Programming Disconnected Operations in Wireless Sensor Networks

CHRISTOPHER OLSSON

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2009

TRITA-ICT-EX-2009:220

Programming Disconnected Operations in Wireless Sensor Networks

Masters Thesis

Christopher Olsson
cholss@kth.se

2009-12-09

Examiner:
Professor Gerald Q. Maguire Jr.
Royal Institute of Technology (KTH)

Supervisor:
Luca Mottola, Ph.D.
Swedish Institute of Computer Science (SICS)

Abstract

Wireless sensor networks, networks of nodes communicating wirelessly with sensing capabilities, are becoming more popular and are utilized by an increasing number of applications. Some wireless sensor networks are implemented because the usual network solutions of an always connected network could not be applied. Specifically this thesis is concerned with the case when the connection between the end-user and the network is not always available, i.e., there is only intermittent connectivity.

This masters thesis gives an introduction and provides some background knowledge concerning wireless sensor networks, specifically focusing on disconnected operation. A set of building blocks will be presented to help programmers deal with programming disconnected operations. Examples to demonstrate our solution is implemented as shell commands using the Contiki operating system. Our solution was tested in the field and compared against a common, monolithic, programming approach. This practical example shows the potential significance of this thesis project in real world applications and allowed an evaluation of both the qualitative and quantitative aspects of our solution. The results of our evaluation prove that our solution offers an easier interface for the programmer to work with at the cost of possible less memory space.

Sammanfattning

Trådlösa sensornätverk, nätverk med noder som kommunicerar trådlöst och har sensorer, blir mer populära och används av i ett ökande antal applikationer. Några trådlösa sensornätverk används för att en vanlig nätverkslösning med ständigt uppkopplade noder inte går att genomföra. Det här examensjobbet är specifikt inriktat på fall när en uppkoppling mellan slutanvändaren och nätverket inte alltid är tillgängligt, t.ex. när det bara är tillfällig uppkoppling.

Detta examensarbete ger en introduktion och bakgrund till trådlösa sensornätverk med fokus på programmering av frånkopplade operationer. Ett antal byggstenar har tagits fram för att hjälpa programmerare att programmera frånkopplade operationer. Exempel för att styrka vår lösning i vår rapport kommer att implementeras som shell-kommandon i operativsystemet Contiki. Vår lösning kommer att testas i verkligheten och jämföras med ett vanligt, monolitisk, programmeringsangreppssätt. Detta praktiska exempel kommer visa den potentiella nyttan av detta examensarbete i verkliga applikationer och tillåta utvärdering av kvalitativa och kvantitativa aspekter på vår lösning. Resultaten från vår utvärdering bevisar att vår lösning erbjuder ett enkelt gränssnitt för programmeraren att arbeta med till en kostnad av möjligen mindre minnesplats.

Table of Contents

List of figures

List of tables

Abbreviations

BTS	Base-station Transceiver
CLI	Command Line Interface
DARPA	(U.S.) Defence Advanced Research Projects Agency
DTN	Delay Tolerant Network
DTNRG	Delay-Tolerant Networking Research Group
FIFO	First In, First Out
HW	Hardware
LIFO	Last In, First Out
LOC	Lines Of Code
MEMS	Micro Electro-Mechanical System
OS	Operating System
RAM	Random Access Memory
RTOS	Real Time Operating Systems
ROM	Read-Only Memory
WSN	Wireless Sensor Network

1 Introduction

A wireless sensor network (WSN) [1] is a wireless network consisting of sensor devices, deployed to monitor physical or environmental conditions. The network nodes, in WSNs called motes, are characterised by having constrained resources, often deployed on a large scale, should have low production cost, be adaptable to environmental changes, and operate unattended (i.e., autonomously). The network must adapt to changes, such as the loss of nodes or obstacles in the terrain attenuating or blocking transmissions. Researchers have in the past defined a wireless sensor network as *"large-scale ad hoc, multihop, unpartitioned, network of largely homogenous tiny, resource-constrained, mostly immobile sensor nodes that would be randomly deployed in the area of interest"* [2]. While this is true for most applications, there are those that do not follow this definition, but are still considered WSNs. An example of the use of a traditional WSN is monitoring battlefields with sensors used as sentries, to provide alerts whenever an enemy is moving in its vicinity [3]; where motes are fully connected with each other and the end-user. Other less classical networks are GlacNet [4] and a project in the Baltic Sea's Bothnian Bay [5] that both are neither large-scale *ad hoc* networks nor have the motes been randomly deployed.

When a connection from the end-user to the network cannot always be sustained it is considered "disconnected" during this time. In the last two scenarios mentioned above programming a network that is to operate over a temporary disconnected medium is called programming disconnected operations.

Due to the advantages and possibilities of WSNs there is a wide range of applications where WSNs can be used. They are most commonly used in military applications and environmental studies, e.g. deploying sensors over a battlefield to detect enemy intrusion instead of using landmines [3] or measuring environmental changes in bodies of water [5,6]. Other common application areas where WSNs are used are: monitoring homes, monitoring vehicle traffic, monitoring people's health, monitoring wildlife habitats, surveillance, disaster discovery, mobile entertainment, home automation, security, and lots more. In the future we expect WSNs to be used in cooperating smart everyday things.

A typical mote consists of a transceiver unit, processing unit, power supply, and one or more sensors; as shown in Figure 1. Depending on the application there might be extended storage space (flash memory), a power generating unit (e.g. a solar cell), or additional hardware that is related to the requirements of the application.

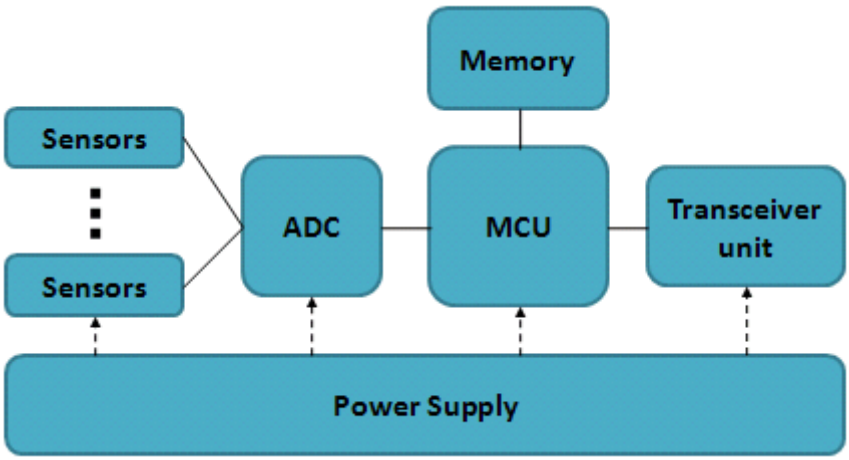

Figure 1: The components of a mote; sensors, power supply, memory, ADC (analog-to-digital converter), MCU (Micro Controller Unit), and transceiver unit.

The deployment of motes can either be done one at a time (carefully positioned at a specific location, e.g. inside machines to monitor temperature and hardware failure) or at random (scattered over a large area for instance from an airplane or a cannon). The flexibility in deployment methods has large advantages over wired sensor networks and also broadens the application spectrum. There are also cases where the environment prohibits the use of a wired sensor network or where it is not feasible to use wired sensor nodes, e.g. when motes are attached to moving objects or deployed in large numbers over a large area. In this setting it is impossible to deploy and wire all nodes. ARGO [6] is such an example, a WSN with thousands of motes floating all over the world's seas following currents and communicating wirelessly.

The majority of WSNs today use a radio link to communicate [7], but light (an example is given in [8]) or even sound is sometimes used. While radio is not the least power consuming method [2], it is the most convenient because of its wide range of use. On a typical mote the radio is the most power consuming subsystem, and since power is the scarcest resource of the node, power usage by the radio should be kept at a minimum. This has lead to research efforts concerning how to minimize power consumption. This research will continue in the future as the volume to store energy is getting smaller due to technology advances. Better power utilisation will make it possible to make *really* small motes. For example, "Smart dust" motes are only a few cubic millimetres in volume [8].

An approach to save energy is to use aggregation [9]. Another solution is to compress the data [10] at the node before transmission since processing instructions uses less power than transmitting additional bytes over a radio link. Another method is letting the nodes power down their radios during periods of time when there is no need to transmit data. It should be noted that this last approach also causes the node to be disconnected for the period of time that its radio is powered down.

Technology advances will result in smaller nodes, higher processing capabilities, and greater memory resources. The hardware challenge to produce these devices is to keep their power usage and manufacturing cost at a minimum. The software should adapt algorithms and protocols to save energy, increase robustness, enable self configuration, increase fault tolerance, and ease usage. There is also a need to simplify the implementation, maintenance, and the user interface to the network.

While in the future it might be possible to use technologies that allow us to send data over greater distances and at lower energy costs to sustain a connection where it is currently impossible, we might never be able to create a fully reliable connection when we are monitoring nodes in a changing environment. For this reason this thesis topic is not only of interest today, but also in the future.

1.1 Problem statement

Programming disconnected operations today is a problem due to the very limited number of abstractions provided to programmers. In this thesis, an application driven approach will be proposed. The proposed solution should be applicable as a general solution. We will identify, implement, and evaluate a set of building block functionality to simplify programming disconnected WSN applications. The results will be tested through programming example software and also used in a real life application. It is hoped that this set of building blocks will increase the interest in developing WSN applications for as wide a variety of scenarios as possible, specifically including those that must support disconnected operation.

With the increased interest of WSNs it is important to look into new ways to further expand the use of WSNs. However, the desired characteristics of a WSN change with the specific requirements that an application has. This thesis is specifically concerned with situations when implementing a WSN *cannot* be done without considering disconnected operations, in particular when connections between the end-user and the WSN cannot be sustained.

Disconnected operation could be due to limited power when the communication distance is so great that it requires a large amount of energy to keep the connection alive or when the power needed to send a large amount of data exceeds the nodes' power resources. The project "Sensor Networks to Monitor Marine environment with Particular Focus on Climate Changes" [5, 11] by SICS and partners is an example of disconnected operation due to the first of these limits (high power consumption due to large distance between the end-user and the senor node). Another reason for periodically closing the connection could be because the end-user's control station cannot remain within range of the WSN. For example, when measuring volcano activity, as done by researchers at Harvard's Sensor Networks lab [12]; although the WSN must be expendable in the case of an eruption, they cannot afford to put the base station too close to the volcano because of the risk losing the equipment and data. Over such long distances it might be wise to shut down the connection to save energy when nodes are not experiencing any volcanic activity.

A problem in these scenarios is developing the disconnected WSN applications, as the connection between the end-user and WSN cannot be sustained. Most work that has addressed the problem of disconnected operations has been based upon application specific solution, while this thesis seeks to develop a solution that can be applied to many different applications. To keep the interest for new applications growing it is crucial that WSNs be portable to as many different scenarios as possible.

1.2 Method

The main goal in this thesis is to provide a solution to help programmers program disconnected operations for WSNs. The solution will be general so it can be adapted to more environments that examined in this thesis. By targeting disconnected operations this thesis will contribute to the WSN community in a unique way -- not seen up to this point.

The solution presented in this thesis will be implemented and tested against another possible solution for programming disconnected operations. Looking at both qualitative and quantitative aspects of our solution will be a part of our analysis and together with live testing will show the significance of the proposed solution in real applications.

1.3 Thesis structure

This thesis begins with background literature study (chapter 0) where the fundamentals of WSNs and its software are described.

Chapter 3 will take the reader deeper into the issues of disconnected scenarios and programming. While chapter 4 presents a new model for categorizing disconnected WSNs and shows the effects of different parameters.

In chapter 5 the design of our solution will be presented and later how it will be implemented. This is followed by evaluating the solution in chapter 6, looking at both qualitative and quantitative aspects.

Conclusions and results are presented in chapter 7, together with a description of future work.

2 Background

In this chapter the background for this thesis will be presented, including a look at some specific applications and software of current interest (such as operating systems and programming abstractions).

2.1 Background on WSN applications

WSNs have found their way into a wide variety of application areas. The most common are military, environmental, health, and other commercial applications. Applications using these WSNs can be used for monitoring, surveillance, targeting, damage assessment, disaster detection, tracking, automation, etc.

2.1.1 Traditional WSN applications

Up to this point defining a traditional WSN has been difficult since almost all WSN applications have been unique applications. Some WSNs have nodes distributed in great numbers, others only a couple nodes; some nodes are randomly distributed, other carefully positioned at exact locations; etc. Since this thesis focuses on disconnection, a traditional WSN is considered as one that does not deal with the issues due to disconnection.

A traditional WSN application is a military application using sensors in cooperation with mines [3]. In this application sensor are placed on a battlefield, near the ground, always connected to each other - so that if one node is removed the end-user is alerted. In this application power efficiency is very important since it takes more energy to send data from nodes that cannot have an antenna placed high above the ground, thus radio communication requires more energy when nodes are at ground level. By communicating between the nodes it is possible for other nodes to take action when an enemy tampers with a real mine to create a breach lane, e.g. a mine nearby the breach could use rocket thrusters to move into the breach lane, filling the gap of the disabled mine. This is discussed as a feature of the Self Healing Minefields application mentioned in [3].

Another typical application is monitoring seabird nesting and behaviour at Great Duck Island [13]. To identify important areas of further work, a WSN application was used to avoid potential impacts of human presence on the animals being monitored. Nodes could be placed in nests *before* the breeding season when the island was inhabited to later measure data about nesting birds. Motes send data to a base station that is accessed through the internet via a two-way satellite link. In their implementation the mote data is always accessible, hence there is no need to approach the deployment site until after the breeding season when birds are no longer nesting. For operating “off the grid” this report discusses the use of disconnected operations as a possible solution, but did not implement such a solution.

2.1.2 Disconnected WSN applications

Scenarios for disconnected applications are commonly found in environmental monitoring, such as monitoring moving animals or geological activities at remote and hazardous locations. Usually the area that the motes need to cover is so great that it is inefficient from an energy resource and hardware cost perspective to maintain a constant connection. In general, an application has to be prepared for disconnected operation when monitoring the behaviour of moving objects (which have motes attached to or incorporated in them). Consider the case of monitoring the movements of cars [14]. Cars could act as carriers for the

nodes - utilizing energy from the car's batteries or generator. Since it may be difficult or uneconomical to provide wireless coverage of the entire road grid, the cars may continuously or periodically make measurements when driving around – but only transfer this data to the end-user when passing fixed base stations, for example, the base stations could be affixed to traffic lights or street lights as these sites also have electrical power available to them and might even have network connectivity. Similarly, passing cars can be used to gather data from sensors monitor structures, e.g. bridges [15].

The ZebraNet project [16] is another example of mobility in WSN, in this case by using a mobile control-station to gather data from nodes attached to zebras. Researchers follow the herd in vehicles to collect data. As there is not a fixed location to send data to, a flooding protocol is used for short range connectivity together with a direct-connection protocol for long range. Gathered data is flooded to nearby neighbours, thus the base-station only needs to come in contact with a few nodes to collect data. To improve this further the direct-connection protocol is used for long distance radio links -- when limited bandwidth and limited storage space at the node does not noticeable affect the efficiency of the protocol due to the small amount of data being sent. This optimizes energy consumption. The connection is set up by nodes only during the day, when nodes search for a control-station. If a control-station is nearby, then data is uploaded. Since the researchers themselves determine when to collect data, they control the disconnection between the control-station and the nodes.

David Jea, et al. [17] propose to use so called "data mules", mobile elements moving around in the vicinity of the deployment area, to collect data from nodes. This concept of data mules is seen in other projects, such as ZebraNet. This method addresses both bottlenecks at access points (hotspots) and data load balancing, but presumes the possibility of having a mobile element, which is not always the case in environmental monitoring. It has also high latency, might not be sufficiently scalable, and for many applications does not fully address the problems related to disconnection, although it gives a general solution to handle disconnected operations in many cases.

Another example of mobility in WSNs is when monitoring whale activity [18]. Several control-stations are placed along paths that whales travel or at regularly visited feeding grounds. When the whale surfaces and exposes their radio antenna data is uploaded to a nearby Shared Wireless Infostation Model (SWIM) control-station. Data is shared among whales so that only a single whale needs to access a SWIM control-station and when a whale uploads its data the previously stored copy is erased. A timestamp is used to determine when data on a whale's memory storage has become obsolete. SWIM stations can work together as an *ad hoc* network to transfer data to shore or use a satellite connection, whenever a satellite passes over the area. This network topology is "multi disconnected": (1) the nodes on the whales are disconnected from each other and the control-station and (2) the SWIM control-station could be disconnected from the end-user on shore if a satellite is used to gather data from the control-station, since the satellite might not always be in such a position that a connection can be established.

A project that both has mobile nodes and is deployed over a large area is the ARGO project [6]. Over 3000 floats (floating motes) are deployed in the world's seas (Figure 2) measuring the temperature and salinity in the water down to a depth of 2000m.

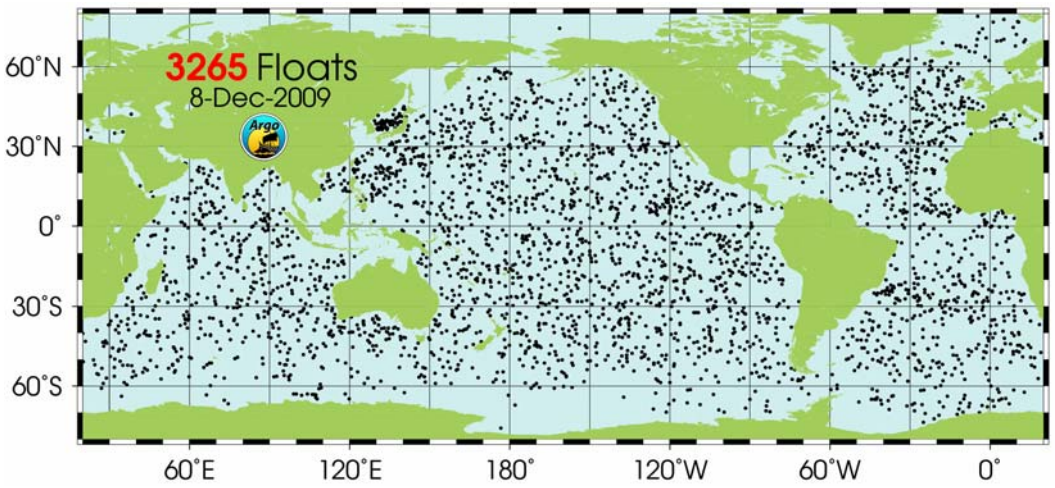

Figure 2: The ARGO project with nodes deployed (updated 4-Mar-2009)
Source: www.argo.ucsd.edu

By using a satellite tracking system it is possible to measure the velocity of the ocean current by looking at the movement of the floats. Most of the time floats are underneath the surface, typically for a 10-day period, without the possibility to connect to the satellite. When a float surfaces it will transmit data for 6-12 hours, then start another dive. All of the floats are disconnected from each other and only communicate via a satellite to the end-user.

