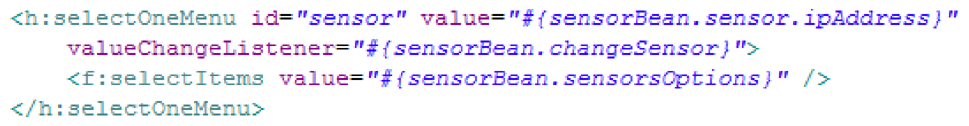
Figure 21: Backing beans

The web page is implemented with Java ServerFaces (JSF). JSF is a framework for simplifying front-end integration in Java EE. The content of the page is written with HTML, JSF tags and Expression Language (EL). JSF tags and EL offer a substitute for scripting, which is usually used when creating dynamic web pages. A dynamic web page has content which the user may interact with, such as a web form. The JSF components allow access to the so called backing beans where the logic for the web page resides, before being presented to the web client the JSF components are compiled to HTML.

The *sensorview* page refer to three backing beans; SensorBean , ChartBean and SensorView, see figure 21. All three are so called *managed beans*. Their entire life cycle, creation to destruction, is managed by the server. The programmer need not explicitly create objects of these classes. Instead when a web page is visited, and said page requires access to one of the backing beans, the bean will be created if it does not already exist. Likewise the bean will be destroyed when its scope ends. The two most commonly used scopes are request and session. A request scoped bean will be destroyed when the web page request has ended, a session scoped bean will be destroyed when the client's session ends. When a client first connects to a web server a session is created on the server for that client, the session holds data which should be persistent over several requests for that client. All three backing beans in this project are request scoped.

SensorBean holds data regarding the choice of sensor. The currently chosen sensor and a list of available sensors can be fetched from this bean. ChartBean holds data configuring the chart.

Settings such as type and color of the chart can be found in this bean. Finally SensorView holds functionality for the two links *Reset* and *Broadcast*.



```
<h:selectOneMenu id="sensor" value="#{sensorBean.sensor.ipAddress}"
    valueChangeListener="#{sensorBean.changeSensor}">
    <f:selectItems value="#{sensorBean.sensorsOptions}" />
</h:selectOneMenu>
```

Figure 22: JSF tags sample code

Figure 22 shows an example where JSF tags and EL are used to create a drop-down menu. This specific code creates the drop-down for selecting sensor. The JSF tag is *h:selectOneMenu*, it creates the drop-down itself. The value attribute of this tag sets what value the drop-down currently has. EL allows this link to be two-way. That is, when the form is submitted the active value of the drop-down will be stored to the instance variable of the managed SensorBean object, when the form is created the value will be read from the same variable. This removes a lot of so called boiler plate code, which simply moves data from front-end to back-end. The valueChangeListener attribute refers to a Java function which is triggered when the value of the dropdown is changed, this link is not two-way as the target is a function. The enclosed tag *f:selectItems* sets what options should be listed in the drop-down. The value attribute here functions much like the value attribute in the *selectOneMenu* tag, difference being here it takes only an instance variable which is a list. The *selectItems* tag will iterate the list to fill the drop-down with options. Hence JSF is used to define the content of the page and process it, to define the layout of the page Cascading Style Sheets (CSS) are used.

When a node comes online it has no knowledge of what ip address the server has, this is provided to the node by means of Service Discovery. However there is no clear implementation of Service Discovery in 6LoWPAN yet, this expired draft seems to be the most recent development [50]. A stopgap solution is to broadcast the servers ipaddress when the network is started, or on demand when a node is added to the network. This functionality is handled by SensorView. When the Broadcast link is clicked, the SensorView bean creates and transmits an UDP multicast to all nodes in a network. Finally the SensorView bean also allows all settings in the form on the web page to be cleared, through the Reset link.

7 Results

The purpose of this thesis was to evaluate 6LoWPAN as a viable alternative for a SenseAir WSN implementation. Though the performance proved to be dependent on what protocols were used higher up in the network protocol stack. Hence, this section will compare the performance of the three different implementations created in the project.

A WSN containing five nodes was deployed in an office environment. The nodes were placed five to ten meters apart from each other, and were placed in the same location for each test run. The lowest index node was closest to the edge router, distance increasing with node index. Node one was therefore closest and node five the farthest away. The nodes that were farthest away routed through a middle node. Three test batches were run, one for each of the implementations, each run took three days. The data is shown in table 12 and figure 23. The reference row indicates the theoretical number of data points which a node should have delivered. It can be concluded that the heavyweight REST protocols performance suffers compared to TCP and UDP, particularly for nodes with weak links.

	Transmitted amount of data points		
	RESTful	TCP	UDP
Reference	**8075**	**8233**	**8316**
Node 1	7936	8224	8304
Node 2	7907	8218	8271
Node 3	7525	8210	8210
Node 4	7071	7966	8092
Node 5	7169	8025	8009

Table 12: Transmission test results

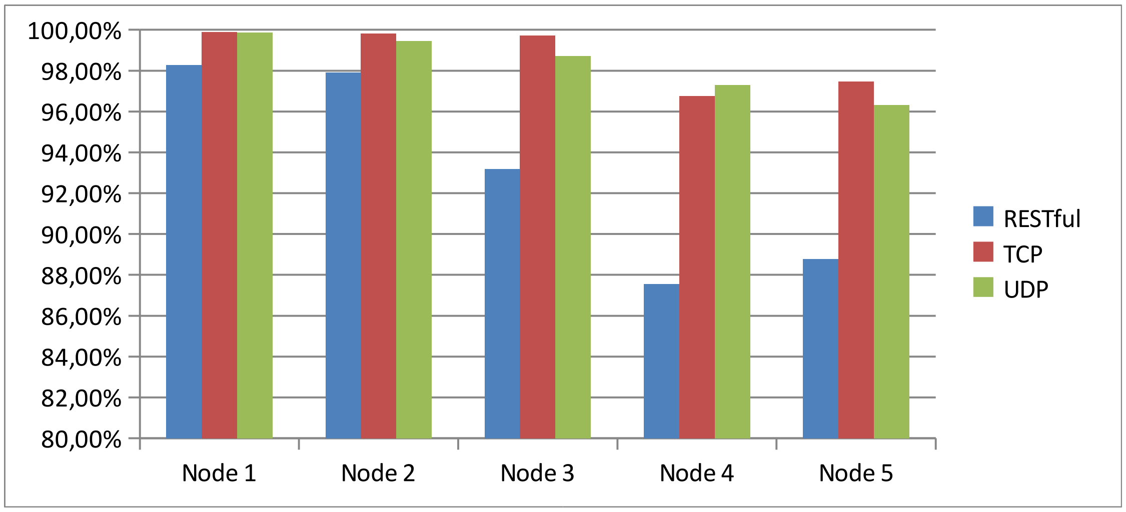


Figure 23: Transmission test results in graph

In addition to transmission performance, accuracy of the Raven system clock was also proved to be a problem. Due to a Contiki configuration issue on Raven hardware, the system clock drift was roughly one second every 20 minutes. Due to this the clocks had to be resynchronized quite often, which was done by sending offset messages at regular intervals.

Another unsolved issue was the the reception of UDP traffic on the nodes. Although the nodes sent out UDP segments without fail, they did not receive them properly. Due to this the UDP implementation differs somewhat from TCP and RESTful in the work flow. In addition the broadcast functionality described in section 6.3.4 does not function properly.

8 Conclusions

The prototype system works as intended. SICSlowpan proved to be a stable implementation of 6LoWPAN. Furthermore it is well integrated into Contiki, an operating system providing excellent support for low-bandwidth devices. Contiki also proved easy to work with, although it did see a lot of development during the course of this project, which made it slightly more difficult to test. Still it should be noted that Contiki was the only operating system offering support for TCP at the time.

The data presentation works as intended. It should be noted that data availability is obscured when longer time intervals are requested by the client. As it is not interesting to present every single data point over longer intervals, an average is calculated. The first threshold is at three days, when an interval greater than this is requested all data points are averaged to the nearest hour. The second threshold is at one week, an interval greater than this and only one data point per day will be shown. However, should only an hours worth of data be present for a day, this hour will come to represent the entire day.

8.1 RESTful collection

The REST architecture is the de facto standard for lightweight machine to machine over the world wide web. Though it may still be too heavyweight for low bandwidth networks such as 6LoWPAN. On the other hand RESTful collection was very easy to implement. With a framework supported by Java EE, it integrated well into an Java EE application server. It also uses an established application level protocol, HTTP. The programmer needs only to acquaint himself with a custom API. The negative performance of REST is in this project partly due to the large overhead. Since only a small amount of data is sent in each transmission, it is not very effective. Using a larger transmit interval and storing data on the node instead will reduce the overhead, and might also increase performance. But for small frequent transmissions REST is not recommended.

Finally efforts to adapt application level protocols for low-power wireless networks are being made by 6LoWAPP, a sister IETF work group to 6LoWPAN. REST may therefore be worth revisiting in the future.

8.2 TCP collection

TCP collection offers significantly lower overhead compared to RESTful collection. However, a custom application protocol has to be specified, which in most cases is not desirable. TCP collections also offers a measure of safety in data transmissions, but this comes at a cost. Even though the overhead is lower compared to the RESTful collection, the overhead to payload ratio is still high. Despite this TCP collection offers quite good performance and should be considered the implementation of choice.