As a result of the high cost of monitoring remote areas, especially bodies of water, such environments are poorly monitored. A solution to the problem of high cost and rarely gathered results is to use sensor networks as proposed in [5, 11], deploying buoys in the Baltic Sea (Figure 3). The buoys are anchored with a line to the sea bottom. On this line a bin containing several nodes will travel, measuring data at specific depths as determined by a pressure sensor. Light and acceleration sensors are attached on top of the boy. For powering the system rechargeable batteries are used. The batteries are recharged by both solar cells and a wave energy generator.

The potentially large amount of measurement data requires more storage space than what the node's RAM can offer. For this application flash memory cards are used. To minimize the energy-cost for transmitting data, the data is compressed before transmission. Transmission is planned to be done by GPRS which has a long range, but at the cost of high energy consumption. Additionally nodes can only be placed near the coast where there is GSM coverage. To further minimize energy consumption the GPRS link between the buoy and control station is disconnected when transmission is completed and remains disconnected during measurements.

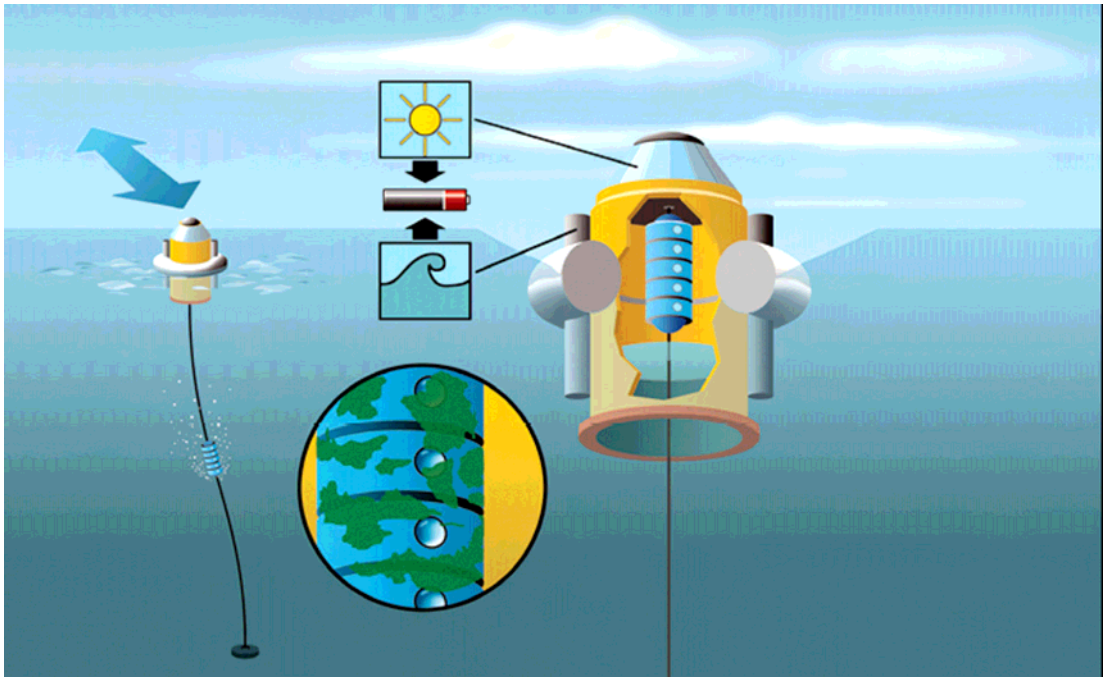

Figure 3: Sensor system for remote water monitoring (exploiting solar and wave powered recharging of the on-board batteries).
Illustration: Bo Reinerdahl

An application where nodes are not mobile, but still may be disconnected is monitoring glacier movement [3]. In this example the WSN is often disconnected from the end-user due to the distance and medium between nodes, base-station, and end-user. When monitoring glaciers the nodes do not communicate with each other and only temporarily communicate with a base station above the glacier on a scheduled basis. This is because the density of the ice reduces signal propagation. Synchronization of the nodes to a schedule is done by using GPS timing. Note that the end-user might always be able to access the base-station through a satellite connection.

2.2 System software

Software development for WSNs occurs primarily on the operating system level or the application layer, or between them using middleware. Middleware is used to connect applications with operating system (OS) and network services [19]. Operating systems differ extensively from OSs for personal computers, but share functionality and needs to a great extent with OSs for embedded systems. When programming applications for WSNs many challenges, such as those mentioned in section 3.2, are a constant concern.

2.2.1 Operating systems

Operating systems for WSNs often needs to deal with resource constrained devices. A typical sensor node is equipped with an 8-bit microcontroller, on the order of 100 Kbytes for code memory, and up to 20 Kbytes of RAM. Such a device has limited processing power, memory, and available energy. The number of different devices is increasing and the application specific nature of sensor networks contributes to this diversity. Therefore, it is desirable that the OS is portable across hardware platforms. Other desired features of the OS are that it should use little code space, support reprogramming of the nodes (during run-time), use little memory allocation, utilize energy efficient algorithms, and in general be efficient about using its resources. Creating an implementation that meets *all* of these requirements

can be quite difficult; therefore it is common that OSs specialize in certain application areas in order to meet some of the requirements very well at the cost of not meeting others.

One way to minimize memory resources is to use an event-driven kernel. A multi-threaded model, in comparison, consumes larger amounts of memory and every thread must have its own stack in memory. Since the system does not know how much stack space the thread needs when memory is allocated at thread creation the stack is over provisioned. On the other hand while event-driven systems work well in sensor network applications; the event-driven programming model can be difficult to manage for programmers. Code generation could facilitate programming such event-driven applications [20].

Examples of potential OSs for wireless network devices are:

- **Contiki** - Light weight event-driven OS that targets WSNs, provides power profiling, dynamic programming, and node shell command line interface (CLI)[21]
- **eCos** – A fully pre-emptive real time operating system (RTOS) running one process with multiple threads, supporting many different platforms [22]
- **LiteOS** – UNIX like OS that specifically targets WSNs, provides shell command line interface, multi-threaded kernel, hierarchical file system, dynamic programming, and online debugging [23]
- **Magnet OS** – Single system image (Java virtual machine), power-aware algorithms for component distribution, statically partitions applications and dynamically places them on the network [24]
- **Mantis –** Multi-threaded OS that targets WSNs, designed to deal with complex tasks such as compression, aggregation, and signal processing; but still be lightweight [25]
- **MicroC/OS-II** – Priority based pre-emptive RTOS with multitasking kernel, performing scheduling with tasking aware interrupt service routines [26]
- **Nano-RK –** Pre-emptive reservation based RTOS. Supports fixed-priority fully pre-emptive scheduler and reservation on system resource consumption [27]
- **Nemesis –** Designed to distribute multimedia applications. It has few hardware abstractions and uses shared libraries resulting in a small kernel [28]
- **NutOS –** Non pre-emptive multithreaded RTOS that featuring events and dynamic heap memory allocation. Uses libraries instead of a fixed kernel block, thus reducing the footprint [29]
- **Tiny OS –** Light weight event-driven OS that targets WSNs, completely non-blocking with a single stack [30]

The hardware of a WSN mote and normal embedded devices are similar allowing OSs for embedded devices to be run on WSN devices; although WSN devices do not necessarily require the real-time properties which are normal in an OS for embedded devices. Three of these OSs are described in the following sections. These examples were chosen because Contiki was developed at SICS and was selected earlier for the project, LiteOS is a UNIX like OS - with a similar user interface to Contiki, and TinyOS is widely used and has expanded into many different areas.

2.2.1.1 Contiki

Contiki [21] was created as a light weight OS for constrained networked embedded systems. While the kernel is event-driven, the OS supports pre-emptive multi-threading, thus benefitting from both event-driven systems and pre-emptive multi-threading. The multi-threading is implemented as an optional application library that can be linked with

programs that explicitly require it. The type of thread used in Contiki has been called a protothread [31].

For reprogramming nodes during run-time, Contiki uses ELF-files for dynamic linking. This allows loading and unloading of individual applications or services during runtime [32].

In Contiki, the kernel provides a minimum of abstractions. Instead abstractions are provided by libraries that have nearly full access to the underlying hardware. Contiki is designed to be portable and has been ported to 14 different platforms and 5 different CPU architectures [33].

To interact with a network node a command-line shell, running on the nodes, can be used. This shell offers UNIX-like commands.

Software-based power profiling is another feature of Contiki. This allows the programmer to measure the energy consumption of a node. Nodes in WSNs seldom have hardware mechanisms for measuring their energy consumption; this is to keep down the cost of the node since a large number of them might have to be produced. By implementing a software-based on-line energy estimation tool, it is possible to estimate the node's energy consumption. This a valuable tool when building a WSN and when evaluating different energy saving algorithms and communication protocols. While the results from this tool are close to reality, they are still estimations. The possibility to see differences and trends in energy consumption for different ways of using of the node and for different algorithms is very valuable. Extensive related work in this area has been done by Tajana Simunic Rosing at Department of Computer Science and Engineering, University of California, San Diego [34].

2.2.1.2 LiteOS

To fulfil the need of an easy to use an OS with a well known interface LiteOS [23] was created, giving programmers a familiar programming environment. LiteOS features a subset of UNIX-like commands, hierarchical UNIX-like local file system, kernel support for dynamic loading and native execution for multithreaded applications, online debugging, and dynamic memory.

The LiteOS architecture can be divided into three categories: kernel, file system, and user environment. The kernel in LiteOS is thread-based, but allows events in user applications using call-back functions. Scheduling is either round-robin or priority based. LiteOS supports dynamic loading and unloading of user applications. All applications are compiled into a modified HEX format; lhex, that is very small in size. Reprogramming is done on the application level using lhex-files.

The file system used in LiteOS is called LiteFS and is similar to the UNIX file system in how it represents different entities, such as data, application binaries, and device drivers. A radio, sensor, and LED are examples of supported device drivers. Read/write operations are mapped directly to hardware, e.g. writing a message to the radio would broadcast it.

The LiteOS user environment consists of a UNIX-like command line interface to sensor nodes. The shell, called LiteShell, runs on the end-user's PC providing a front-end that interacts with the user. The nodes receive translated messages as compressed tokens that are easy to parse. This approach minimizes the run-time footprint on the nodes, while allowing the shell to be easily extended.

2.2.1.3 TinyOS

TinyOS [30] is a tiny embedded operating system targeting WSNs. The entire operating system requires only 226 bytes of RAM and 3.5KB of instruction memory. TinyOS started out as a project at University of California, Berkeley in corporation with Intel Research, as a project in the DARPA NEST program (Smart Dust [8]). It is one of the first OSs targeting WSNs and was developed to cope with technological advances in integrated, low-power, CMOS communication devices and sensors.

The philosophy behind TinyOS is to get the work done as quickly as possible, and then go to sleep as soon as possible. TinyOS is interrupt driven and uses an event-driven architecture so that high concurrency can be handled in a very small amount of memory space. Scheduling is implemented through a two-level structure; short hardware events are preformed immediately while longer applications are run as tasks. The tasks can be pre-empted by events, are time flexible, and run in FIFO order, in comparison to events that are time-critical and use first-in first-out (FIFO) semantics.

Programs are built out of software components that communicate with each other via interfaces. A component consists of four parts: command-handlers, event-handlers, a fixed-size memory segment frame, and tasks. Tasks, commands, and handlers execute in the context of frames.

Since frames are of fixed size, static memory allocation enables the allocation size to be known at compile time, reducing memory allocation overhead. Memory is organized as a single shared stack, no heap, and no function pointers.

2.2.2 Programming abstractions

Programming WSN applications differs very little from programming other embedded hardware, as the fact that OSs can be used in both environments proves. As noted earlier, real-time properties are often required in embedded systems, but are not always required in WSNs.

On top of the operating systems, additional abstractions are often provided to facilitate programming and to hide complexity. Examples of such an abstraction are TinyDB [35] for TinyOS, implemented as a library. TinyDB handles cooperative data acquisition using macroprogramming. Conversely, in node-centric programming the focus is on an individual node rather than the aggregate system. The programmer writes code for each node which may enable the executing code to be more efficient, but is more time-consuming, error prone, and requires significant expertise in the area. Macroprogramming offers a more generic solution by way of high-level programming, by utilizing suitable programming abstractions [36].

To support such programming abstractions, OS-specific programming languages are often developed, such as NesC [37] for TinyOS, which is an extension of the C-programming language. Another WSN-specific programming language, but one not bound to an OS, as it is platform independent, is SPIDEY [38].

2.2.2.1 TinyDB

TinyDB is a system that gathers data via queries from nodes running TinyOS. Instead of writing embedded C-code for each node, TinyDB offer a SQL-like interface together with additional parameters to specify the data to be extracted. Given a query TinyDB collects the data, filters it, aggregates it, and routes it to the end-user.

The goal of TinyDB is to make programming a WSN application significantly easier. Specifically, it allows data-driven applications to be developed and deployed quicker than what was previously possible. Some of the features provided in TinyDB are: metadata management through a metadata catalog, high level queries with a declarative query language, network topology tracking, multiple queries, and incremental deployment via query sharing that allows for nodes with TinyDB to be added to the network at a later stage.

```
SELECT roomno, AVG(light), AVG(volume)
FROM sensors
GROUP BY roomno
HAVING AVG(light) > I AND AVG(volume) > v
EPOCH DURATION 5min
```

Example 1: Query for TinyDB

In Example 1 sensor nodes monitor rooms in a building. The code is used to learn which rooms are in use; by defining a room as being 'in use' as having an average amount of light greater than a threshold and an average volume larger than *v*. This will be checked every fifth minute.

2.2.2.2 SPIDEY

The programming language SPIDEY was created to meet the new challenges emerging in decentralized architectures, where several data sinks might be used to collect data. SPIDEY offers a high-level, application oriented way of defining logical neighbourhoods. The neighbourhoods can be defined declaratively based on the type of nodes, together with requirements about the cost of communication.

SPIDEY was conceived to be an extension of an existing programming language, making it available to a wide range of OSs. Programming in SPIDEY revolves around two concepts: nodes and neighbourhoods, code examples for each concept can be seen in Example 2.

```
node template Device
  static  Function
  static  Type
  static  Location
  dynamic Reading
  dynamic BatteryPower

create node ts from Device
  Function as "sensor"
  Type as "temperature"
  Location as "rooml"
  Reading as getTempReading()
  BatteryPower as getBatteryPower()
```

```
neighborhood template HighTempSens(threshold)
  with Function = "sensor" and
       Type = "temperature" and
       Reading > threshold

create neighborhood hts100
  from HighTempSens(threshold : 100)
  max hops 2
  credits 30
```

Example 2: SPIDEY code for a logical node (top) and logical neighborhood (bottom).

A node is defined by a node template. This template is later used to derive actual instances of logical nodes. A static declaration represents information that does not vary in time; unlike

dynamic information, such as sensed data. A node is created by binding attributes to constant values or functions.

A neighbourhood is first defined as a template, which defines the nodes belonging to the neighbourhood. The logical neighbourhood is then instantiated by specifying where and how it is supposed to be constructed and maintained. In the bottom code example of Example 2 a neighbourhood template is defined as consisting of temperature sensors whose reading is above a given threshold, then instantiated in terms of nodes that meet or exceeds a threshold of 100, are a maximum of 2 hops away, and for which communication requires spending less than a maximum of 30 "credits". These credits represent the communication cost – each node has a function to calculate this cost.

No matter what OS is used and which abstraction that is used to implement the application, there might be several reasons that changes need to be made to the programming *after* deployment. Reprogramming of the network can therefore be expected in most applications.

2.2.3 Reprogramming

From the perspective of constrained resource programming, WSNs and other embedded devices share many similarities since it is important not to waste energy or memory space as both could significantly shorten the application's lifetime. Even though the first step of programming is important and builds a foundation on top of which changes might be made, it is important that the system offers reprogramming in an as easy to use way as possible [39].

There are different methods for reprogramming nodes: full system image replacement, changing different small parts of the binary image, passing interpreted code, virtual machines, and loadable native code modules. One challenge of reprogramming is to keep the energy cost as low as possible during the reprogramming phase, both by minimising the number of bytes sent over the network and minimizing the effort it takes to process the data at each node. Virtual machines have the advantage over native code that the code can be smaller, but the drawback is usually increased energy spent on interpreting the code, especially for long running programs [21]. In some scenarios using long running programs the run-time processing eats up the energy saved by the smaller code. Replacing the full system image is an easy and straight forward approach, but has a high overhead. Editing parts of the binary image during run-time works well for networks where all nodes run the same binary image, but can easily become complicated if they do not. With loadable modules only parts of the system need to be modified, but this method requires support from the OS. The best reprogramming solution would be to use a mix of a virtual machine code and native code, giving good energy efficiency [32].

After setting up a WSN there might be several reasons why a software update is needed. There might be problems with bugs in the software that were not discovered before deployment, or the network needs some new functionality. Since it might not be plausible or even feasible to collect all nodes in order to reprogram them; thus it is in the interest of the programmer that the nodes support reprogramming during run-time.

2.3 Related work

Most related work concerning disconnected operations concerns disconnections within the WSN. However, a small amount of work has been done when the disconnection occurs between the end-user (base-station) and the rest of the WSN. In particular, some work has been done regarding data management [14] [40] and storage [41]. Other work is more related

to networking within the WSN, for example DARPA's Disruption Tolerant Networking program [42], other DARPA funded DTN programs, and research by the Delay-Tolerant Networking Research Group (DTNRG) [43]. One project related to this thesis is the "Disruption Tolerant Shell" project by Martin Lucac et al. [44]. However, their work is application driven as for a certain scenario they are looking at synchronisation between nodes and management, not from a general point of view toward disconnected operations.

The above projects focus on routing of packets within the WSN. This thesis approaches a similar problem with disconnection, but at an abstract software level, with an aim of offering a tool for programmers to program disconnected operations. We have not found any published prior work that deals with disconnection in the fashion presented in this thesis.

3 Motivation

This chapter examines more deeply the issues that arise in disconnected scenarios and programming for applications that must address these issues. First we will identify different ways of connecting and being connected to a WSN, then look at challenges in WSNs scenarios due to disconnected communication.

3.1 Connection to a WSN

There are three important cases when a connection between an end-user and the WSN is not available. Sometimes a connection between the end-user and the WSN and (even) inside the WSN is deliberately disconnected; for example, periodically disconnecting to save power. In other cases it may only be possible to sustain a connection on an irregular basis; for example, when monitoring a heard of moving animals [13]. Finally, there is the case of a broken connection because of unreliable links, failed nodes, or other unpredicted events that causes a temporarily prevented connection. The whole chain of connections from one mote to the end-user is shown in Figure 4.

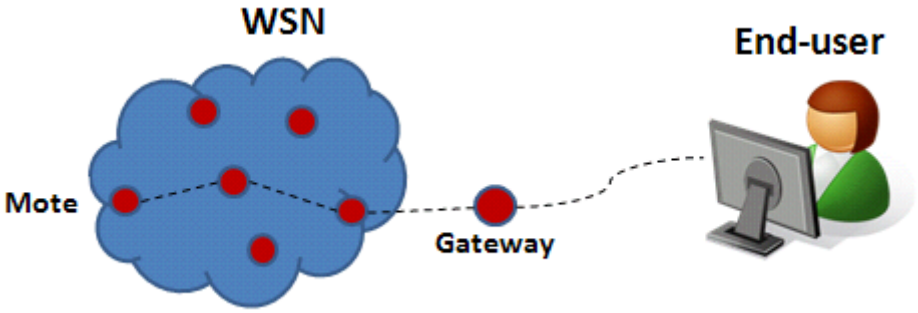

Figure 4: Architecture showing a connection between a mote and an end-user.