8.3 UDP collection

UDP collection is the most lightweight of the implementations. This is due to the UDP protocol which does not use hand-shaking and therefore offers much lover overhead. But this also comes with the drawback that no guarantee is given for a successful transmission.

9 Future Work

A number of improvements and additions can be made to the prototype, a short list will be presented in this section.

Presently only CO_2 data is fetched. But the prototype also supports temperature and humidity values with minor modifications to the node software.

Two-way communication is a slightly more complex task. With two-way communication a node would act both as client and server, which would allow it to receive requests from the web server. With this feature the nodes could be remotely reconfigured while up and running. A desirable request could be to change the transmitting interval. Another request could be to fetch other sensor data besides from those already being periodically transmitted.

Time synchronization is done with the offset mark packet when the node is powered on. From that point no more time synchronization will take place unless there has been a hard-reset. This has proven to be a problem as the node system clock tend to drift, due to configuration issues in Contiki. After three days of running time each node's system clock was about three minutes ahead of the sever clock. During longer periods, for example years, the time difference would be in range of hours. A more frequent time synchronization would be recommended functionality to implement in the future.

References

[1] SenseAir. Maintenance-free Gas Sensors. http://senseair.com. Accessed 2011-01-24.

[2] Shu, X. (2010). Wireless CO_2 Sensors.

[3] ZigBee Alliance. ZigBee Alliance. http://www.zigbee.org. Accessed 2011-01-24.

[4] Montenegro, G. et al. (2007). Transmission of IPv6 Packets over IEEE 802.15.4 Networks. http://www.ietf.org/rfc/rfc4944.txt. Accessed 2011-01-24.

[5] Dunkels, A. & Vasseur, J-P. (2010). Interconnecting Smart Objects with IP: The Next Internet. Burlington: Morgan Kaufmann. ISBN: 9780123751652.

[6] Deering, S. et al. (1998). Internet Protocol, Version 6 (IPv6) Specification. http://www.ietf.org/rfc/rfc2460.txt. Accessed 2011-01-24.

[7] Mid Sweden University. Environmental Monitoring. http://www.miun.se/en/Research/Our-Research/Centers-and-Institutes/STC1/Research-within-STC/Wireless-Sensor-Systems/Environmental-monitoring-. Accessed 2011-01-24.

[8] Acreo AB. Acreo. http://www.acreo.se. Accessed 2011-01-24.

[9] Cook, D. & Das, S. (2004). Smart Environments: Technology, Protocols and Applications. Wiley-Interscience. ISBN: 9780471544487.

[10] IEEE. Institute of Electrical and Electronics Engineers. http://www.ieee.org. Accessed 2011-01-24.

[11] IEEE (2006). Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs). http://standards.ieee.org. Accessed 2011-01-24.

[12] ITU. (1994). ITU-T Recommendation X.200. http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.200-199407-I!!PDF-E&type=items. Accessed 2011-01-24.

[13] Bormann, C. & Shelby, Z. (2009). 6LoWPAN: The Wireless Embedded Internet. Chichester, UK: John Wiley & Sons. ISBN: 9780470747995.

[14] Information Sciences Institute University of Southern California. (1981). Internet Protocol DARPA Internet Program Protocol Specification. http://tools.ietf.org/html/rfc791. Accessed 2011-01-24.

[15] Information Sciences Institute University of Southern California. (1981). Transmission Control Protocol DARPA Internet Program Protocol Specification. http://www.ietf.org/rfc/rfc793.txt. Accessed 2011-01-24.

[16] Kozierok, C. M. (2010). The TCP/IP Guide. http://www.tcpipguide.com. Accessed 2011-01-24.

[17] Postel, J. (1980). User Datagram Protocol. http://tools.ietf.org/html/rfc0768. Accessed 2011-01-24.

[18] Fielding, R. (1999). Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html. Accessed 2011-01-24.

[19] Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architecture.

[20] Modbus Organization (2006). The Modbus Organization. http://www.modbus.org. Accessed 2011-01-24.

[21] Modbus Organization (2006). Modbus Application Protocol Specification V1.1b. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf. Accessed 2011-01-24.

[22] Modbus Organization (2006). Modbus over Serial Line Specification and Implementation Guide V1.02. http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf. Accessed 2011-01-24.

[23] SICS. The Contiki Operating System. http://www.sics.se/contiki. Accessed 2011-01-24.

[24] IETF. Internet Engineering Task Force. http://www.ietf.org. Accessed 2011-01-24.

[25] Buettner, M. et al. (2006). X-MAC: A Short Preamble MAC Protocol For Duty-Cycled Wireless Sensor Networks . http://www.cs.colorado.edu/~rhan/Papers/xmac_sensys06.pdf. Accessed 2011-02-14.