Three types of connectivity are described by Kay Römer and Friedemann Mattern: connected, intermittent, and sporadic [2]. If there is always a connection between nodes, then the network is connected; if nodes may be occasionally partitioned, then the connectivity is said to be intermittent; and if nodes are isolated most of the time and only communicate with other nodes occasionally, then the connectivity is sporadic. In this thesis the last two types, intermittent and sporadic, are both considered as disconnected scenarios.

The area addressed by this thesis has many similarities with other topics, such as: “delay-tolerant networking” (DTN), “disruption tolerant networks”, and “opportunistic connectivity”, amongst others. In most of these topics the focus is on the connections **within** the WSN, specifically the network layer dealing with routing of packets. However, little research has focused on investigating how to program solutions for when there is **not** continuous connectivity between the end-user and the rest of the WSN.

Whenever we want to access a WSN for any reason, be it for (re-)programming the nodes, updating software, querying nodes for their current status, changing tasks, or acquiring data, this access can be done in different ways. Generally we communicate via some kind of gateway. This gateway could be one of the motes, a data sink, or a special control-station or base-station. The gateway node might also function as an access-point for the rest of the WSN to access other networks.

The different ways of connecting to a WSN and the different types of WSNs increases the complexity of the system. This thesis project tries to simplify this process. We will assume in

all our scenarios that an end-user wishes to connect to a WSN. In general this end-user will be connected via a wired or wireless connection to a node that could be part of the WSN, but might not always be part of the WSN. See Figure 5.

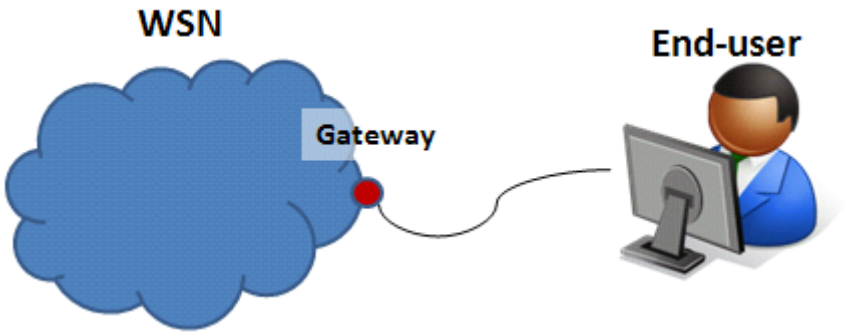

Figure 5: Gateway connecting an end-user with a WSN.

The disconnection that is discussed in this thesis concerns both the connection between the node that is directly connected to the end-user and the connection from this node to the rest of the WSN (as shown in Figure 6). An alternative way of describing this disconnection is that the WSN is partitioned into one part that the end-user is able to communicate with and another part of the WSN that is currently disconnected from the first part.

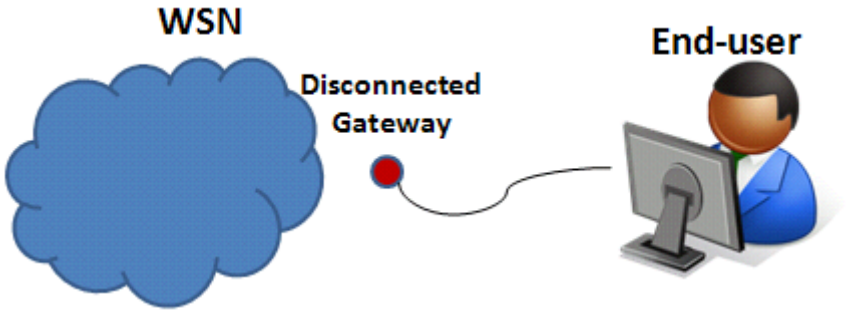

Figure 6: The end-user is disconnected from the WSN.

3.2 Challenges

Challenges concerning WSNs can be divided into two areas: hardware and software. Hardware challenges are to increase performance, create flexible platforms, and reduce the component sizes and cost. Hardware advances such as processor speed and memory capacity amongst others have followed Moore's Law, resulting in the speed or capacity doubling every second year for a given cost. Software challenges are related to programming environments, algorithms, and efficient code. It is not only important to make best use of hardware resources, but also to facilitate programming by the programmers.

To follow trends in embedded systems and offer WSNs solutions in a wide variety of applications it is important that these challenges are met. Also, it is important that suitable program building blocks (often implemented as libraries) be available in the programming environments that the programmers are most likely going to use.

3.2.1 General

An important area of software challenges is producing an easy to understand environment and interface for programmers to work with. Not only experienced programmers will use the network, but researchers who might not have a background in programming computer networks may need to develop applications for a WSN. Therefore, making it easy to create, deploy, and operate a WSN is important, as it does not matter how energy efficient software is if no one uses it. This leads to a more human centric focus for programming WSN applications. This is in contrast to the traditional node centric focus of developing software for embedded systems, which viewed programmers' time as having low cost.

In WSN applications the constraints due to limited resources are always an important to address. While hardware advances give greater processing capabilities, larger memory resources, and smaller components – there is still a need for cheaper, less power consuming, and more flexible platforms.

A major challenge in producing software for WSN is due to the scarce energy resource of the node. Thus the challenge is to produce energy efficient software, as this will increase the lifetime of the network. For nodes that do not have the possibility to recharge their battery (or be refueled), if the energy consumption can be halved, the lifetime of the node is doubled.

3.2.2 Disconnected operation

Programming WSNs is often done in conjunction with a specific application implementation, making the solution suitable only for this specific application and not reusable for disconnected networking. Not only are all deployments of WSNs with disconnected applications unique in their application, but also their solution to disconnected operation is also unique. Therefore, many applications might benefit from a common solution. One challenge is to create such a common solution.

Another issue in disconnected networks that does not relate to routing packets and making good use of storing and collecting data is the user interface for programming disconnected operations. The expanding use of WSNs in different applications today results in scenarios where the programmer needs to operate the WSN over an intermittent or sporadic connection. This is a challenge that is not solved by any of the applications mentioned in section 2.1.2.

Most of the typical challenges of disconnected operations lie *within* the WSN, for instance collecting results and routing packets within the WSN. Examples of problems that can occur during result collection include a mote experiencing a memory overflow while waiting for connectivity to be re-established in order to transmit its measurements. Another challenge can be the uncertainty that a given software update has disseminated fully, i.e., that every mote has been updated.

When the end-user issues a data collection request, this request is spread through the WSN to cause the relevant motes to return their sensed data. When the WSN needs to be reprogrammed there is a similar course of events. In both cases the end-user relies on the routing of the packets to ensure that every node gets the relevant information. When all nodes within the WSN have direct connectivity with the end-user the end-user can ensure that each node receives the information. A more difficult case is when the end-user can only communicate directly with one or a small number of nodes, thus the user must depend upon these nodes to act as relays for the required communication.

3.2.3 Programming Disconnected Operations

We have so far shown that there are a number of applications that are affected by disconnection (see section 2.1.2), and will continue to be so in the future. A novel approach to programming an abstraction to handle such disconnection and the challenges that follow is needed. Today no such abstraction exists.

When programming disconnected operations it is important to know what kind of disconnection you are programming for. When the WSN is in a disconnected state it is usually the WSN that has disconnected itself by turning off its transceiver or the connection was for some reason lost. In the first case, it is the WSN that activates and deactivates the connection, i.e. connectivity is determined by how the WSN was programmed to function. In this case the end-user usually knows for how long the connection will be down, or knows what the schedule of connectivity is planned to be. If this time is known by the end-user, then programming disconnected operations is much easier since the end-user does not have to synchronize when the node will be reachable by the end-user with the time the end-user is accessing the network. Another way around this is tell the nodes about the desired operation(s) and when there is a connection execute these operations. In this case, buffering of operations and possible associated data is needed. Also necessary is some way of marking operations with a timestamp or a comparable identification of the state of the network so no an isolated node starts to execute old operations when it (re-)connects after a long time.

It is obvious (by definition) that sending packets between nodes is impossible when they are disconnected from each other. In the case of packet-switched communication, every problem that arises from this disconnection is related to packets not being sent or received. Unfortunately, unexpected disconnections can occur when updating software in the network, when reprogramming, during data collection/transfer, and when querying nodes for status.

There can also be indirect effects when programming if the WSN is **partially** disconnected. For example, when using data management equalization algorithms (to spread load or storage of data over nodes [40]) and altering information dissemination, as the end-user might not know that some parts of the network are currently disconnected.

4 Classification of disconnected WSNs

A classification model is needed to classify different WSNs and eventually their differences in disconnection. Knowing these differences it should be possible to better adapt the approach to the problem of disconnected operations. Classification of a WSN can be done in different ways and a WSN can be characterized in different ways using different parameters.

4.1 Classification model

In the classification model for disconnected WSNs presented here three parameters are used. These three parameters are: entry point, network topology, and mobility of the network. Details for each of these parameters will be presented below.

Entry point characterization concerns the means of connecting to a WSN. The entry point refers to the connection from the end-user to the gateway to the WSN. One way to connect is through a **fixed** gateway, i.e., always using the same (intermediate) node to access the WSN – as illustrated in Figure 7. A second way is to connect to **any random node** or a subset of nodes that are connected to the rest of the WSN – as illustrated in Figure 8.

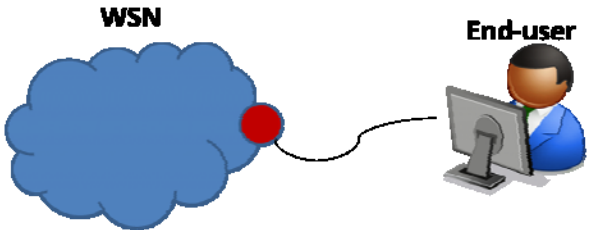

Figure 7: Fixed entry point

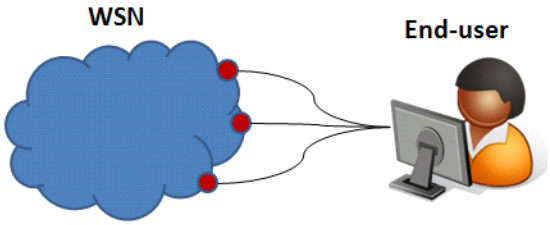

Figure 8: Any entry point

The network topology can be divided into three categories: fully connected nodes, disconnected islands of nodes, and isolated nodes. These three types of network topologies are shown in Figure 9. Due to characteristics of WSNs categorising a WSN based upon network topology is not a straight forward task. If the network should be able to adapt to changes in topology, then a failing node might change the topology from fully connected to disconnected islands **or** a disconnected island might change to isolated nodes. Note that a fully connected network could transition in stages (i.e., via link failures to become disconnected islands) to isolated nodes, or this transition from fully connected to only isolated nodes could even happen in a single step. In this thesis we are assuming that the system finds itself most of the time and that a change in network topology is unlikely and temporary.

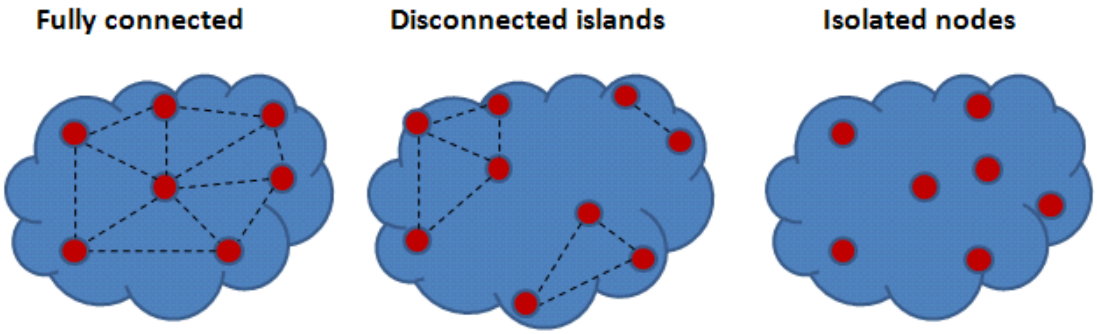

Figure 9: Three types of network topologies.

Mobility of the network is a third parameter for classifying a WSN. The network (as a whole) or parts of the network could be mobile. Mobility of the nodes may lead to changes in the network topology (as described above). Recent work by David Culler et al. [46] discusses that "mobility changes everything" in WSN. A common scenario for mobility in WSNs is when a node (mote) is attached to an animal, as shown in Figure 10. Note that if multiple nodes were carried by the animal, then the network formed by these nodes would be mobile. One of the important questions is if the nodes are moving coherently together (i.e., all moving on the same path), in this case there is a different expectation for the stability of the network topology, than in the case when paths of the nodes are not coherent (i.e., they are moving along different paths). In this second case, link failures will be more likely.

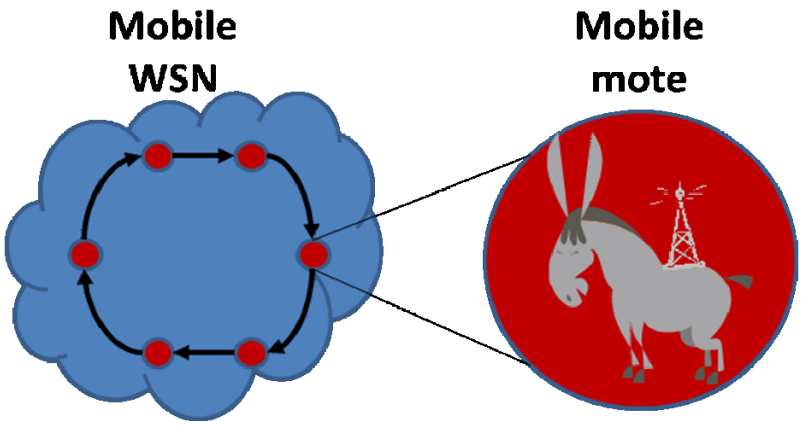

Figure 10: Mobile WSN containing mobile motes, e.g. motes attached to animals.

These three parameters enable us to classify a WSN in a three dimensional space. With two classes of entry point (fixed or mobile), three network topologies (fully connected, Disconnected islands, and isolated nodes), and with or without network mobility – results in 12 different classes (as illustrated in Figure 11).

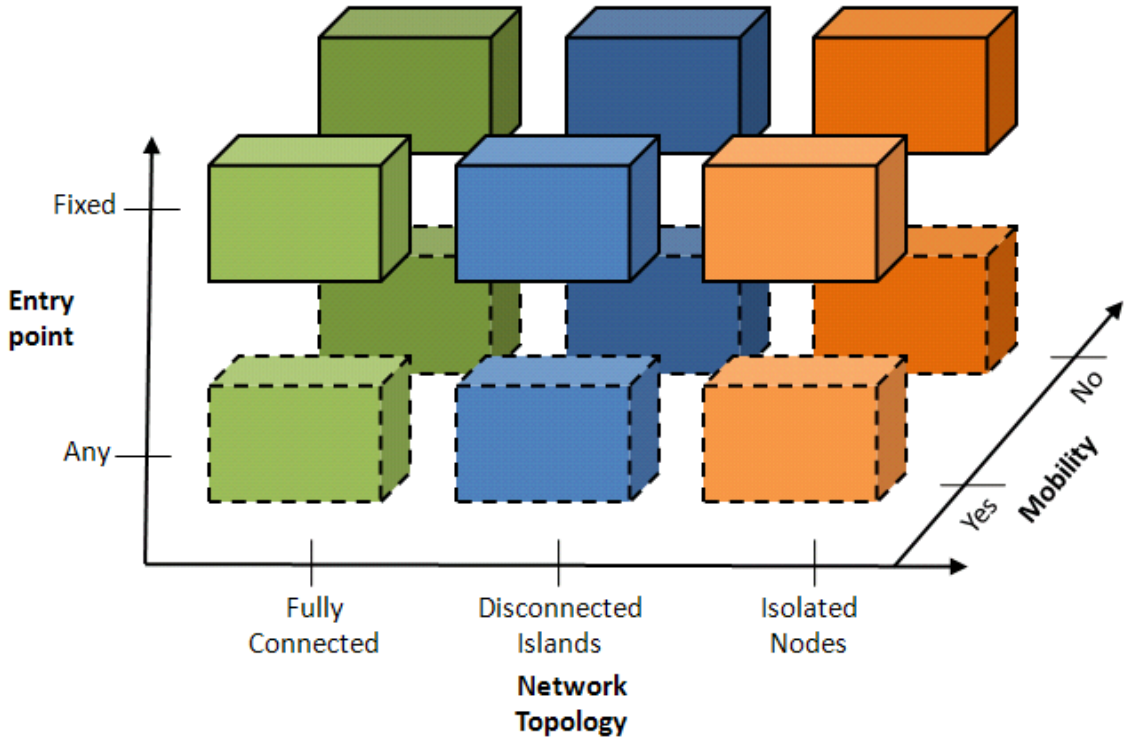

Figure 11: WSN classification graph with three dimensions: entry point, network topology, and mobility.

Some of the combinations in the classification graph are not relevant to this thesis project. For example a WSN with isolated nodes, where the nodes do not communicate with each other for any reason, that the end-user must access through each individual node (i.e., each node is the only entry-point to access this node) gives no programming challenge and if a wire is used to connect to the node is not a wireless network by definition; hence it is not of further interest in this thesis. While a collection of isolated nodes with a fixed entry-point that must be used to communicate with any of the nodes (i.e., this entry point acts as a gateway) is clearly a WSN – since the nodes are communication with the gateway wirelessly. These two scenarios are shown in Figure 12 and Figure 13.

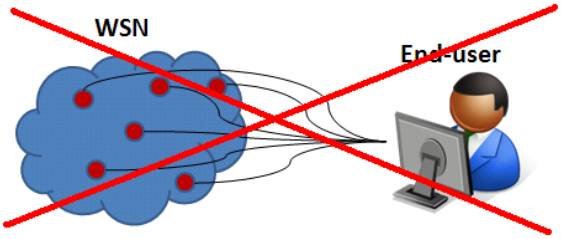

Figure 12. Isolated nodes accessed one by one.

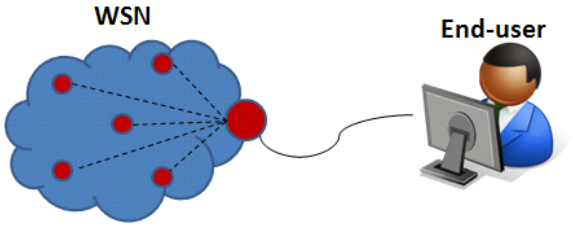

Figure 13. Isolated nodes accessed through a fixed gateway.