[26] TinyOS Alliance. TinyOS. http://www.tinyos.net. Accessed 2011-01-24.

[27] Real Time Engineers Ltd. A FREE real time operating system (RTOS) for small embedded systems. http://www.freertos.org. Accessed 2011-01-24.

[28] Sensinode Ltd. Sensinode. http://www.sensinode.com. Accessed 2011-01-24.

[29] Atmel Corporation. Atmel Corporation. http://www2.atmel.com. Accessed 2011-01-24.

[30] Zolertia. Zolertia. http://zolertia.sourceforge.net. Accessed 2011-01-24.

[31] Atmel Corporation. (2009). ZigBit 2.4 GHz Wireless Modules. http://www.atmel.com/dyn/resources/prod_documents/doc8226.pdf. Accessed 2011-01-24.

[32] Atmel Corporation. (2008). AVR2016: RZRAVEN Hardware User's Guide. http://www.atmel.com/dyn/resources/prod_documents/doc8117.pdf. Accessed 2011-01-24.

[33] Atmel Corporation. (2010). Atmega164A/164PA/324A/324PA/644A/644PA/1284/1284P. http://www.atmel.com/dyn/resources/prod_documents/doc8272.pdf. Accessed 2011-01-24.

[34] Atmel Corporation. (2011). ATmega329P/3290P Summary Preliminary. http://www.atmel.com/dyn/resources/prod_documents/doc8021.pdf. Accessed 2011-01-24.

[35] Atmel Corporation. (2009). AT86RF230. http://www.atmel.com/dyn/resources/prod_documents/doc5131.pdf. Accessed 2011-01-24.

[36] Atmel Corporation. (2006). JTAGICE mkII Quick Start Guide. http://www.atmel.com/dyn/resources/prod_documents/doc2562.pdf. Accessed 2011-01-24.

[37] Canonical Ltd. Ubuntu. http://www.ubuntu.com. Accessed 2011-01-24.

[38] SICS. Jackdaw RNDIS RPL border router. http://www.sics.se/contiki/wiki/index.php/Jackdaw_RNDIS_RPL_border_router. Accessed 2011-04-08.

[39] Litech Systems Design. Linux IPv6 Router Advertisement Daemon. http://www.litech.org/radvd. Accessed 2011-01-24.

[40] The PHP Group. PHP: Hypertext Processor. http://www.php.net. Accessed 2011-01-24.

[41] Oracle. Java EE at a Glance. http://www.oracle.com/technetwork/java/javaee/overview/index.html. Accessed 2011-01-24.

[42] The Computer Language Benchmarks Game. http://shootout.alioth.debian.org/u32/benchmark.php?test=all&lang=java&lang2=csharp. Accessed 2011-01-24.

[43] Oracle. Glassfish Server. http://www.oracle.com/us/products/middleware/application-server/oracle-glassfish-server/index.html. Accessed 2011-01-24.

[44] JBoss Community. JBoss Application Server. http://www.jboss.org/jbossas. Accessed 2011-01-24.

[45] Eclipse Foundation. Eclipse. http://eclipse.org. Accessed 2011-01-24.

[46] Oracle. MySQL:: The worlds most popular open source database. http://www.mysql.com. Accessed 2011-01-24.

[47] phpMyAdmin. phpMyAdmin. http://www.phpmyadmin.net. Accessed 2011-01-24.

[48] Sun Microsystems, Inc. (2009). JAX-RS: Java API for RESTful Web Services v1.1.

[49] Oracle. Java Server Faces. http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html. Accessed 2011-01-24.

[50] Kim, K. et al. (2009). Simple Service Location Protocol (SSLP) for 6LoWPAN. http://tools.ietf.org/html/draft-daniel-6lowpan-sslp-02. Accessed 2011-01-24.

[51] Free Software Foundation, Inc. AVR Downloader/UploaDEr. http://savannah.nongnu.org/projects/avrdude. Accessed 2011-01-24.

[52] Wireshark Foundation. Wireshark. http://www.wireshark.org. Accessed 2011-01-24.

[53] SICS. Tutorial: Running Contiki with uIPv6 and SICSlowpan Support on the Atmel Raven. http://www.sics.se/contiki/tutorials/tutorial-running-contiki-with-uipv6-and-sicslowpan-support-on-the-atmel-raven.html. Accessed 2011-03-08.

Appendix A – Software Tools

This section lists all software tools used in the project.

Cooja

Cooja is an advanced Java-based simulation environment for Contiki operating system. It was used to examine Contiki and test the application code [23].

AVR Studio 4

AVR Studio 4 is an Integrated Development Environment (IDE) that is used to build and debug applications for 8-bit AVR microcontrollers [29].

AVRDude

AVR Downloader/UploaDER is a Linux tool for downloading/uploading firmware and fuses to AVR boards [51]. It was used to program the Ravens and the usb stick.

Eclipse

Eclipse is an Integrated Development Environment, which supports a wide range of languages [45]. In this project it is used for programming in both for C and Java. In Java it is also used to build and run the software, as well as uploading software to the JBoss AS 6 server.

A large number of plugins were used together with Eclipse, most of them are available from the Eclipse repository and are therefore not listed here.

phpMyAdmin

phpMyAdmin is a database management tool. It was used to create, maintain and examine the database [47].

Wireshark

Wireshark is a network protocol analyzer, it was used to inspect and debug network traffic [52].

Appendix B – Node Manual

Soldering

This manuals purpose is to show how the AVR Raven and K30 are connected.

There are a number of IO interfaces available on Raven [hardware guide]. The J202 interface, marked in figure 24, is used to communicate with the K30 sensor. PCB connections J202-1 and J202-2 are positioned at the top of the J202 header and will not be connected externally. These two port pins are used for the internal communication between the two MCUs on Raven, 1284P and 3290P. The next two PCB connections, J202-3 and J202-4, are used for external communication with the K30 sensor. The port pin to the left is PD2, which functions as Rx, and is marked in the figure as *1284P RX*. The port pin to the right is PD3, which functions as Tx and is marked in the figure as *1284P TX*. 1284Ps Rx pin is connected to K30s Tx, marked in the figure as *K30 TX*. The Tx pin of 1284P is connected to the Rx pin of K30, marked as *K30 RX*. PCB connection J202-9, positioned in the lower left corner of the J202 interface, is connected to internal ground, 0V. This port pin, named *1284P G0* in the figure, is connected to the ground pin on K30, *K30 G0*.

Both the Raven and K30 are powered by an external 9V power source. The J401 interface on the Raven is used to connect the board to an external power source. The port pin positioned to the left, J401-1, is connected to external 9V. The next pin, J401-2, is connected to ground. K30 is connected to the same 9V power source, as shown in the figure.