The examples above suggest that there is an intermediate situation that is of interest: when the user can communicate with isolated nodes or disconnected islands via a number of different gateways. There is also the related situation when a fully connected network has multiple gateways – this situation is interesting as it increases the robustness of the communication between the end-user and the nodes; as a link failure that results in a partitioning of the network into disconnected islands might still provide connectivity with the end user if there is *at least one* gateway to each node or island.

Table 1. Examples of applications classified according to the proposed model.

Mobility	Entry Point	Network topology	Example
Yes	Fixed	Fully Connected	
Yes	Fixed	Isolated Islands	ZebraNet [13]
Yes	Fixed	Isolated Nodes	Argo [6]
Yes	Any	Fully Connected	
Yes	Any	Isolated Islands	
Yes	Any	Isolated Nodes	
No	Fixed	Fully Connected	
No	Fixed	Isolated Islands	
No	Fixed	Isolated Nodes	Glacier [4]
No	Any	Fully Connected	Rolls Royce WIDAGATE [47]
No	Any	Isolated Islands	
No	Any	Isolated Nodes	Buoys [11]

4.2 Other classification parameters

In other classification models, for classifying WSN applications, many other parameters have been used. Kay Römer et al. [2] looked at a design space using the following parameters to differentiate applications: deployment (in regard to method and time), mobility of nodes, resources, implementation cost, energy source, heterogeneity, modality, topology, coverage, connectivity, size, and lifetime. Table 2 shows an example of this method of classification for two disconnected WSNs: ZebraNet and a WSN used for herding.

Table 2: Example of a classification table. [2]

	Deployment	Mobility	Resources	Cost	Energy	Heterogeneity	Modality
ZebraNet	Manual, one-time	all, continuous, passive	matchbox	-	battery	nodes, gateway	radio
Herding	Random, iterative	all, continuous, passive	brick	-	battery	homogeneous	radio

	Infrastructure	Topology	Coverage	Connectivity	Size	Lifetime
ZebraNet	Base station, GPS	Graph	Dense (every animal)	sporadic	Tens-thousands	One year
Herding	Satellite	Star	Sparse	Intermittent	1300 deployed, 3000 planned	4-5 years

Another example of classification of WSN applications is given in [36], where a two-dimensional classification model is described. The model has two parameters, space and time, which are each divided into two segments: global and local for space, and periodic and event-driven for time.

As described above, this thesis utilizes its own classification method. This method will be used later to (1) evaluate the implementation of the solution and (2) to make programming disconnected operations easier.

5 Design and implementation

This chapter begins with a presentation of the general problem and its general design solution. Following this is an examination of specific problems derived using the classification model presented in the previous chapter. From these specific problems a set of building blocks will be presented. The following chapter evaluates these building blocks, also called modules.

5.1 General problem

As stated in section 1.1 on page 2, a WSN application programmer who is developing disconnected WSN applications must currently implement an application specific solution, as a general solution or solutions do not currently exist. A general solution that addressed the problem of disconnected operations could be applied to many different applications. Simplifying the development of disconnected WSN applications could foster the development of new WSN applications by an expanded set of developers.

5.2 Design of a general solution

To simplify programming of disconnected operations we introduce a number of modules as a part of the solution (illustrated in Figure 14). These modules can be used in conjunction with other modules or by themselves and allows for different implementations with different functionality. A module can communicate with other modules on the same node or through the network with modules on other nodes. The end-user might both give input and receive output from modules. Due to the use of a common interface, these different implementations can be interchanged with each other to meet the needs of a specific application. The re-use of hardware and software modules has been successful in many other areas, by reducing development time and effort, increasing quality - as an improvement in a module can be applied to all instances of its use, enabling “best of breed” selection of modules, etc. In the following sections we will present this common interface and some of the modules that we believe are necessary to address the problems faced when developing applications for disconnected WSNs.

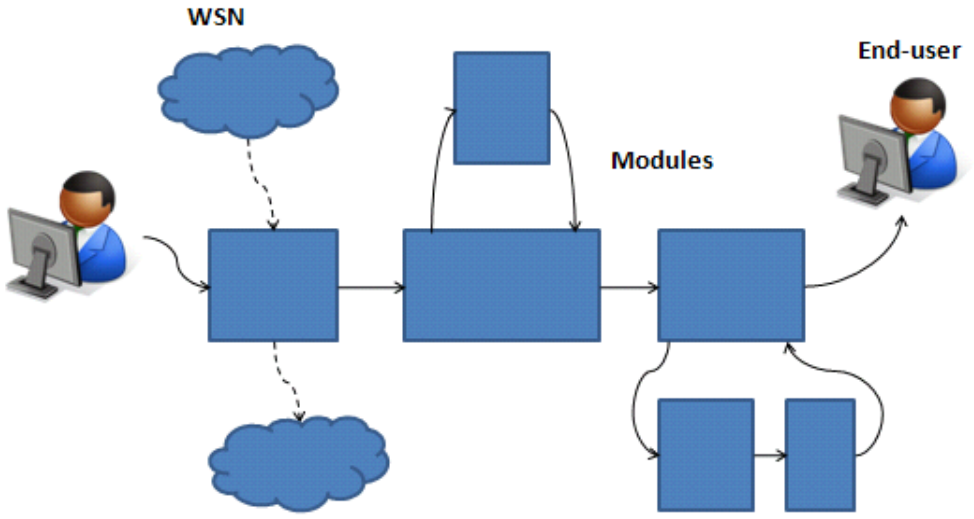

Figure 14: Utilization of modules in different ways

One of the benefits of this modular design is that it permits the application programmer to easily replace one module with another. For example, a module used to communicate with other nodes could be replaced by another module that has a different power profile. This can have significant benefits, if only local optimizations are necessary to meet the application's overall power requirements.

5.3 Proposed Modules

After examining a number of WSN applications a need for a general solution to enable these applications to be used in conjunction with disconnected operation is considered. We in this chapter propose a small set of modules to facilitate the development of future disconnected WSN applications. The modules are: UserWait, Buffer/Store, Replicate, Collect, and Schedule. These modules will each be described in detail in the following subsections.

5.3.1 Userwait

A common feature of many WSN application is the need for a process to execute up to some point (specified by the programmer), where it should wait for the user to continue the process. The initial execution of this process could be activated by an end-user connecting to the WSN or initiated automatically based on some event (time of day, sensor threshold, completion of another task, etc.). Userwait divides a series of processing into portions that can be executed without the presence of the user and wait barriers that require communication with the end user before the processing can continue.

A typical example of when such a module would be used is in a disconnected WSN where the end-user is only connected to the network for a limited time. The nodes will run the userwait module individually when measuring data and storing it locally. When the end-user connects and announces its presence the nodes will react to this and start sending their stored data to the end-user.

5.3.1.1 Design

The design of this module could be divided into two sub-modules: one sub-module that manages the execution of the process and the other sub-module communicates with the end-user and acts on the first sub-module by allowing it to continue the execution of the process. The first sub-module could be further divided into three parts: one part that handles execution of the process before the break, one part that handles the break and waiting for the end-user to connect, and a third part that handles the execution of the following process. This is illustrated in Figure 15.

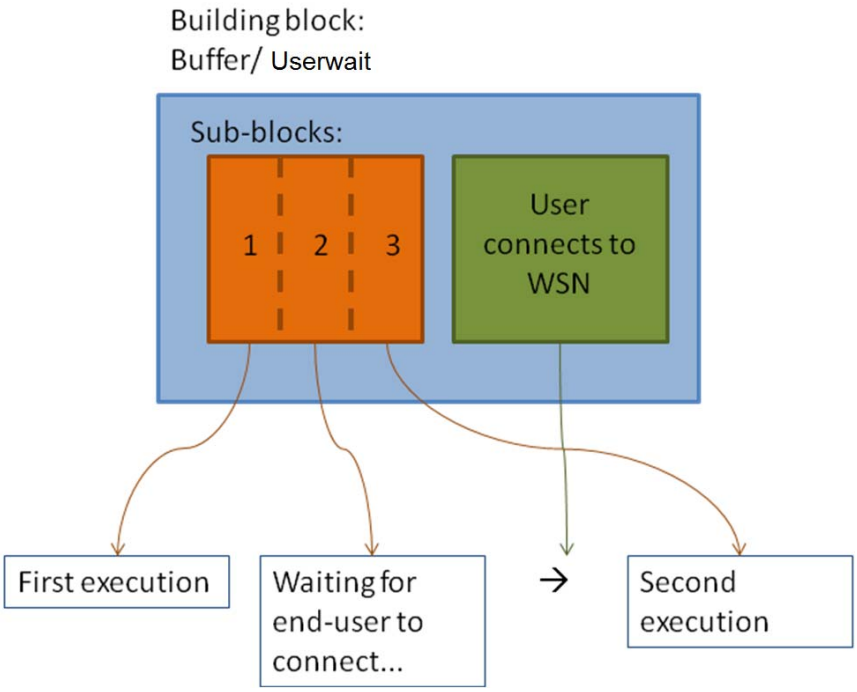

Figure 15: Userwait module with sub-modules

5.3.1.2 Requirements

The programmer specifies what is to be executed before waiting for input from the end-user and what is to be executed afterwards. Communication between the two sub-modules (1) announces that the first sub-module has completed its execution and that the second sub-module is waiting before proceeding (2) communication from the end-user to the module, and (3) communication from this end-user signalling module to the sub-module that is waiting indicating that the end-user wants the processing to continue.

5.3.1.3 Discussion

This design could be used to allow a measurement process to collected data in a WSN that will be buffered (by the first sub-module) until the end-user connects and activates the third sub-module (e.g. sending measured data to the end-user).

Announcing the presence of the end-user could be done by broadcasting the end-user's presence over the whole WSN independent of the entry point that is being used by this user to connect to the (previously disconnected) WSN.

More generally we can think of the activity as: Execute process; signal that you are at the barrier; wait at the barrier; execute the process. As there can be multiple such barriers in the WSN, especially if the WSN is being used to perform several different tasks. This implies that we need labels for the barriers – so that the end-user can know which barrier the execution is waiting at, in order to be able to say which of these is to continue.

5.3.2 Buffer/Store

In many WSN applications there is a need to temporarily store data. In both connected and disconnected WSNs this is needed.

An example of when a buffer/store module would be used is when data traversing through the network, or between different modules, needs to wait for a certain event to happen before it can continue. In a disconnected scenario it is likely that the buffer/store module will be used by another module made for disconnected scenarios (e.g userwait module) and therefore might have dual functionalities for better utilization, e.g. it might both write and read data depending on the situation.

5.3.2.1 Design

The Buffer/Store module stores data locally at a node when given data. The stored data will now be available from this buffer, for use by other modules. This module is simple in its design and different implementations of this module can be constructed depending on the policy used for storage. In a disconnected scenario this module would store data when given input and should be able to output all its previous data upon request. Depending on the policy the data that has been output could be removed from the buffer or not.

5.3.2.2 Requirements

To identify this buffered data some sort of identifier is needed. The data can be retrieved later using this identifier. The module must be able to receive input in some way and return the buffered data as output. This data could be stored in a local file system.

5.3.2.3 Discussion

The Buffer/Store module functionality is needed both for buffering scenarios and for persistent storage in the nodes of the WSN. Many different implementations of this module are possible and each of them might be useful in specific situations depending on the application.

These different implementations have different properties, hence raising a number of questions, such as whether the data can be asked for only once or more than once; whether asking for the data using this identifier is idempotent, i.e., that it returns the same data, or if it only returns the data currently stored using this identifier; when should data be deleted; when can identifiers be re-used; what is the valid scope of an identifier (for example is it a locally valid ID or globally valid ID – in the later case can it be guaranteed to be unique); … . These questions need to be taken into consideration when implementing a new module.

5.3.3 Replicate

In many applications copies of data values need to be made available to more than one node, i.e., the data needs to be replicated. As this is a general problem, it should have a general solution; but the solution may need to be parameterized. Forwarding data from one node to another can be achieved by the degenerate case of replication to a single node.

An example where the replicate functionality is helpful is in mobile disconnected WSNs where data gathering relies on nodes being in the vicinity of a sink that collects data. The end-user would not always be able to connect to moving nodes, but the sink would be static and might always reachable. A node that will never be near the sink could replicate its data to other nodes that might upload their own data along with the replicated data from other nodes to the sink when the node is near a sink.

5.3.3.1 Design

Data or copies of data can be broadcast or unicast from a node to other nodes in the network[1]. Data produced locally on the node is replicated to other nodes so that several nodes share the same data. Replicated data received from other nodes, just like locally produced data, is stored locally.

5.3.3.2 Requirements

The replicate module must be able to take input and provide output, as well as communicate with other nodes. What policy should be used for data replication should be decided by the programmer when using this module. The policy(s) selected for communication will also be coupled to the means of communication – which can also be encapsulated by this module.

5.3.3.3 Discussion

Replicating data can be done in different ways in order to achieve different goals, therefore it is important that this module can be tailored as needed. For example, an instance of this module might replicate data only to the closest neighbours, while another instance might replicate the data to all nodes in the network. Replication of data (and processing) can be used for load balancing, to provide redundancy so data is not lost if a node goes down, and to locally compute functions over data in a region.

5.3.4 Collect

The Collect module is designed to run on a node acting as a sink, i.e., it will collect data from other nodes. Note that in most cases this module will inform other nodes of its existence, enabling the nodes to route data to this sink.

Collecting data could both be done by the end-user as well as single nodes, possibly acting as gateways. In a time synchronized network nodes could at a given time instance all start transmitting data to some destination that could be the end-user or a sink that would collect the data.

5.3.4.1 Design

The Collect module can have two major designs. One design opens a connection to the network and starts listening if any node is sending measured data, while another design would pull data from a source and collects it. This module's design should make it easy for the application programmer to collect data from many nodes, then process and store the data for subsequent transmission to the end-user. Having this collection occur within the WSN (particularly at the gateway) is quite suitable for disconnected WSNs, as it allows the data to be staged for later collection.

Some of the design questions that need to be addressed are: can there be multiple sinks, can a sink indicate what type of data it is interested in, can a sink indicate that it is no longer interested in data, can a sink indicate that it is interested in data at some specific point (in the future) in time, can a sink indicate a maximum rate for data from any single source, etc.

[1] Any sensors in the WSN will be attached to a node, hence using replication the sensor data can be sent to multiple other nodes from the node connected to the sensor.

5.3.4.2 Requirements

Important when collecting data is the type of communication to be used. In some settings a reliable way of transferring large amounts of data over several hops is desirable. There also needs to be a way to identify the data to be collected.

5.3.4.3 Discussion

Collecting data is crucial in WSNs, as one of the major uses of WSNs today is to collect data from sensors that are attached to the nodes of the WSN. The method chosen for communication should be that method that best suits the *specific* application. For a disconnected WSN it is advantageous to be able to retrieve data when it is convenient for the end-user, therefore a pulling collect is needed and not only a collect that announces its presence and then only waits for nodes to send data.

5.3.5 Schedule

The Schedule module is the most complex module to implement due to its many parameters and because it ties many of the other modules together. While it does not directly address disconnected operations, it helps the programmer to schedule the execution of the other modules. This module might also be used to allow the end-user to schedule when and how to run commands (that have been implemented by the programmer who developed the application).

The schedule module would be very helpful in WSNs where different operations need to be executed at different time instances. A typical example is monitoring animals or environments with different cycles, e.g. nocturnal animals that are active during night and sleeps during the day. In this case it might be wise to schedule that the radio should shut off during the day and turn on again during the night when the animal is active to start measuring data.

5.3.5.1 Design

The schedule module should be divided into one part that handles end-user requests for managing tasks and one module that executes the tasks at the specified (scheduled) time. The first part is generally implemented via a command shell that takes user requests to execute specific commands/tasks and schedules them. Some of these commands/tasks may modify the current schedule, for example increasing or decreasing the priority of a task, removing a task from the schedule, adding a task to the schedule at a particular point in time, adding a task to follow the completion or another task, etc. The second part is generally referred to as the *scheduler*.

5.3.5.2 Requirements

Several types of user inputs are needed for this modules functionality. It is needed to identify different tasks, schedule them according to starting time and runt time. The end-user should be able to remove existing tasks and also add new tasks according to parameters such as the time interval, frequency, and priority.

5.3.5.3 Discussion

The schedule module allows for better utilization of all the other modules. Many different ways of implementing this module is possible. There are several parameters to take in consideration and many different algorithms can be used to compute a schedule according to these parameters. One of the most important design questions is if the scheduler should be a pre-emptive scheduler or not.

5.4 Implementation

Five different modules have been presented to aid programming disconnected WSNs. These modules will be implemented as shell commands in the Contiki OS, enabling them to be used in a shell environment.

5.4.1 Contiki Shell

Many operating systems for WSN offer a shell interface to the user, see for example the previously mentioned OSs: Contiki[21], LiteOS[23], and TinyOS[30].

The Contiki shell is similar to most other shells, using the ability to send data between commands through “pipes”, and allow commands to run in the foreground or background. Hence by implementing the modules as shell commands the functionality of the commands is transportable to other OSs that uses a shell. However, the actual code itself is not portable since it is optimised to work with the Contiki OS. The code would have to be modified and adapted to work on another OS.

The UNIX shell [48] utilizes standard input and output of bytes between programs via a pipe. In this thesis we also propose to use a shell based solution with pipes. Thus the common user interface to all programs will be the shell input and output. The Contiki OS does not use standard input/output filehandlers as in an UNIX based OS, instead the shell utilizes an *ad hoc* implementation to handle data via pipes.

Shell commands are easy for programmers to use. This is particularly important when the user is not an experienced programmer - but instead is often a scientist with little programming experience. The interface that a shell often offers is easily understood.

The Contiki shell differs from other shells, such as the UNIX based OS shells, in that two instances of the same command cannot be run at the same time. This may or may not lead to unexpected behaviour for end-users (especially if there are multiple simultaneous end-users of a single WSN). In conjunction with the Userwait module this may lead to blocking the use of commands for the end-user for indefinite time, or even lead to deadlocks.

5.4.2 Alternatives

Another ways of implementing the modules would be by implementing a new programming language. This approach would give the programmer a more complex interface, but allow the programmer more detailed control. As the scope of the language would be quite small it could be built as a “little language” [49] on top of an existing language, since the extensions are only intended to support programming of disconnected operations.

5.4.3 Implementation details

The implementation of the proposed set of modules will be described in this section. The approach that has been taken is a scripting based approach in which the end-user enters commands that are to be run in a shell via a command line interface. These commands have been implemented specifically for the Contiki shell, but the general idea is not specific to Contiki. Therefore other OSs with a shell (such as TinyOS and LiteOS) can benefit from similar commands.

Code examples will be given in a separate text-box with explanatory text following the example code. All commands in the body of the thesis will be set in an *italic* font face to highlight the command.

5.4.3.1 UserWait

The *userwait* command takes two separate input arguments. The first argument is a command, or set of commands, that when executed is expected to be able to take input data, and when executed without input data it should output its previously stored data. The *store* command is built this way. The second input argument for *userwait* can be any subset of Contiki commands that will take the first command's output as its data input.