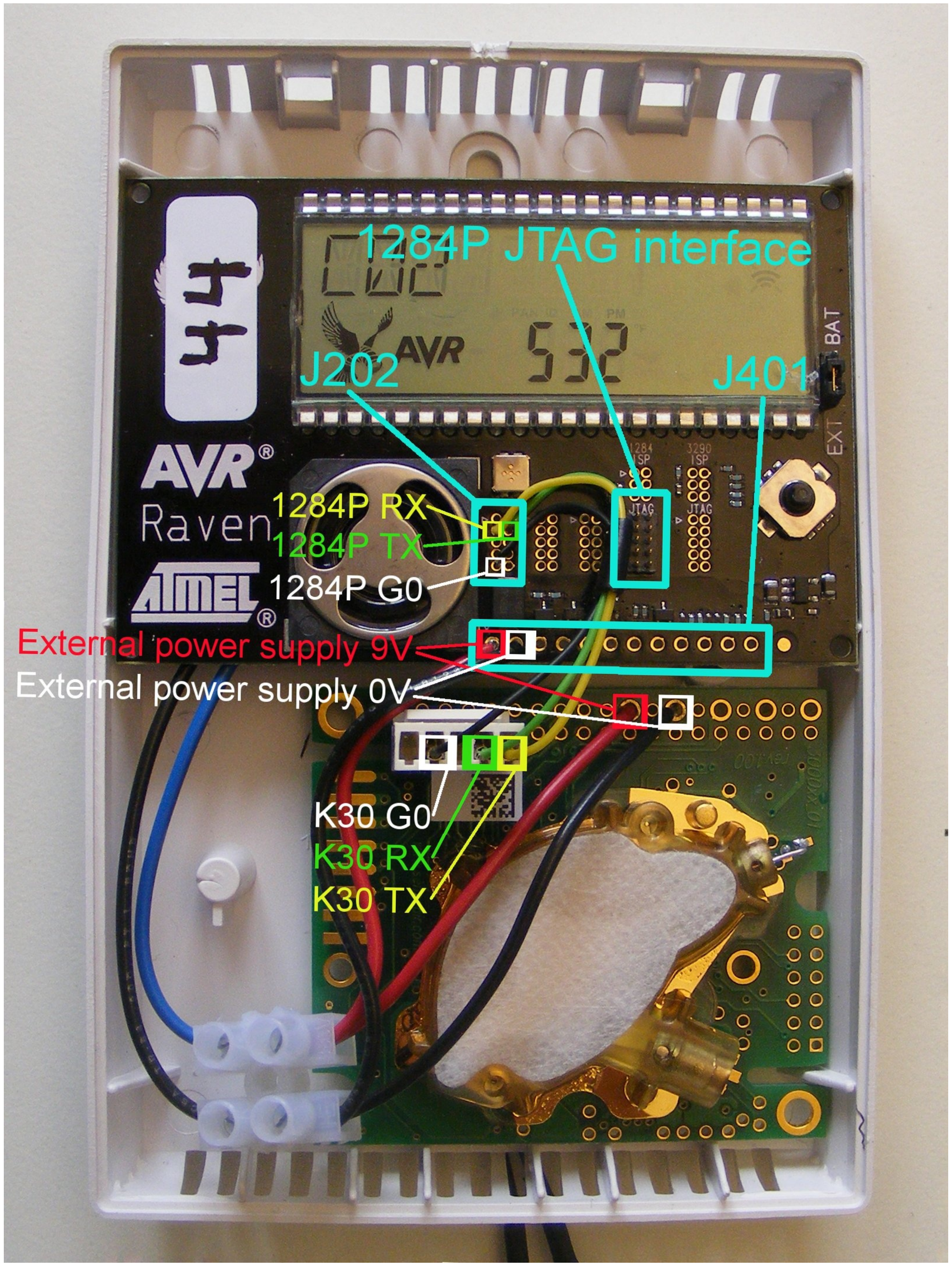


Figure 24: Node soldering map

Programming

Application (1284P)

This programming manual explains the procedures required to program an AVR Raven board with TCP collection code.

To program a node a JTAGICE mkII and a PC running Ubuntu is required [36][37]. AVRdude has to be installed on Ubuntu as well [51].

1. Copy */usr/bin/avr-split* from the project's software directory to the */usr/bin* folder of Ubuntu. If necessary modify permissions to allow for execution.
2. Extract the *contiki* folder from *contiki.tar.gz* to the desired location.
3. Extract the implementation folder *TCP* from *TCP.tar.gz* to the same location as contiki folder.
4. Navigate to following folder *contiki/platform/avr-raven.* Open the file *contiki-raven-main.c*. At row 117 change the last byte of the *mac_address* to a desirable number, this should be unique for each node. The IP address will be created from this MAC address.
5. Connect the JTAGICE mkII to a USB port on the PC.
6. Connect the JTAG of the JTAGICE mkII to the 1284P MCUs JTAG interface on the Raven, as shown in figure 25. The location of JTAG interface of 1284P is shown in figure 24.
7. Use a 9V DC power source to power the node. Turn on the JTAGICE mkII.
8. Open a terminal on the PC and navigate to the location where the application code is located, in the *TCP* folder. **Remember that the contiki folder MUST be located at the same location as the implementation folder.**
9. This command line will delete all the old compilation files.
   ```
   make clean
   ```
10. This command will compile the application code and the Contiki operating system located in the *contiki* folder.
    ```
    make
    ```
11. This command line will split the compilation output file, *TCP.elf*, into *TCP.elf.eep* and *TCP.elf.hex*. The *.eep* and *.hex* files are used to program the Raven board.
    ```
    avr-split TCP.elf
    ```
12. This command line will program the 1284P with the newly compiled application code with correct fuse bits [53].
    ```
    avrdude -pm1284p -cjtag2 -Pusb -u -Uflash:w:TCP.elf.hex:a
    ```

```
-Ueeprom:w:TCP.elf.eep:a -Ulfuse:w:0xE2:m -Uhfuse:w:0x99:m
-Uefuse:w:0xFF:m
```

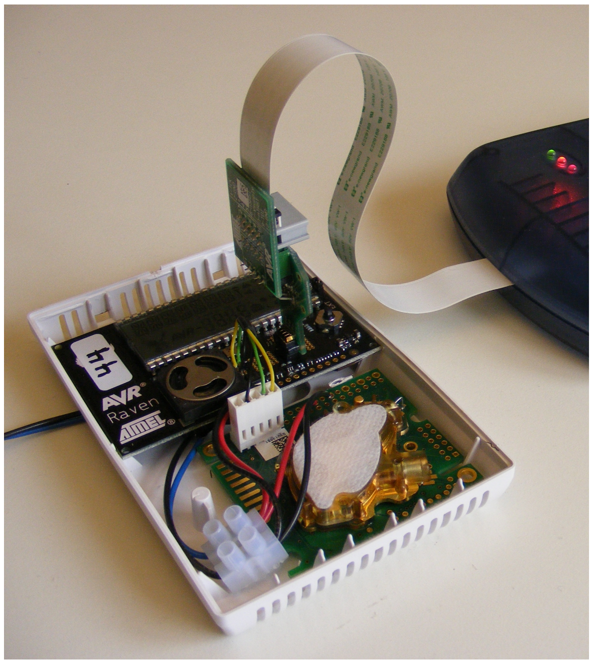


Figure 25: Programming a node

LCD (3290P)

1. Extract the *lcd* folder from *lcd.tar-gz* to the desired location.
2. Connect the JTAGICE mkII to a USB port on the PC.
3. Connect the JTAG of the JTAGICE mkII to the 3290P MCUs JTAG interface on the Raven, as shown in figure 25. Note that the location of JTAG interface of 3290P is shown in figure 26.
4. Open a terminal on the PC and navigate to the *lcd* folder.
   Positioned in the *lcd* folder, run the following commands.
   ```
   make clean
   make
   avr-split ravenlcd_3290.elf
   avrdude -pm3290p -cjtag2 -Pusb -u -Uflash:w:ravenlcd_3290.elf.hex:a
   -Ueeprom:w:ravenlcd_3290.elf.eep:a -Ulfuse:w:0xE2:m -Uhfuse:w:0x99:m
   -Uefuse:w:0xFF:m
   ```

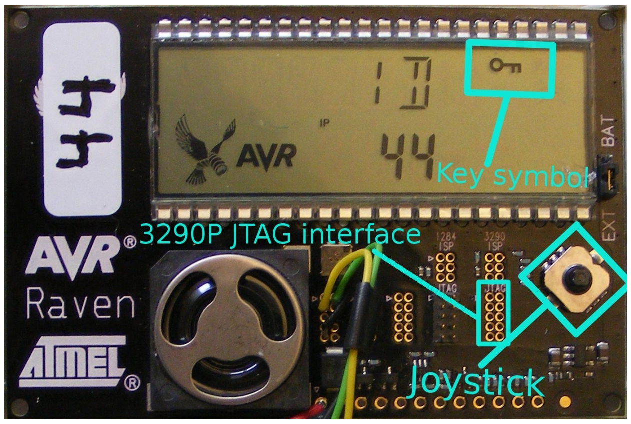


Figure 26: Raven LCD

Operation

This manuals purpose is to aid the user with navigating the LCD on AVR Raven board.

When the node is powered on the 3290P MCU, which controls the LCD, will get the id from the 1284P and show it on the LCD. The id is the last eight bits of the IP address.

When sensor data is fetched from the K30 the LCD will enter a *cycling mode*, where it cycles between displaying CO_2 and the temperature value on the LCD.

AVR Raven has a joystick positioned to the right just below the LCD, marked in figure 26. This joystick is used to navigate in the LCD options menu. The joystick can be moved in four directions, up, down, left and right.

Pressing left, up or down on the joystick when in cycling mode will switch between sensor data types shown on the LCD.

Pressing right on the joystick when in cycling mode enters the *options menu*. The options in options menu are aligned to the right on the LCD and a key symbol is turned on,see figure 26, which shows that user has entered the options menu. Pressing left within the options menu will return to the cycling mode. Up and down sticks are used to navigate the different options. Right stick within options menu will choose an option.

Presently there are five options available.

- CO2 – enters the settings menu for CO_2
    - ON – turns on the CO_2 and returns back to cycling mode.
    - OFF – turns off the CO_2 and returns back to cycling mode.
- TEMP – enters the settings menu for temperature
    - ON – turns on the temperature and returns back to cycling mode.

    - OFF – turns off the temperature and returns back to cycling mode.
- HUM – enters the settings menu for humidity.
    - ON – humidity is not used presently, thus no changes will be made by this setting. Returns back to the cycling mode.
    - OFF – returns back to cycling mode.
- ID – shows the node id on the LCD display, can be seen in figure 26.
- RESET – resets all the settings to default and returns back to cycling mode, although no data will be shown until next sensor data is fetched by 1284P.

Should the node have contact with the server an antenna symbol will be turned on, which indicates this. When the node is not in range to the server or if contact with the server is lost, the antenna symbol will be turned off.

Should something be wrong with the communication between 1284P and the sensor, the cycling mode will stop. Instead OFFLINE will be shown on the display, indicating that something probably is wrong with the sensor communication.

Appendix C – Edge Router Manual

Programming (Jackdaw)

For a general reference on programming the Jackdaw, see the Contiki web page [53].