The *userwait* command shares information about input arguments with the *userconnect* command. The second string of commands, which was given as input to userwait, will only be executed when the end-user runs the *userconnect* command. When running the *userconnect* command, output from the first command is passed as input to the second command string. The *userconnect* command can be executed several times until the *userwait* command stops receiving data, then the *userconnect* command can only be run once more.

By programming several nodes, or just one node, with the *userwait* command it is possible, together with the existing Contiki Shell commands *netcmd* or *nodecmd*, to activate the second command string on all nodes in the network, or just one node, when connected to another node.

5.4.3.2 Store

The *store* command is a very basic command and is easily implemented using Contiki's existing file system. This command is very useful in a disconnected application scenario, specifically with the "*userwait*" command. For example, one might want to store data locally before any further processing of the data occurs.

Store has two modes; as long as it is given input through its pipe it simply *stores* the received data, as soon as it receives a null-byte it outputs all of the data to the following command in the pipe. Note that the end of input is signalled as a null-byte.

The reason for implementing *store* as a separate command, rather than building it in to the *userwait* command, is because a user might want to utilize a number of different policies for storing data; depending upon characteristics of the data and of its inter-arrival times and duration of storage. Two examples of different implementations of *store* are given below. One is queue-based, perhaps the most typical scenario, where the first data that is written to a queue is the first to be taken from the queue, a so-called "first in, first out" (FIFO) queue discipline – see Figure 16. Another implementation provides stack based storage, such that the last data that is written to a stack is the first to be read, a so-called "last in, first out" (LIFO) queue discipline. Both these disciplines have been implemented in our solution. The

FIFO store-command is named only *store* in our solution, and the second LIFO discipline is called *store2*.

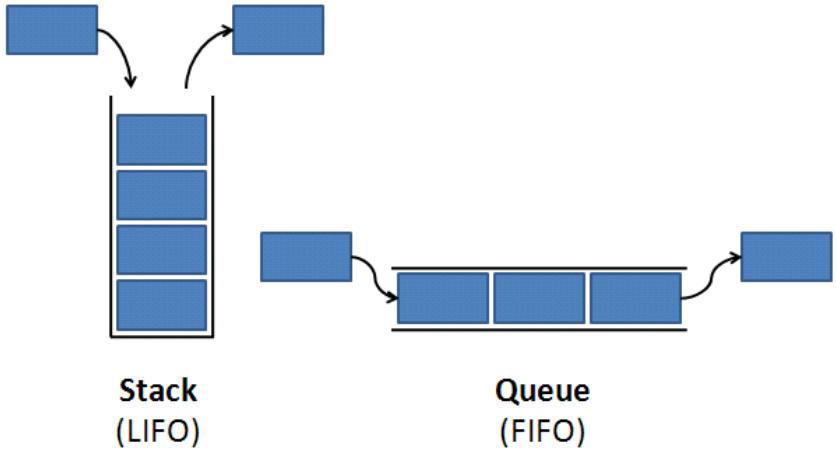

Figure 16: Stack and queue policy

```
Shell>echo test | userwait {store buffer_file}{write output_file} &
Shell>userconnect
```

In the code example above all of the commands in the first line will be run in the background, as specified by the "&" parameter .This allows the system to execute multiple commands and for the user to enter additional commands while *userwait* is waiting. *Echo* will output "test" as input data to userwait, *userwait* will run the command *store* with the string "test" as input. *Store* will write its input data (in this case the string "test") to the file "buffer_file" with FIFO policy. *Store2* command could be used for LIFO storing policy. When no more data is received userwait will start waiting on a continuation event. This event is caused by the user running *userconnect*. The *userwait* will run the *store* command without input (a 0 byte length input) which tells *store* to output all the previously stored data. The output data is sent as input to the following *write* command that will write the input to a file named "output_file". Note that *echo* and *write* are existing Contiki shell commands.

```
Shell>sendcmd node-id {userconnect}
Shell>netcmd {userconnect}
```

Userconnect can be run when the end-user is connected to the network. It will activate *userwait* commands running in any part of the network. All of Contiki's built-in commands can be used in conjunction with the new commands that this thesis project introduces. Note that the command *sendcmd* tells one specific node to run a command; alternatively *netcmd* can be used to tell all nodes in the network to run a command. In the first line of the example code the *netcmd* is used to tell all of the nodes to execute a *userconnect*, while the second line using the *sendcmd* is used to tell a specific node (identified by node-id[2]) to run the *userconnect* command.

5.4.3.3 Replicate

The *replicate* command takes the input data and broadcasts it to other nodes, then outputs this data as its output (i.e., as input to the following command), see Figure 17.

[2] For details of how the nodes are identified and how an end-user or program can know these identities see Contiki Reference Manual [43x].

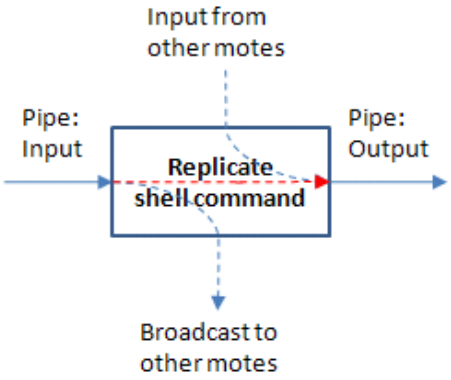

Figure 17. Replicate command with "send through" functionality (highlighted by red arrow)

Note in the example above that the replicated data from other motes is also output. An alternative version of the *replicate* command might only replicate input data and only output data received from other nodes, as shown in Figure 18. In our implementation it is possible to choose between these two policies by giving an input parameter to the replicate command. The input is "1" for the above policy and "2" for the policy below.

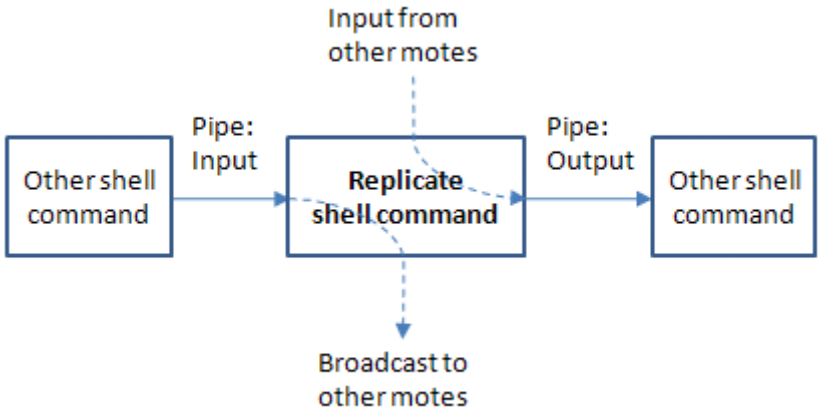

Figure 18: Replicate command data passing

The *replicate* command will generally be located between pipes of other commands.

```
Shell>echo test | replicate 1 | write file
```

In the above code example, the command *echo* sends the string "test" as its output through a pipe to the *replicate* command. The replicate command will subsequently receive this data, and then broadcast it to other nodes. Having given the input "1" to the replicate command , it will output data that comes in through the pipe as well as replicated data from other nodes. Note that these other nodes must be waiting for such a broadcast by being in a replicate command themselves. After broadcasting the input data to all of the other nodes, the replicate command outputs the string "test" through the pipe. In this example, this output becomes the input to a *write* command that writes the string to a file named "file".

5.4.3.4 Collect

As mentioned in section 5.3.4, two basic designs were possible for the collect module. Since a collect shell command already exists in Contiki we will only implement the "pulling collect". The existing command starts by announcing itself as a sink node, then starts to listen for certain packets sent by the motes. The sink waits until it receives data.

Since there is no way for the sink node to collect only certain data from the motes a pulling collect command was created in this module. The functionality of the *pullcollect* command could be achieved by using the *collect* command and other existing commands available in the Contiki shell. However, we have chosen to implement the *pullcollect* command as a building block of its own. This makes it independent of other specific commands and simplifies usage for the programmer, amongst other advantages (mentioned in section 5.2).

The *Pullcollect* command acts as a "pulling-command" that pulls data from other nodes. This command takes a filename as input. The filename will specify the name of the file whose contents are to be transmitted by each node in response. *Pullcollect* sends a message to all nodes in the network telling them to open and read the contents of the file specified by a filename, then each node will send the contents of this file back to the node issuing the *pullcollect* command. If the file does not exist or the file exists but does not contain any content, then an empty packet of zero length will be sent back. *Pullcollect* will output the data interleaved messages as it receives the data, independent of what node it received it from. It is therefore important that data has some kind of stamp or difference so it is later possible to identify data from a certain node, e.g. a node could prepend its node ID at the beginning of the data.

```
Shell>pullcollect file | write collected_data
```

In the above example, the *pullcollect* command issues a pulling collect at all nodes connected to the network. All nodes attempt to open the file named "file", if this is successful they will each attempt to read the contents of this file. Each node will reply by either sending the contents of the file back to the node issuing the *pullcollect* command or sending an empty packet. Received data is outputted to *write* command through the pipe and written to the file named "collected_data".

5.4.3.5 Scheduling

The schedule module is divided into two parts; one shell command part and a server that runs in the background. The command *discsched* adds, removes, or lists scheduled tasks, depending on its arguments. The server waits until the scheduled time to start the first task to be run. Note that the scheduler utilizes its local sense of time to determine when these tasks are scheduled.

When a task is added it is inserted into a list that is sorted by the time the task is scheduled to run. Tasks in this implementation do not have user specified priorities. Any task that is scheduled to run while another task is already running will interrupt and terminate the currently running task, before starting to run the new scheduled task. At the same time as the task is terminated it is removed from the schedule list. A task is identified by a task identification number that is specified by the program. When adding a task it is possible to make it cyclic, by giving input arguments that specifies the period of the cycle and the number of cycles. There is only one instance of a cyclic task in the sorted list, but whenever the task has finished or been interrupted it is added again to the list with a new start time and a decremented number of cycles left to run.

```
Shell>discsched
```

If the *discsched* command is run without any input argument, as in the example above, then if there is one or more tasks in the waiting list the command will print information about each task. This information includes the ID (number) of the task, the starting time when it is scheduled to run, how long it should run, the commands to be run, and also information about

cycling if the task is cyclic. The local time (i.e., a timer used by the scheduler) is incremented every second, starting at the value 0 when the discsched server started. This command will also output the local time that the scheduler uses for scheduling the tasks.

```
Shell>discsched 10
```

To remove a task from the list of scheduled tasks, the *discsched* command is run with a single input argument, the identification number of the task to be removed. If the task exists, then it will be removed from the list. If the task does not exist, the command will do nothing. When removing a cyclic task the instance of the task in the list will be removed, hence no future cycles of this task will be executed. The example above removes the task with ID 10.

```
Shell>discsched 7200 600 {echo test | write file}
```

The above code example adds a new task to the schedule. The new task will be scheduled to be run after the server has been active for two hours (7200 seconds). This task will run for a maximum of ten minutes (600 seconds). The commands to be run at the specified time are given as the last input argument to the *discsched* command. In this example the *echo* command will simply output the string “test” through the pipe as input to the *write* command that will write this string to the file named “file”. The command will only execute on the one node at where it was initialized.

When specifying how long a task is to be run, it is important to know that a task can finish before this time, although it is still kept in the task scheduler. Instantaneous tasks will run in under a second, unless for some reason the task execution hangs – in this case, the task will be terminated by the scheduler when the time for the task has elapsed.

```
Shell>discsched 1 5 100 25 {blink 5}
```

The above example adds a new cyclic task to the schedule. This task will start immediately at time value “1” (given by the first input argument), run for 5 seconds (given by the second input argument), the command “blink 5” will be executed (causing a LED on the mote to blink 5 times), the number of iterations is defined by the third input argument and the period of each iteration is specified by the fourth input argument. In this example the task will be repeated 100 times with a period of 25 seconds. The run time is included in the period, i.e. in the above example the task will be run every 25th second (as shown in Figure 19) giving a 20 second period in between for other tasks to execute.

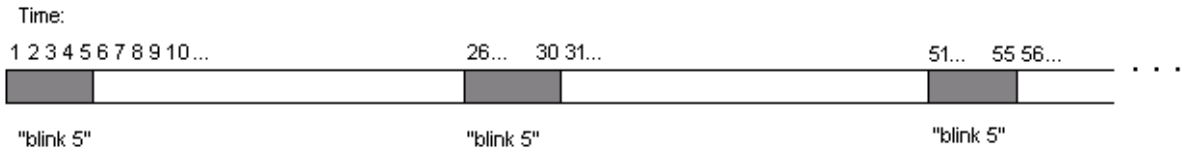

Figure 19: Execution of a cyclic task

6 Evaluation

In this chapter an evaluation of the shell-based solution and commands will be presented. This evaluation will look into different aspects of the solution, compare it with similar solutions and evaluate its performance by implementing it in a live application. Finally, the solution will be evaluated from a qualitative point of view as well as a quantitative one.

The qualitative aspect when evaluating this solution concerns how well and correctly this solution (using shell commands) works. The quantitative aspect concerns the performance of the implemented solution with regard to energy consumption, code size, and coupling between modules.

6.1 Evaluation scenarios

Due to the differences between disconnected WSNs and how they might be programmed two scenarios will be discussed. The first scenario will be a WSN with mobile nodes and the second scenario will be with static nodes.

6.1.1 Scenario with mobile nodes

This scenario represents a typical application of an animal monitoring WSN. For example, monitoring badger habitats [50]. Here badger families move around in different areas and placed between these areas is the base station. The base station is placed where many of the badgers will visit. The base-station will be at a fixed location and the end-user will opportunistically connect to the base-station to gather data. When motes attached to the badgers approach the base-station the mote's data will be transferred. Since badgers are nocturnal animals the radio will be shut off during most of the day when the badgers are sleeping and less active.

6.1.2 Scenario with static nodes

This scenario represents a typical application of an environmental monitoring WSN. For example, monitoring sounds in a forest [51] where static nodes are used. Here the nodes are attached to trees throughout a forest. Collecting data in such a network could be done opportunistically by announcing the presence of a sink node so that a routing table can be set up at each node, so that subsequently all of the nodes can send their collected data to this sink node.

6.1.3 Shell implementation

Examples of shell commands to address these two specific cases using our solution are presented in this subsection. The scenarios differ in the way that the mobile nodes may move around, thus only when the nodes are near the sink it is possible to upload data to it, if they ever are near. In the static scenario nodes will have fixed locations where all nodes are connected which makes it possible for the sink to communicate with every node at any time instance. The difference in the two scenarios requires different implementations.

6.1.3.1 Mobile

To represent the mobile scenario with monitoring badgers, or similar cases, a predefined movement pattern will be used. This movement pattern represents the case where nodes are separated from the sink, then one node at a time approaches the sink and uploads data to it. In this specific case we used a total of four nodes: three mobile nodes and one sink node. The three nodes will be out of communication range of the sink, but within communication range of each other. Between the three nodes data will be replicated.

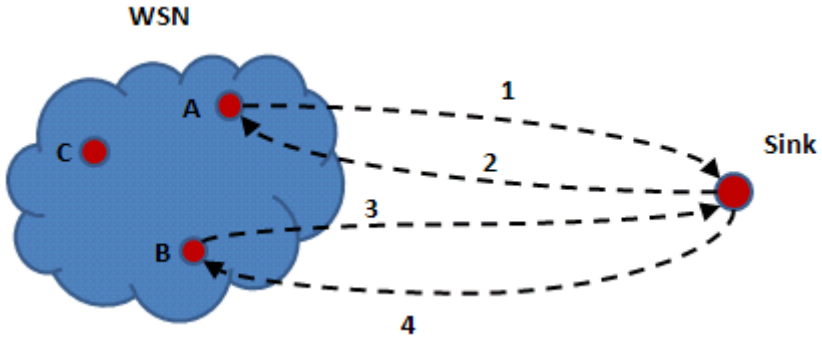

Figure 20: Two nodes, A and B, move along the dotted lines in the numbered order to connect to the sink and upload data.

The order of how nodes move is shown in Figure 20. Two nodes, A and B, will one after the other approach the sink. All nodes in the network will replicate to nearby neighbors. In this scenario node C will never approach the sink, but since its data is replicated to nodes A and B the sink will receive node C's data when nodes A or B approach the sink.

```
Node > discsched 0 140 { dummy_data | replicate 1 | userwait {store fileX} {send}}
Node > discsched 141 20 {radioOff}
```

The above code will be programmed into the three mobile nodes. Each node gets two tasks added to its scheduler (referred to as task numbers 1 and 2). Immediately at start up the mobile nodes' task 1 will execute the *dummy data* command that generates test specific data once every second. In a real deployment a command that reads data from one or more sensors would replace the *dummy_data* command. The dummy data is sent through a pipe to the *replicate* command, which replicates data to nearby nodes. The input argument "1" tells the replicate command to both replicate to other nodes and locally store the generated dummy data. Finally, the data is received through the pipe by the *userwait* command that in turn forwards its input to the *store* command that saves the data locally into a file *fileX*. When a node runs the *userconnect* command, then the second part of the *userwait* command is activated and stored data is sent to the sink node by the *send* command.

Task 2 will execute at time instance 141, after task 1 has finished at time instance 140. Task 2 will turn off the radio in order to save energy. This would represent the scenario of monitoring cyclic environments where there are periods where the node does not need to communicate with other nodes.

```
Sink > collect | write collect_data &
Sink > discsched 30 5 6 30 { netcmd {userconnect} }
```

At the sink-node the above code will be programmed. The two code-lines will run one set of commands (those shown on the first line) in the background, while the second line will add a new task, here called task 1, to the scheduler. The first line of code will start the *collect*

command. All data received by this command will be forwarded through a pipe to the *write* command that will save collected data to a file *collect_data*.

After 30 seconds, Task 1 will run the *netcmd* command. This command will be rescheduled by the scheduler to run every 30 seconds and be repeated 6 times. *netcmd* is an existing command in Contiki that takes a text string as an input argument and tries to run this on all the nodes in the network as a command. In this example the *netcmd* will cause all nodes to run the *userconnect* command. When a node approaches the sink and receives a netcmd-message from the sink the userconnect command will be executed activating the node to send collected data to the sink.

6.1.3.2 Static

In the case of the static node scenario, a set of four nodes will be used, with one of them acting as a sink. The sink is added to the network by the end user to gather data from the other nodes. The three static nodes will be programmed differently that the sink node. In this scenario the nodes generate data individually. When the sink is added to the network it will issue a pulling collect causing the nodes to route their data through the network to the sink.

```
Nodes> dummy_data | write fileX
```

The above code will be run on the static nodes. The *dummy_data* command will generate data for this test scenario. In a real deployment this would be replaced with a command or several commands that read data from sensors. The generated data will be forwarded though a pipe to the *write* command that writes the data to file *fileX*.

This code example contained no specific active commands for using the disconnected operations provided in our shell solution. However, it is important that the nodes are programmed to handle data pulling requests. Therefore, in order for the sink to pull data from the nodes the nodes need to be programmed with code that will respond to such a pulling request.

```
Sink  > pullcollect fileX | write gathered_data
```

The sink node will be programmed with the above code. The *pullcollect* command will send a message, containing the name of the file to be pulled, throughout the network to activate the code that responds to the *pullcollect* command at each of the static nodes. The nodes will read the file specified, then send the contents of this fiole back to the node issuing the *pullcollect* command. Data is received by the sink via the pullcollect command and is forwarded through a pipe to the *write* command that writes the data in file *gathered_data*.