Before carrying on with this manual, **ensure step one and two of Appendix B – Programming (1284P) has been performed.**

1. Extract the *jackdaw* folder from the *jackdaw.tar.gz* to the same place where contiki folder is located.
2. Open a terminal and navigate to the *jackdaw* folder.
3. Run the following commands:
   ```
   make clean
   make
   avr-split ravenusbstick.elf
   ```
4. Connect the RZUSBSTICK to a USB port on the PC.
5. Connect the JTAGICE mkII to the JTAG interface of the RZUSBSTICK and turn on the JTAGICE.
6. Use the following command line to program the RZUSBSTICK with Jackdaw code and to set correct fuse bits required for the RZUSBSTICK.
   ```
   avrdude -pusb1287 -cjtag2 -Pusb -u -Uflash:w:ravenusbstick.elf.hex:a
   -Ueeprom:w:ravenusbstick.elf.eep:a -Ulfuse:w:0xDE:m -Uhfuse:w:0x99:m
   -Uefuse:w:0xFB:m
   ```

Installation

1. Ensure that Ubuntu is installed as operating system [37].
2. Install radvd by running
   ```
   sudo apt-get radvd
   ```
3. Copy */etc/radvd.conf* from the project's software directory to */etc.*
4. In */etc/sysctl.conf*:
   Uncomment line *net.ipv6.conf.all.forwarding=1*
   Add line *net.ipv6.conf.default.rp_filter=0*
   Add line *net.ipv6.conf.all.rp_filter=0*
5. Copy *netconf.sh* to a suitable location for execution. Typically either your home directory or */usr/bin*.

6. Edit *netconf.sh* if necessary to customize the ipaddress of eth0 and usb0

Operation

1. Boot computer and login
2. Ensure that the Jackdaw is connected to the computer, and that is has been configured by the system by running
   ```
   sudo ifconfig
   ```
   There should be a post under usb0, which should be confirmed as a NIC.
3. Run
   ```
   ./netconf.sh
   ```
4. Ensure that eth0 and usb0 has received global addresses by running
   ```
   sudo ifconfig
   ```
5. Optional: radvd should start on boot up, in case radvd did not start properly run
   ```
   sudo /etc/init.d/radvd restart
   ```

Appendix D – Server Manual

Installation

1. Ensure that Ubuntu is installed as operating system [37].
2. Download and install MySQL database server [46].
3. Create a mysql user *senseuser* with password *sensepass,* create a database *sensebase* and grant all privileges to *senseuser.* Consult MySQL web page for guides and references [46].
4. Import the *sensebase.sql* file found in the project's software directory to the sensebase database.
5. Download and install JBoss AS 6 to */opt/jboss-6.0.0* [44].
6. Copy the contents of */opt/jboss-6.0.0* found in the project's software directory to */opt/jboss-6.0.0* on the server.
7. Edit */opt/jboss-6.0.0/server/default/deploy/mysql-ds.xml* to match your database server configuration.
8. Edit if necessary */opt/jboss-6.0.0/bin/run.conf* to bind the web server to a different IP address. Modify *-Djboss.bind.address* variable.
9. Make *www-data* owner of *jboss-6.0.0* by running
   ```
   sudo chown -R www-data:www-data /opt/jboss-6.0.0
   ```
10. Add your user to the group *www-data.* This is done easiest through the OS GUI, consult Ubuntu documentation.
11. Grant full access to *jboss-6.0.0* to the group *www-data* by running
    ```
    sudo chmod g+rwx /opt/jboss-6.0.0
    ```
12. Copy the contents of */home/senseair/sas* found in the project's software directory to */home/{user}/sas* on the server, where {user} is the home folder of the user.
13. Edit if necessary */home/{user}/sas/jndi.properties* to the IPv4 address of the web server.

Operation

This user manual assumes access to the Raven cards and the PC edge router used in the project. The Ravens are preprogrammed with the TCP implementation, as such only the TCP version can easily be run.

Before attempting to start the server, ensure that the edge router is up and running.

1. Boot the PC and log on to the Ubuntu operating system with user/password : senseair/sensepass
2. First the web server must be started. In the same terminal run.
   ```
   ./webstart.sh
   ```
   Wait for the server to report it has started. This process might take several minutes!
3. If the sensenet application has not previously been loaded on to the server, do so now by navigating to the administration console, http://192.168.2.131:8080/admin-console. Login using admin/admin as user/pass. Browse to *Web Application (WAR)s,* choose *Add resource* and follow the guide, sensenet.war is the file to be uploaded. This step is only necessary if it's a fresh server install or if the sensenet application was explicitly removed from the server. The server will retain the application between reboots, hence this step is not necessary between simple reboots.
4. Next the TCP server must be started. Open a new terminal, and run
   ```
   ./tcpstart.sh
   ```
   Wait for the server to report it has started. This should take only a few seconds. This step will fail if the web server has not started properly, or has not finished booting!
5. Connect all nodes to power
6. Open a web browser on the PC and navigate to http://192.168.2.131:8080/sensenet/. This web page should be available from other computers as well, assuming there is a functional IPv6 network connecting them.
   At http://192.168.2.131:8080/sensenet/faces/auto.html one can find a page which auto refreshes itself, though on this page the date interval is locked.