6.1.4 Monolithic implementation

The shell solution should be compared against a monolithic approach. A monolithic approach is defined by programming scenario specific code in a unified and single file manner. Using both approaches, the shell-based and monolithic, we will program the two scenarios mentioned in sections 6.1.3.1 and 6.1.3.2. In both the mobile and static scenario the sink and nodes will be programmed separately. Both scenarios will be programmed in the same environment as the command solution, i.e. the Contiki OS with appropriate libraries added.

6.1.4.1 Mobile

On the nodes two threads and handlers for incoming data will be running in the monolithic implementation. The first thread will handle data generation, write the data to a file locally on the node using Contiki's file system, and will also replicate this data to nearby nodes. The scheduling of when to generate data and when to shut off the radio will be handled in the first thread using a timer. The first thread will implement the same functionality as the *scheduling* command, the schedule server, the *dummy_data* command, parts of the *replicate* command, and parts of the *store* command.

One handler will receive replicated data from other nodes and write to the data-file. This handler will implement the same functionality as the part of the *replicate* command that deals with receiving data.

The other handler will wait for the end-user node to announce its presence by sending a message, and then start a second thread that reads the data-file and sends its contentsi to the end-user node. This handler and thread will implement the same functionality as the *userwait* command and the part of the *store* command that finally reads the content of the data file.

On the sink only one thread will run and it will handle scheduling of when to announce the presence of the sink. There will also be one handler that receives data and write to a local file that the end-user can access. This thread and handler will implement the same functionality as all the commands that are run on sink node, i.e. *discsched*, *netcmd*, *userconnect*, *collect*, and *write* command.

6.1.4.2 Static

On the node two threads will be running and one handler. The first thread will only generate the data and write it to a data file using Contiki's file system. This thread represents the *dummy*_data command and the *write* command that writes the data to a file.

The second thread is activated by the handler that will wait for the sink node to announce its presence in the network. When the second thread is activated it will read the data file and send it to the sink node. This thread and handler will implement the same functionality as the part of the *pullcollect* command that is activated by the sink node.

The sink node in this scenario will only run one thread and one handler. The thread will handle the announcement of the sink node in the network. The handler will receive data and write it to a file locally on the sink node that the end-user can access. This thread and handler will implement the same functionality as the *write* command and parts of the *pullcollect* command.

6.2 Results

In this section results from implementing the shell-based solution in different scenarios will be presented as well as results from the monolithic solution. First we will look at the two scenarios from a qualitative point of view and later from a quantitative view.

6.2.1 Qualitative

For the qualitative evaluation we will consider both the mobile scenario and the static scenario. Further we will consider how well the building block solution based on shell programming assists the programmer.

6.2.1.1 Building block solution

Using building blocks, also called modules, enables these modules to work together in several ways with few changes and little effort needed from the end-user. Additionally, since programs are seldom initially perfect, and the prerequisites and environment might change over time or bugs may be discovered in the code, there are two ways to deal with the need for changes. One could either discard the solution by replacing it with a complete new solution, or one can make changes to the current solution to a sufficient level. In the case of changing the code we will now discuss. By using separated modules it is not necessary to change the whole solution, but only one module. It is also possible to use several versions of modules for different cases without having to have complete copies of all other code. It is also easier to change only specific modules than digging into a big block of code and making changes spread throughout the code.

Unfortunately, the building block solution is not perfect for all cases. There may be a scenario that requires a unique set of commands that either does not match the existing commands (hence new commands are needed) or the scenario requireis a functionality that requires old commands to be heavily modified. Even in such a scenario it is still likely that a building block solution, in comparison to a monolithic one, would offer a simpler interface to work with and be a simpler solution to expand at a later stage. This is something we show when evaluating coupling types in section 6.2.2.3.

In the case of needing different policies for a module, we have proven, by implementing some of these different policies, that application specific demands of policies does not present a problem for the building block solution. Modules with different policies can be implemented either by different commands (e.g. *store* and *store2* commands in section 5.4.3.2), or the command can take an input argument that indicates the policy to use (e.g. the *replicate* command in section 5.4.3.3). The latter alternative should be used if both policies are expected to be used within the same application. The end user has to weigh pros and cons of either using only one of two similar commands (with different policies) or adding some code overhead in one command that can utilize both policies.

When considering the mobile and the static scenarios it is beneficial to use a building block solution in both cases. This is true, even if the end-user might want to use different methods of storing and collecting data, as they only need to change the modules rather than modify a large block of code. If the programmer who has programming experience enough to write well structured code the effort to change the code can be expected less than if the programmer was inexperienced.

The different types of commands provided in our solution cover a large range of problems and address many of the problems in programming disconnected operations. Even if some commands might be further developed, this set of commands offers a basic solution to the programmer for disconnected operations.

Using a shell to implement the building block solution has several benefits. The shell implementation’s strongest benefit is its simple and easy to use interface. This is the reason why a shell interface is still popular among many operating systems today, even though the idea originated in OSs that were created decades ago. Another advantage is that it fits well into a building block solution where commands can represent individual modules.

6.2.2 Quantitative

In this quantitative evaluation the commands will be evaluated in comparison with a monolithic code approach in the two deployment scenarios. It is important to keep in mind that the monolithic code has been written **after** the first command solution, therefore the programmer had a better understanding of the problems that needed to be addressed in these two scenarios. In addition, the programmer also had a better general knowledge of programming, both generally and specifically in Contiki.

In the following sections the energy consumption, code size, and design coupling of the solutions will be presented. We will not look into routing issues, loss of data, or retransmissions – even though these will affect the energy consumption – as these issues primarily concern areas that are outside the domain of this thesis, such as the type of media access and control (MAC) layer that is used, how packets are routed in the network, and other aspect of how the motes transmit packets.

During the energy consumption tests the following equipment and settings were used:

- 3 battery driven Sentilla nodes
- 1 Sentilla sink
- MAC protocol: X-MAC[52]
- Contiki OS version 1.41
- Channel: 22

6.2.2.1 Energy consumption

The energy consumption in the two scenarios for both the shell-solution and the monolithic solution have been monitored with Contiki's own software based on-line energy based estimation mechanism, the power profiler [53]. The power profiling in Contiki is based on energy consumption estimations of each hardware device on the mote. When these devices are used, e.g. radio unit or CPU, a predefined estimated value of energy consumption is added to the estimate of the system's total energy consumption. Even if the value from the power profiler is not exact, it still provides a good tool to measure the estimated energy consumption and differences between different solutions.

When comparing the energy consumption in the mobile scenario there are factors that we were unable to control, but that could alter the outcome of the test. One factor is the human factor; the tests were performed by moving and placing nodes by hand, thus the distance between nodes could vary. In addition, the time to move the nodes may not be reproduced in the separate tests. Another factor is network interference from other devices using this same radio spectrum. Unfortunately, these factors affecting radio performance can have a substantial effect on the total energy consumption, since the radio transceiver consumes the largest about of energy at each node.

The energy consumption in the following tests will be presented in watt seconds (Ws), where one Ws is equal to one joule (J) – the SI-unit (International Standard of Units) for energy.

6.2.2.1.1 Mobile scenario

In Figure 21 a graph of energy consumption over time is shown for the mobile scenario using the shell solution. All nodes were started at approximately the same time, e.g. within one second. In this scenario mote 1 was never near the sink.

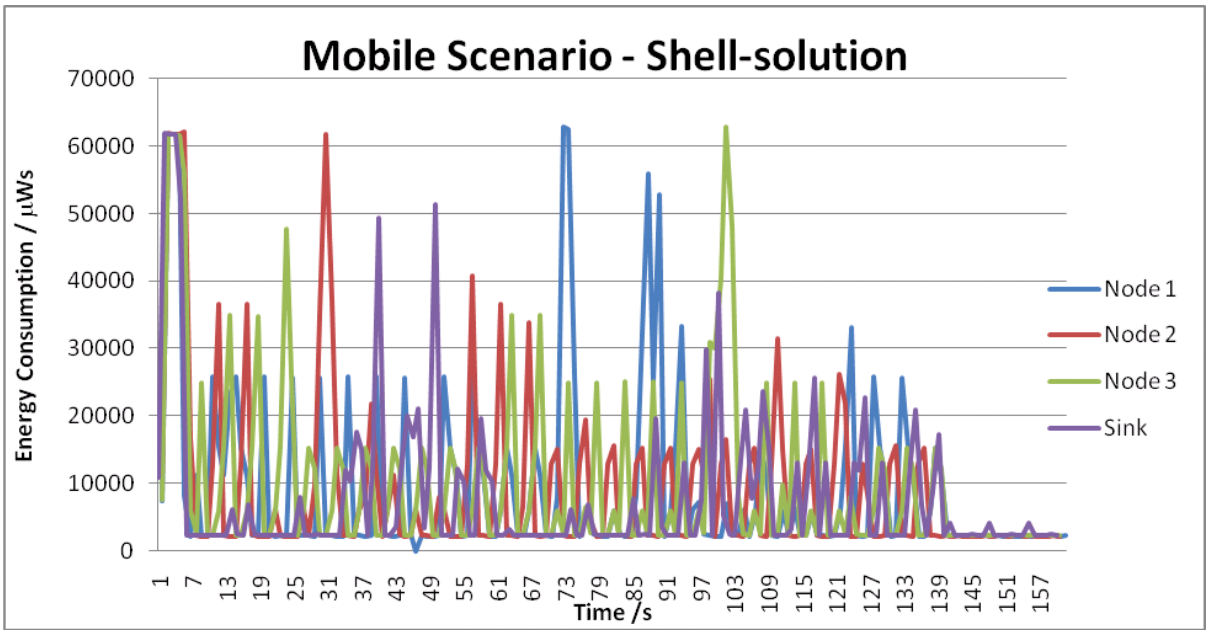

Figure 21: Power consumption over time for the mobile scenario using the shell-solution. The energy presented is the average energy during one second.

In the graph we can see a similarity in the start-up of all nodes where a lot of energy is consumed; with a peak energy consumption of approximately 61700 µWs. It is also possible to see that the energy consumption generally never decreases under around 2300 µWs. Table 3 shows the events that happened along with the approximate time of the event in the scenario using the shell solution. While it is possible to say that some peaks in the chart follows the events, some peaks could occur for reasons (e.g. network disturbance) other than the described events.

Table 3: Events for the mobile scenario using the shell solution.

Approximate Time (s)	**Event**
15	**Node2:** leaves the network and moves towards the sink
40	**Node2:** finished transmitting to sink, going towards network
70	**Node3:** leaves the network and moves towards the sink
100	**Node 3:** finished transmitting to sink, going towards network
140	**Node 1,2,3:** Radio shut off

In Figure 22 the energy consumption is shown for the mobile scenario using the monolithic version of the code. In this scenario mote 3 was never near the sink. As in the shell solution we see a similar pattern for start-up. In this case the peak energy consumption the same as the shell solution; around 61700 µWs. This is as expected, since it is not until after the start-up that the different configurations of the nodes will affect the way that the nodes behave, thus effecting their energy consumption. The threshold value that the consumption never decreases under is also the same as the shell solution, around 2300 µWs. Table 4 shows the

events and their approximate time of occurrence for the monolithic solution in the mobile scenario.

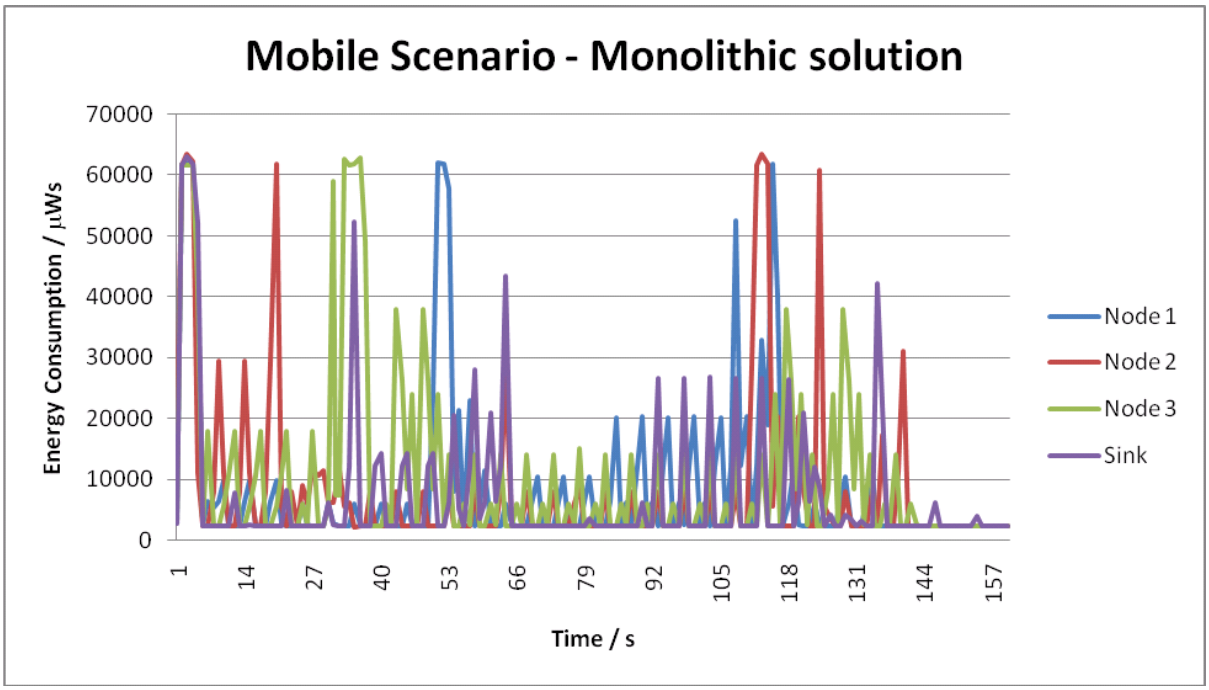

Figure 22: Energy consumption for the mobile scenario using the monolithic solution.

Table 4: Events for the mobile scenario using the monolithic solution

Approximate Time (s)	**Event**
20	**Node 1:** leaves the network and moves towards the sink
70	**Node 1:** finished transmitting to sink, going towards network
80	**Node 2:** leaves the network and moves towards the sink
130	**Node 2:** finished transmitting to sink, going towards network
140	**Node 1,2,3:** Radio shut off

From the two tests in the mobile scenario, using both the shell-based implementation and the monolithic implementation, we can see that they both operate within the same range of energy consumption. Due to many uncontrolled factors affecting the radio, it is not possible to give any final conclusions just by looking at the two graphs. However, we can compare these two sets of results.

Figure 23 the total energy consumption over time is shown by aggregating all nodes energy consumption (i.e., the power consumed by all nodes and the sink). It is difficult to see any systematic differences between the two solutions, however we can compare the highest and lowest values. By looking at the accumulated energy consumption over time it is easier to compare the two solutions against each other. This is shown in Figure 24

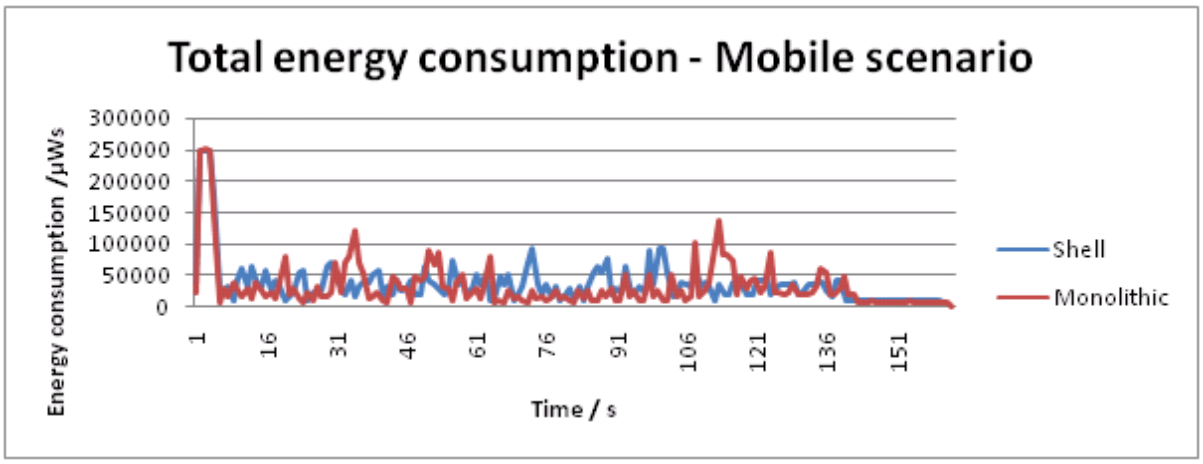

Figure 23: Energy consumption of all nodes for each solution in the mobile scenario

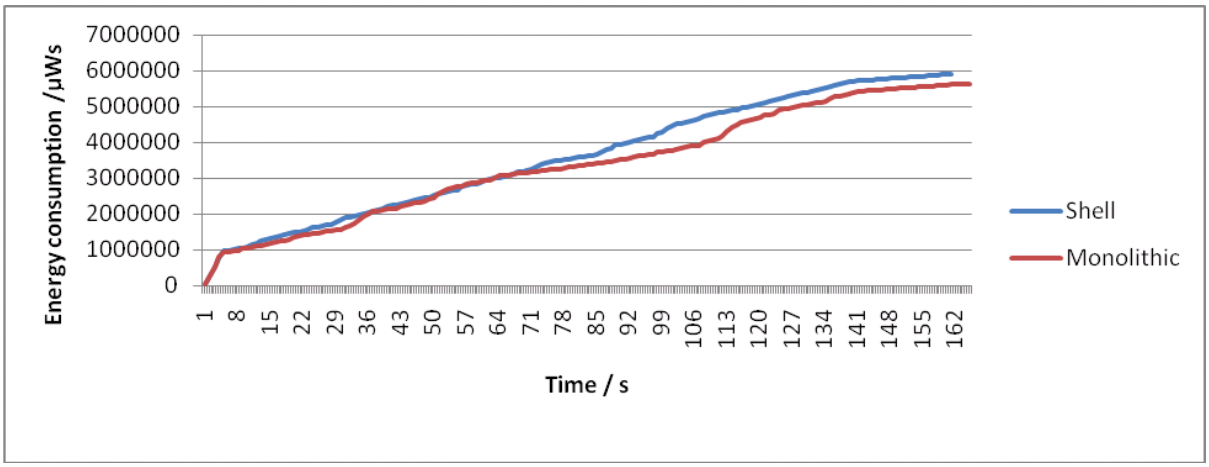

Figure 24: Accumulated energy consumption over time for the mobile scenario

In Figure 24 saw that in terms of the total energy consumption over time that the monolithic solution is more energy efficient. The shell solution has an average of 8.1% higher power consumption, ranging from 20.0% at most to -2.5% at a minimum. In Table 5 the total energy consumption from each note and all nodes together are shown. From these results we see that the total power consumption of all nodes in the shell solution is 6.2% larger in the shell implementation as compared to the monolithic solution.

Table 5: Mobile scenario power consumption. The difference is in comparison to the monolithic solution where the values show the shell solutions increase.

Mobile scenario	Shell (µWs)	Monolithic (µWs)	Difference
Node 1	*1 550 599	1 379 550	Average 5.5%
Node 2	1 492 689	1 314 894	
Node 3	1 617 247	* 1 722 221	
Sink	1 256 882	1 155 101	8.8%
Total	5 917 317	5 571 766	6.2%

*Motes that did not go towards the sink.

Summarizing the mobile scenario we can see that there is a difference in energy consumption between the two solution approaches. In the mobile scenario the monolithic solution has low total energy consumption. Since both solutions use the same libraries and protocols for sending and receiving packets the difference must lie in how these protocols and libraries are **used** by the implementation.

From these experiments we draw the conclusion that during these specific experiments the monolithic solution requires lower total energy consumption and lower energy consumption at each node. If the energy consumption of the nodes is the major concern in a mobile scenario, such as the one we have tested, then the monolithic solution is preferable to the shell-based solution – however, the difference might not be significant in a specific scenario. It is unclear if energy consumption could be further reduced by changing the routing algorithms and if this difference in power consumption would be greater than the difference in power consumption of these two different approaches for implementing a solution.

6.2.2.1.2 Monolithic scenario

Testing of the static scenario for the monolithic and shell solution was conducted under more similar circumstances than the mobile scenario, since there are fewer events in this scenario and there was no need for the experimenter to move the nodes about during the experiment. In both experiments concerning the static scenario the nodes generated the same amount of data before the sink node connected to the network and started collecting data. Still network interference and other external factors could affect the network's performance and these could be different in the two tests.

Figure 25 shows the energy consumption as a function of time in the static scenario using the shell solution. All of the nodes were started up at the same time and at approximately 108 seconds into the test the sink is powered on and connected to the network. The sink node has therefore no energy consumption up to this point in time.

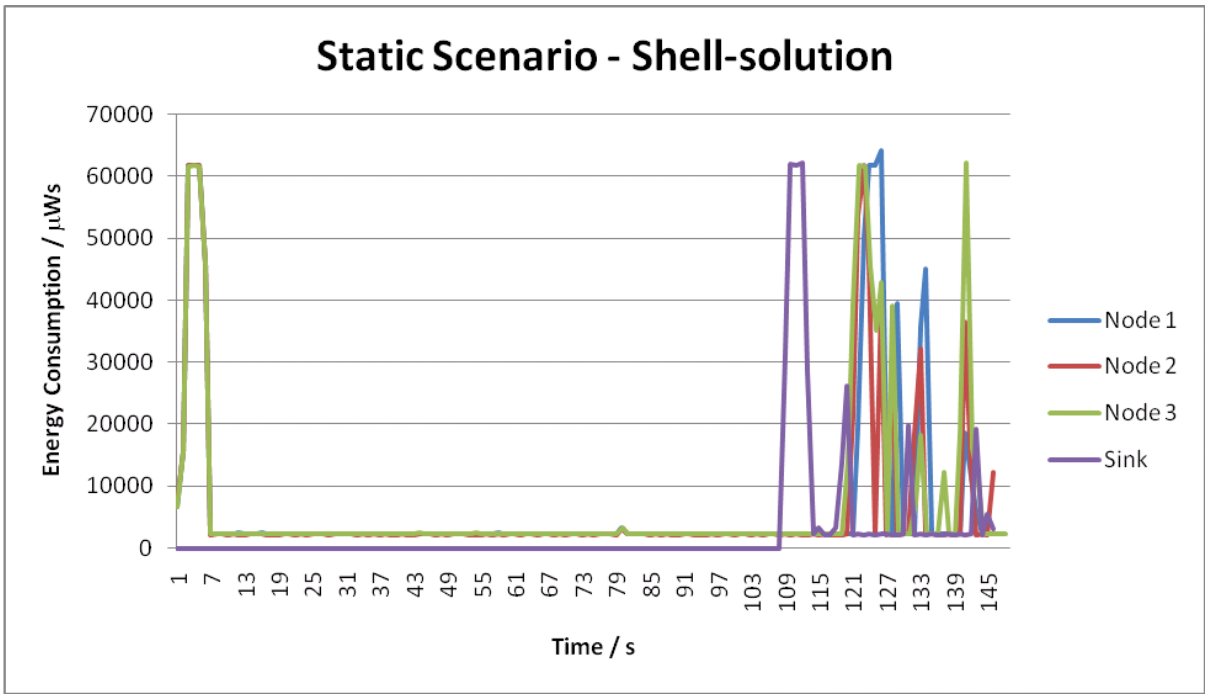

Figure 25: Energy consumption for the static scenario using the shell solution.

Figure 26 shows the energy consumption as a function of time for the static scenario using the monolithic solution. This test was performed under the same conditions and with the same choice of settings as the shell solution test. The combined (total) energy consumption of all nodes as a function of time for both tests is shown in Figure 27. In this plot we note that the shell solution has higher peak power usage. The minimal thresholds for both solutions are the same. Compared with the mobile scenario the initial energy consumption is higher in the mobile scenario than in the static scenario, this is due to the number of nodes started at the

same time. In the mobile all four nodes are started in the beginning, in the static the sink is started at a later stage, as can be seen in Figure 25 and Figure 26.

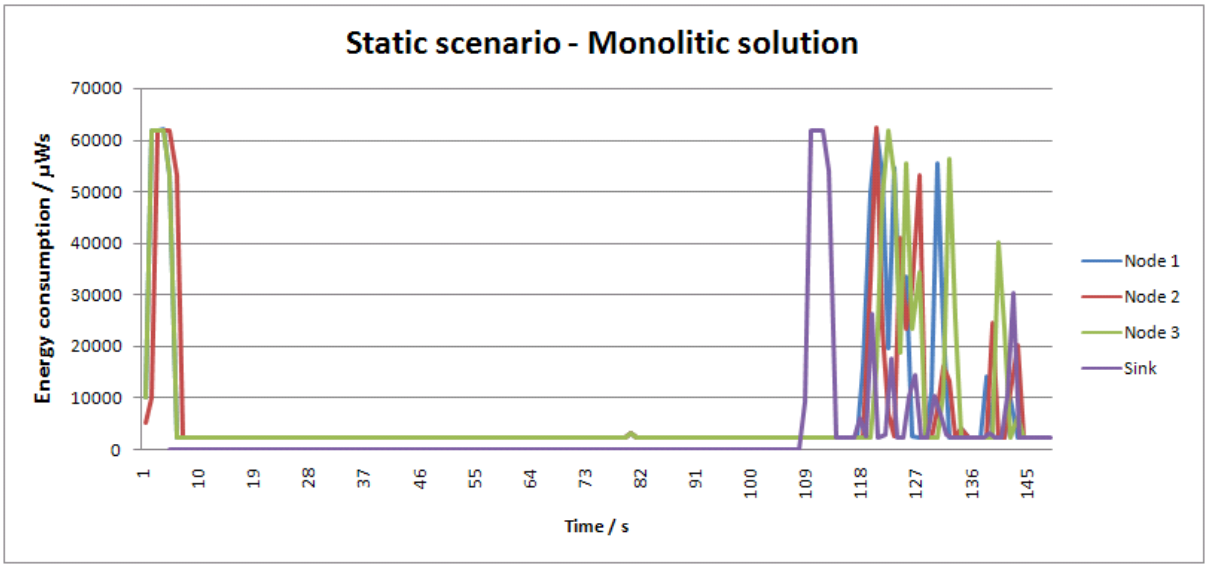

Figure 26: Energy consumption for the static scenario using the monolithic solution.

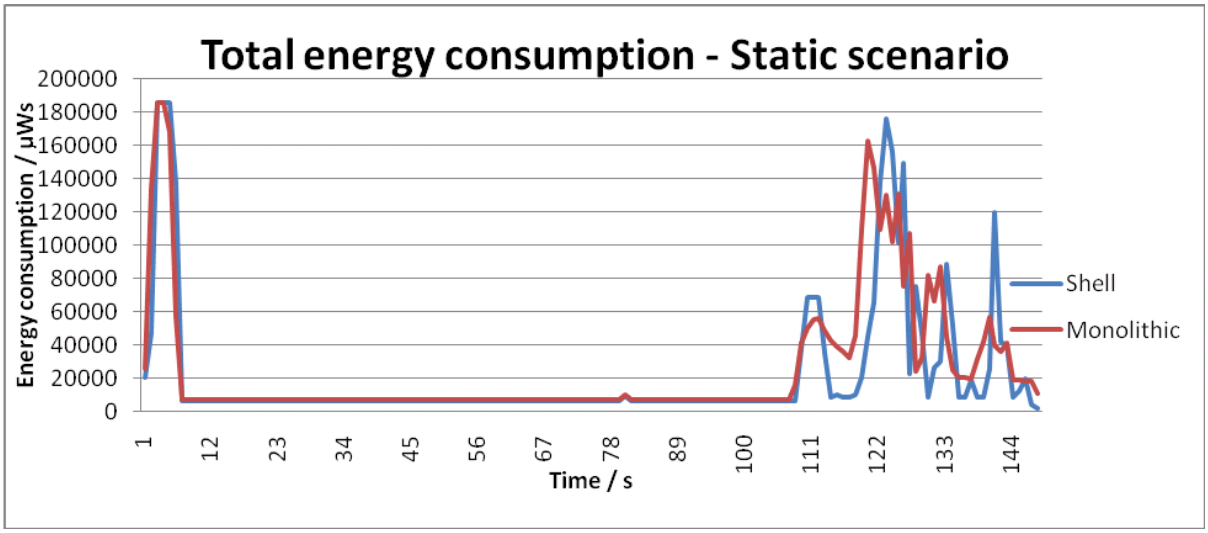

Figure 27: Energy consumption of all nodes for each solution in the static scenario

Looking at the total energy consumption over time, as shown in Figure 28, we can see that the monolithic solution that has a greater total energy consumption. Here the shell solution, in comparison to the monolithic solution on average consumes 3.2% less power (ranging from 2.0% to -17.9%). Table 6 shows the total energy consumption for each node and all the nodes combined. From these results we observe there is almost no difference between the power consumption of the different nodes, but the sink in the shell solution is 11.3% more energy efficient that the monolithic solution.

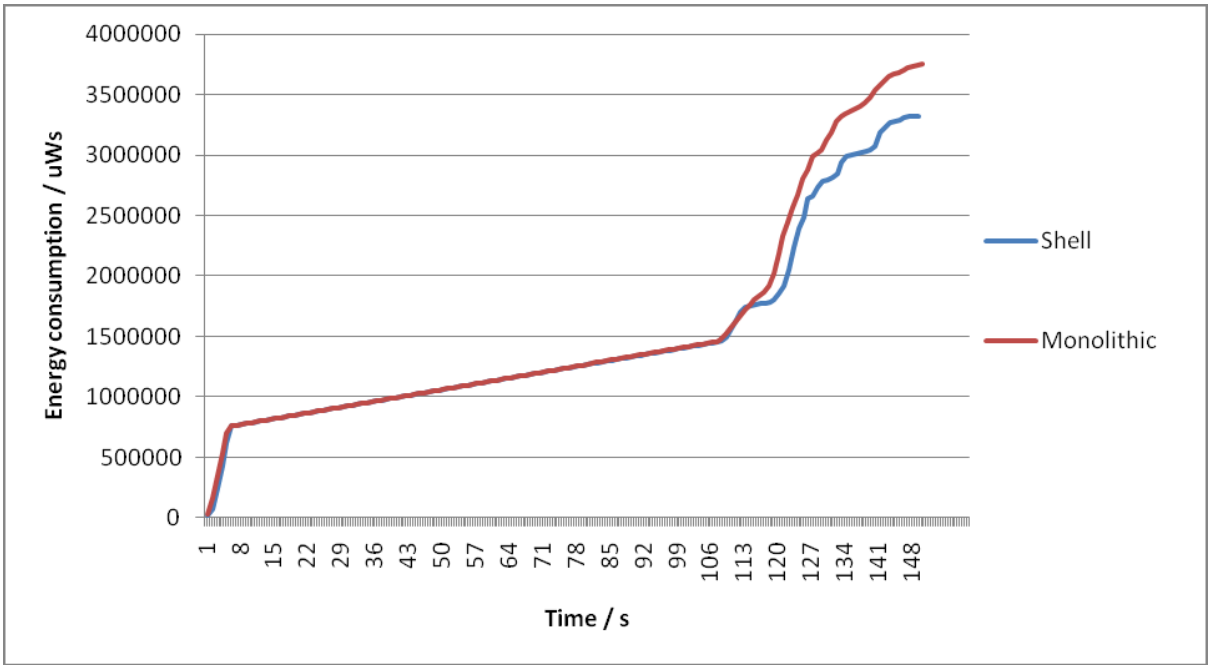

Figure 28: Accumulated energy consumption over time for the static scenario

Static scenario	Shell (μWs)	Monolithic (μWs)	Difference
Node 1	976 374	973 845	Average. 0.5%
Node 2	915 041	915 021	
Node 3	1 031 445	1 018 298	
Sink	399 314	450 269	-11.3%
Total	3 322 174	3 357 433	-1.1%

Table 6: Static scenario power consumption

Summarizing the results from the static test we can see that during the time when no radio communication between nodes or sink was active both solutions had very similar power consumption. Despite the fact that the shell-solution uses more processes (one for each command) and also the scheduler server is always running in the background (even if no tasks are to be scheduled) there seems to be no difference in energy consumption. What seems to matter is how the radio on the nodes is used for sending and receiving packets. Additional reasons for differences could be local disturbances as well as details of the implementations. From these results we can conclude that there is almost no difference between these two approaches to a solution in the total power consumption in the static case, although there was roughly a 6% difference in the mobile case.

Because of the difficulty to replicate these scenarios we have made further tests to see the difference in radio characteristics for similar scenario to our static scenario. These tests were made with three nodes sending data to a sink. We looked at the total energy consumption and later calculated the mean total energy value of all samples. With this value we calculated the difference for each sample. We calculated the confidence value for this difference with a confidence level of 95%. The confidence interval with a 95% level means that 95% of all measured values, are within a certain interval. This interval can later be used to assume that in following measurements 95% of these measurements should also be within this interval. The samples of difference can be seen in Table 7 and our results can be seen in Table 8. Having an interval of 0 ± 11.4 % shows that the differences for the two scenarios (-1.1% and 6.2%) are within the interval and therefore within the span of normal deviation of the radio

energy consumption. Therefore we can not draw a conclusion that the monolithic solution or shell based solution is significantly better than the other in regard to energy consumption.

Table 7: Measurements

Difference (%)
-6.2
-23.3
17.6
26.3
-26.0
-15.6
17.1
-17.9
14.6
-5.1
18.4

Table 8: Confidence interval calculation results

Parameters / formulas	**Values**
Mean value:	0
Standard deviation:	19.2
T-distribution (95% level):	11.4
Interval	**0 ± 11.4** (-11.4 – 11.4)

6.2.2.2 Code size

When comparing the code size of each solution we have looked both at the number of lines of code and at the number of bytes actually programmed in the node, e.g. the total size after the code has been optimized and compiled.

Comparing code size is not as trivial as it might seem. A programmer might be concerned about how much code needs to be written to implement a solution using the two different approaches. Naïvely we might assume that the programmer writes a monolithic implementation from scratch for every scenario, however, this would ignore the fact that much of the code from one implementation to the next can be reused. Due to open source efforts more and more code is being reused by others (in addition to reuse by a given programmer). With the shell-solution much of the code is already implemented in the modules, and the programmer only needs to combine these modules to create their program. In this comparison we count the number of lines of code (LOC) using the tool CLOC [54] version 1.08. CLOC is a Perl script program that takes code file as input and calculates the LOC, blank lines, and commented lines. In our comparison we will only look at LOC.

Table 9: Shows bytes of code

Scenario	**Shell solution**	**Monolithic - Sink**	**Monolithic - Mote**
Static	893	100	148
Mobile		121	203

The shell solution has a total of 893 LOC and the monolithic solution has a total 572 LOC, hence the shell solution is 56% larger than the monolithic solution. The static scenario required 248 LOC and the mobile scenario 324 LOC. The code for the shell solution is larger for several obvious reasons:

1. The shell solution has more overhead since each command is implemented as a process with process names, input arguments, local variables, etc. The shell solution has a minimum of 11 processes and handlers, while the monolithic solution uses between 3-5 processes and handlers.
2. Where the monolithic solution only has the functionality needed for this specific scenario, the shell solution offers the full functionality of all of the commands, even if some of the commands are not used.
3. With the shell solution it is possible to program nodes for other scenarios besides these specific mobile and static scenarios. The monolithic solution requires a new implementation for each scenario, as has been done for the mobile and static scenario. The shell solution is re-usable and generic. It should be noted that the number of LOC needed to implement a certain scenario using shell-commands is of the order of 1-10 lines while the monolithic solution requires hundreds of lines. Specifically in these two scenarios no more than two lines of code were used on each node.

After looking at the code size in terms of lines of code, we compared the actual number of bytes programmed in the nodes. This was done by first compiling the code using the standard GCC compiler provided in the Contiki development environment. After compilation the code is programmed in the nodes and the actual number of bytes programmed for the node is presented in Table 10. The compiler used was a modified GCC compiler, msp430-gcc, that was run with option ”-Os” that tells the compiler to optimize for the smallest code size.

Table 10: Shows bytes programmed to the nodes

Scenario	Shell solution	Monolithic - Sink	Monolithic - Mote
Static	44 708	29 600	29 112
Mobile		29 784	29 354

Calculating the difference between the shell solution and the different code in the four monolithic solutions the shell solution requires an average of 52% more bytes per node. However, the shell solution uses the same image and code independent of the node type (sink, mobile note, etc.) hence the same number of bytes is programmed in all nodes. If the programmer knew that some commands would never be run on some nodes, e.g., the sink might not need commands for using sensors if no sensors exist, he could customize the image for each type of nodes making it smaller, thus saving memory space. In environments offering dynamic runtime linking it is also possible for the programmer to upload some code to the nodes and at a later stage upload more commands when needed.

The reasons why the shell solution results in more bytes after compilation is the same reasons as why more LOC are needed. The second reason, of the three reasons mentioned above, is the biggest contributor to the shell solution requiring more bytes to programmed. Some of the code will be optimized by the compiler, but the compiler does not know which commands are going to be used on the specific nodes and therefore has to include all commands. This has the effect of putting similar commands together in libraries instead of linking commands one by one. The results is that the user only sees the functionality at the library level and not at the command level when linking libraries, hence linking whole libraries even if some commands are not used or planned to be used. Therefore the image size is larger for the shell solution. If the programmer knows that some commands will never be used in a certain scenario, then these commands could be commented out in the code and the

compiler will not emit code for them, thus reducing the number of bytes that have to be programmed in the nodes. This could be achieved by using conditions in the code that the compiler would act upon. Preferably all conditions could be set in a separate configuration file so no change would have to be made to the actual code. This would result in no memory overhead at the nodes, at the cost of a slight increase in compile time.

The overhead of the commands for programming disconnected operations is calculated by first compiling the nodes with shell support and our commands, then only programming with shell support without our commands and calculating the overhead as the difference in the number of bytes. The overhead of using our commands in the shell solution is shown in Table 11. It covers all the commands written specifically for disconnected operations. If the functionality of all the commands is needed only 7704 bytes are added to the image no matter what other shell commands are used. Therefore a significant portion of the overhead is due to the actual shell run-time, not in our commands.

Table 11: Disconnected operation commands overhead

Shell with Disconnected Operation support	**Shell**	**Difference**
44 708	37 004	7 704

Only compiling nodes with support to run shell commands, without adding any actual commands except the build in commands (i.e., *help*, *kill*, *killall*, and *null* command) the image would then be 29 286 bytes. This is about the same image size as the monolithic solution, but offers the end user the ability to choose *any* existing commands to his or her liking and that best suited for the scenario- at the cost of adding the size for these commands on top of this base image size. Adding all of the commands for disconnected operations would give a total image size of 36 990 bytes.

An additional consideration is the number of bytes of memory that have to be used at run-time since in the processor that has been used for these experiments one of the constraints is the amount of memory that is available for data.

6.2.2.3 Coupling

Here we will look at seven types of coupling for the different solutions. The types of coupling used were defined by Stevens, et al. [55]. Seven types of coupling will be used, ranging from tight to loose. These types are presented in Table 12. A tight couple design can be considered to be more difficult to debug and maintain while a loosely coupled design is easy to understand and re-use.

Table 12: Description of coupling types for software modules

Coupling Type	**Description**
Content (tight)	One module relies on the internal working of another. Changing one module requires changes in the other as well.
Common	Two or more modules share some global state, e.g., a variable.
External	Two or more modules share a common data format.
Control	One module controls the flow of another, e.g., passing information that determines how to execute.

Coupling Type	Description
Stamp	Two or more modules share a common data format, but each of them uses a different part with no overlapping.
Data	Two or more modules share data through a typed interface, e.g., a function call.
Messages (loose)	Two or more modules share data through an untyped interface, e.g., via message passing.

In this case there are some different cases of granularity that can be applied to the shell and monolithic solution. In shell solution a command can be considered a module, as well as a process or a single function. Looking at shell coupling types we have chosen to look at command granularity, since it is the most intuitive approach because shell commands are what programmers ultimately use. In the monolithic solution a module can be defined as a process or a single function. For the monolithic solution the coupling types will be compared at process granularity. Other granularities for both solutions have been considered, but are not considered here in our comparison between the two solutions. In this comparison we look at the solution as a whole and do not need finer granularity in the types of coupling.

Table 13: Design coupling for the two solutions

Coupling Type	Monolithic solution	Shell solution
Content	No	No
Common	Yes	No
External	No	No
Control	Yes	Yes
Stamp	No	No
Data	No	No
Message	Yes	Yes

The monolithic solution has common coupling where several processes share global variables, such as timers, flags and file pointers. Messages between sink and nodes control when uploading of data between a node and a sink is supposed to occur, thus both control and message coupling types are present in the monolithic solution.

The shell solution has control coupling, since there are commands that control the execution of other commands, such as: *userwait* and *discsched*. Also message coupling is present since messages through pipes are used between almost all commands and also messages are sent over the network between nodes.

Analyzing these different coupling types it is expected that the monolithic solution is more complex and has more and tighter coupling types. This is where a shell solution benefits most in comparison to a monolithic solution. By using commands as modules it is easier to program nodes directly, re-use code where commands do not necessarily have to be changed, and to extend the solution by adding more commands. The tighter coupling types in the monolithic solution makes it more difficult to use, to re-use, and to expand.

7 Conclusions and Future work

In this chapter some conclusions of the thesis will be briefly discussed. These are considered in the context of the initial goals and the experimental results. Subsequently possible future work that could build upon this thesis will also be described.

7.1 Conclusions

The main goal for this thesis was to devise a solution that would help programmers to program disconnected operations for WSNs. This was done by using the existing shell in the Contiki OS and writing additional commands that would help programmers both generally and specifically with the targeted disconnected operations. To determine if this solution really helps programmers an experiment could have been done with a number of different programmers who each had to use and evaluate the commands for some time in implementing real life deployments of WSN. However, such an evaluation was not feasible within the scope of this master's thesis project, hence we simply *assume* that using the well known approach of shell commands would meet the goal.

Based upon the results of our evaluation there is no obvious relation between power consumption and our shell solution. In both scenarios the power consumption was very similar except when the radio was used, then both scenarios gave different results indicating that the shell solution and the monolithic solution differed in power consumption – but without a clear indication of whether the differences are significant. However, there is a large difference in the power consumption in the case of the mobile scenario that might favour the monolithic solution.

The monolithic solution requires less code than the shell solution, but at the cost of reduced flexibility of the code. The shell solution requires less code to be written to implement different scenarios. The ability to incrementally modify the code by sending only changes to the nodes was not evaluated, but there are some indicatations that this might result in an advantage for the shell based solution.

Finally the types of coupling that the shell solution offers in comparison to the monolithic illustrate the advantage of the shell solution. Both solutions share some coupling types, but the monolithic has one more coupling type with tighter coupling indicating that the shell provides a simpler programming interface solution.

A general and obvious difference between the solutions is that when programming nodes using a monolithic solution code has to be written for each new scenario. This is not the case for the shell solution since commands can be re-used and used in different ways to fulfil the needs that a new scenario puts on the programming solution. However, no systematic analysis of potential code re-use for different scenarios was done for code from a given monolithic solution.

7.2 Personal reflections

As a result of doing this thesis project I have realised that WSN is a broad area with many different application areas that all might affect the requirements of both hardware and software in many ways. Creating general solutions for WSNs is difficult, but is needed to increase the interest in and use of WSNs.

To encourage others to start working in this area I think that it would be beneficial for the WSN-community if their work aimed at making WSNs easier to use, with respect to

programming, maintenance, and deployment of solutions. The easier it is to use WSNs the more people will do so, resulting in WSNs becoming more interesting both for research and to industry. The increased interest will lead to more people trying to improve WSNs from many aspects.

If I were to redo this thesis project more time would have been spent on further improving the resources usage of commands, such as CPU usage, memory usage, and code size. I would also have investigated if there were more commands that could be used for disconnected operations.

7.3 Future work

A step to further help programmers in programming disconnected operations would be to implement our shell solution on other platforms that also use shell commands. Such platforms have been discussed in section 2.2.1, such as TinyOs and LiteOs - as both provide the possibility to program nodes through shell commands.

The commands provided in this solution could be further expanded and improved. Additional versions of commands with different policies could be implemented or commands could have added functionality to make them more general, hence becoming suitable for more scenarios. For example, the scheduling command would benefit from another parameter defining each task's priority.

Finally there is another way of using shell commands than has not been investigated in this thesis. By extending the shell commands into a scripting language, or providing such an environment for commands to be programmed in, the programming interface to the programmers could be improved. Such an extension would be beneficial for more complex scenarios where the normal usage of commands and pipes would not suffice.

References

[1]. **I.F. Akyildiz, W.Su, Y. Sankarasubramaniam, and E. Caylirci.** *Wireless sensor networks: a survey.* Computer Networks (Elsevier) Journal, pp.393-422, March 2002.

[2]. **Kay Römer and Friedemann Mattern.** *The Design Space of Wireless Sensor Networks.* IEEE Wireless Communications, volume 11, number 6, pages 54-61, December 2004.

[3]. **Willial M. Merrill, Fredric Newberg, Katht Sohrabi, William Kaiser, and Greg Pottie.** *Collaborative networking requirements for unattended ground sensor systems.* In proceedings of IEEE Aerospace Conference. Volume 5, pages 5_2153 – 5_2165, March 8-15, 2003.

[4]. **Kirk Martinez, Royan Ong, and Jane Hart.** *Glacsweb: a sensor network for hostile environments,* Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, Santa Clara, USA, pages 81-87, 4-7 October 2004.

[5]. **Henrik Brånstad.** *Smarta bojar håller koll på klimatet.* 2, VinnovaNytt, issue 2, page 12, 2007.

[6]. Argo - part of the integrated global observation strategy. [Onlinex] [Cited: 16 March 2009.] http://www.argo.ucsd.edu

[7]. **Ugur Cetintemel, Andrew Flinders, and Ye Sun.** *Power-Efficient Data Dissemination in Wireless Sensor Networks.* Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access, San Diego, CA, USA, September 19, 2003

[8]. **Joseph M. Kahn, Randy H. Katz, and Kristofer S. J. Pister.** *Emerging Challenges: Mobile Networking for 'Smart Dust'.* Journal of communication and networks, vol. 2, no3, pp. 188-196 (13 ref.), 2000.

[9]. **Chalermek Instanagonwiwat, Deborah Estrin, Ramesh Gocindan, and John Heidemann.** *Impact of Network Density on Data Aggregation in Wireless Sensor Networks.* Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), page 457, 2002.

[10]. **Nicolas Tsiftes.** *Using Data Compression for Energy-Efficient Reprogramming of Wireless Sensor Networks.* Master Thesis in Computer Science, Stockholm University, Stockholm, 2007.

[11]. **Thiemo Voigt, Nicolas Tsiftes, and Zhitao He.** *Remote Water Monitoring With Sensor Networking Technology.* ERCIM News, issue 76, page 39-40, 2009.

[12]. **Geoff Werner-Allen, Konrad Lorincz, Jeffrey Johnson, Jonathan Lees, and Matt Welsh.** *Fidelity and Yield in a Volcano Monitoring Sensor Network.* Operating Systems Design and Implementation, Proceedings of the 7th symposium on Operating systems design and implementation, Seattle, Washington , pages 381 – 396, 2006.

[13]. **Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson.** *Wireless sensor networks for habitat monitoring.* Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications, Atlanta, Georgia, USA, pages 88 – 97. ACM, New York, NY, USA, 2006.

[14]. **Wolfgang Lindner and Samuel Madden**. *Data Management Issues in Disconnected Sensor Networks.* In Proceedings of Informatik 2004. 34th Jahrestagung der Gesellschaft fÎr Informatik e.V., Bonn, Germany, Vol. 2, September 2004, pages 349-354.

[15]. **Atul Prakash, Beng Heng, Billy Lau, and Vineet Kamat.** *Dependable opportunistic communication in a multitier sensor network architecture.* Workshop on Research Directions in Situational Self-managed Proactive Computing in Wireless Ad-hoc networks, St. Louis, March 1-3, 2009.

[16]. **Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, and Daniel Rubenstein.** *Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet.* ACM SIGARCH Computer Architecture News Volume 30 , Issue 5, pages 96 - 107, December 2002.

[17]. **David Jea, Arun A. Somasundara, and Mani B. Srivastava**. *Multiple Controlled Mobile Elements (Data Mules) for Data Collection in Sensor Networks,* 2005 IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS '05), June 2005.

[18]. **Tara Small and Zygmunt J. Haas.** *The Shared Wireless Infostation Model - A New Ad Hoc Networking Paradigm (or Where there is a Whale, there is a Way),* ACM MOBIHOC'03, Annapolis, Maryland, June 1-3, 2003.

[19]. **Kay Römer, Oliver Kasten, and Friedemann Mattern.** *Middleware Challenges for Wireless Sensor Networks.* ACM SIGMOBILE Mobile Computing and Communications Review, volume 6, issue 4, pages 59-61. ACM, New York, NY, 2002.

[20]. Wikipedia – Event-driven programming. [Onlinex] [Cited: 20 April 2009.] http://en.wikipedia.org/wiki/Event-driven_programming

[21]. **Adam Dunkels, Björn Grönvallm, and Thiemo Voigt.** *Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors.* LCN, Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, pages 455-462, IEEE Computer Society Washington, DC, USA 2004.

[22]. eCos. [Onlinex] [Cited: 16 April 2009.] http://ecos.sourceware.org/about.html.

[23]. **Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He**. *The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks.* In Proceedings of the 7th International Conference on Information Processing in Sensor Network, pages 233-244. IEEE Computer Society, Washington. 2008.

[24]. MagnetOS. [Onlinex] [Cited: 16 April 2009.] http://www.cs.cornell.edu/People/egs/magnetos/

[25]. **Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Aadam Torgerson, and Richard Han.** *MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms.* Mobile Networks and Applications, volume 10, issue 4, pages 563-579. Kluwer Academic Publishers, Hingham, MA, August 2005.

[26]. µC/OS-II RTOS. [Onlinex] [Cited: 16 April 2009.] http://www.micrium.com/products/rtos/kernel/rtos.html

[27]. Nano-RK. [Onlinex] [Cited: 16 April 2009.] http://www.nanork.org/

[28]. **Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden.** *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.* IEEE Journal of Selected Areas in Communications, volume 14, number 7, pages 1280-1297, 1966.

[29]. Nut/OS – Modular RTOS. [Onlinex] [Cited: 16 April 2009x] http://www.ethernut.de/en/firmware/nutos.html

[30]. **Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister.** *System Architecture Directions for Networked Sensors.* ACM SIGPLAN Notices, Volume 35, Issue 11, pages 93-104. ACM, New York, NY, 2000.

[31]. **Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali.** *Protothreads: Simplifying event-Driven Programming of Memory-Constrained Embedded Systems,* Conference On Embedded Networked Sensor Systems, Proceedings of the 4th international conference on Embedded networked sensor systems, pages 29-42, ACM, New York, NY, USA, 2006.

[32]. **Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt.** *Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks.* Swedish Institute of Computer Science, Kista, 2006.

[33]. **Adam Dunkels.** Contiki Crash Course. Wireless sensor network programming: an introductio. [Online course notesx] http://www.ee.kth.se/~mikaelj/wsn_course.shtml. Swedish Institute of Computer Science, Kista, 2008

[34]. Tajana's corner of the web. [Onlinex] [Cited: 20 April 2009x] http://www.cse.ucsd.edu/~trosing/index.html

[35]. **Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong.** *TinyDB: An Acquisitional Query Processing System for Sensor Networks*. ACM Transactions on Database Systems (TODS), volume 30, issue 1, pages 122-173. ACM, New York, NY, March 2005,

[36]. **Luca Mottola and Gian P. Picco.** *Programming Wireless Sensor Networks: Fundamental Concepts and State-of-the-Art*. Politecnico di Milano, Milano, Italy, November 2008.

[37]. **David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler.** *The nesC Language: A Holistic Approach to Networked Embedded Systems.* In Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.

[38]. **Luca Mottola and Gian P. Picco.** Logical neighborhoods : A pro*gramming abstraction for wireless sensor networks.* ss IEEE International Conference on Distributed Computing in Sensor Systems No2, volume 4206, pages 150-168. Springer Berlin / Heidelberg, 2006.

[39]. **Qiang Wang, Yaoyao Zhu, and Liang Cheng.** *Reprogramming Wireless Sensor Networks: Challenges and Approaches.* IEEE Network, volume 20, issue 3, pages 48-55, May-June 2006.

[40]. **Rabin Patra and Sergiu Nedveschi.** *DTNLite: A Reliable Data Transfer Architechture for Sensor Networks.* Applications, Technologies, Architectures, and Protocols for Computer Communication, Proceedings of the 2006 SIGCOMM workshop on Challenged networks, pages 253 – 260, ACM, New York, NY, USA, 2003.

[41]. **Liqian Luo, Chengdu Huang, Tarek Abdelzaher, and John Stankovic.** *EnviroStore: A Cooperative Storage System for Disconnected Operation in Sensor Networks*. INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE Publication, Anchorage, AK, pages 1802-1810, 6-12 May 2007

[42]. Disruption Tolerant Networking. [Onlinex] [Cited: 10 March 2009] http://www.darpa.mil/sto/solicitations/DTN/.

[43]. Delay Tolerant Networking Research Group. [Onlinex] [Cited: 10 March 2009] http://www.dtnrg.org.

[44]. **Martin Lukac, Lewis Girod, and Deborah Estrin.** *Disruption tolerant shell.* Applications, Technologies, Architectures, and Protocols for Computer Communication. Proceedings of the 2006 SIGCOMM workshop on Challenged networks, Pisa, Italy, pages 189 – 196. ACM, New York, NY, USA, 2006. ISBN: 1-59593-572-X.

[45]. Contiki 2.X Reference Manual. Generated by DoxyGen 1.4.1. [Generated: 2 July 2007.] http://www.sics.se/~bg/telos/contiki-2.x-snap11.pdf.

[46]. **Prabal Dutta and David Culler**. *Mobility Changes Everything in Low-Power Wireless Sensornets.* In Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII), Monte Verita, Switzerland, May 18-20, 2009.

[47]. WIDAGATE: Wireless Data Acquisition in Gas Turbine Engine Testing. [Online] [Cited 20 Nov 2009x] http://www.ee.ucl.ac.uk/research/WIDAGATE

[48]. UNIX Shell. [Online] [Cited 20 Nov 2009] http://en.wikipedia.org/wiki/Unix_shell.

[49]. **Olin Shivers.** *A Universal Scripting Framework or Lambda: the ultimate "little language"*. Lecture Notes In Computer Science; Vol. 1179. Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security, pages 254-265. Springer-Verlag, 1996. ISBN:3-540-62031-1

[50]. WildSensing: A Hybrid Framework of Mobile and Sensor Nodes for Wildlife Monitoring. [Onlinex] [Cited 2 Nov 2009] http://www.cl.cam.ac.uk/research/srg/metos/wildsensing/index.html

[51]. **Liqian Luo, Qing Cao, Chengdu Huang, Tarek Abdelzaher, John A Stankovic, and Michael Ward.** *EnviroMic: Towards Cooperative Storage and Retrieval in Audio Sensor Networks*. ICDCS. Proceedings of the 27th International Conference on Distributed Computing Systems, page 34. IEEE Computer Society, Washington, DC, USA, 2007. ISBN: 0-7695-2837-3.

[52]. **Michael Buettner, Gary Yee, Eric Anderson, and Richard Han**. *X-MAC: AA Short Peamble MAC Protocol For Duty-Cycled Wireless Sensor Networks.* Technical Report, Department of Computer Science, University of Colorado at Boulder, CU-CS-1008-06, May 2006. http://www.cs.colorado.edu/department/publications/reports/docs/CU-CS-1008-06.pdf

[53]. **Adam Dunkels, Fredrik Österlind, Nicolas Tsiftes, and Zhitao He.** *Software-Based On-Line Energy Estimation for Sensor Nodes.* Proceedings of the 4th workshop on Embedded networked sensors, Cork, Ireland, pages 28 – 32. ACM, New York, NY, USA, 2007. ISBN: 978-1-59593-694-3.

[54]. CLOC – Count Lines of code. [Onlinex] [Cited 20 Nov 2009] http://cloc.sourceforge.net/

[55]. **Wayne P Stevens, Glenford James Myers, and Larry L. Constantine.** *Structured Design.* Classics in software engineering, pages 205 – 232. Yourdon Press, Upper Saddle River, NJ, USA, 1979. ISBN: 0-917072-14-6

TRITA-ICT-EX-2009:220

www.kth.se