Master's Thesis

A Multi-Agent Middleware with a Power Management Console for Efficient Wireless Sensor Networks

Reinhard Berlach, Bakk.techn.

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Begutachter: Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Betreuer: Dipl.-Ing. Philipp M. Glatz, Bakk.techn.

Graz, May 2011

Kurzfassung

Drahtlose Sensornetzwerke(WSNs) haben dieselben Probleme wie alle anderen tragbaren Geräte auch. Die Leistungsanforderungen der Hard- und Software steigen schneller als die Leistungsfähigkeit der Energieträger. Damit ist es vor allem für lange Laufzeiten, wie sie bei WSNs gefordert sind, notwendig, Strategien zu entwickeln, wie man Energie sparen kann.

Dies kann man erreichen, indem man den Knoten in einem Netzwerk mehr als eine Aufgabe überträgt. Um die Vielzahl der verschiedenen Applikationen managen zu können, wird eine Middleware implementiert. Diese kann mit Hilfe von Multithreading-Technologien die Aufgaben unabhängig voneinander ausführen. Diese einzelnen Tasks oder auch Agents genannt sind in der Middleware nicht mehr an einen Knoten gebunden.
Neben dieser Multi-Agent-Umgebung implementiert die Middleware auch ein Powermanagement, um die beschränkte Ressource *Energie* effizienter zu verwalten. Dieses steuert je nach Zustand des Netzes die Agenten. Diese können vom Powermanagement von einem Knoten zu einem anderen bewegt werden. Dies ermöglicht es das *globale* Energiebudget des Netzes effizienter auszunutzen. Ebenso kann das Powermanagement den Dutycycle im Netzwerk verändern, um mehr Energie zu sparen.
Die Strukturierung des Netzes in verschiedene Virtuelle Organisationen, ermöglicht es dem Verwalter des WSN, die unterschiedlichen Anforderungen an das Netzwerk leichter zu erfüllen.

Die Middleware erlaubt es dem User einfach und schnell verschiedene Aufgaben an unterschiedliche Knoten im Netzwerk zu verteilen. Diese können zur Laufzeit des Netzwerkes umdisponiert werden. Durch die Möglichkeit, den Dutycycle zur Laufzeit zu verändern, bietet die Middleware ein Service an, den Energieverbrauch des Netzwerkes an die Datenlast anzupassen. Dies ermöglicht eine effizientere Nutzung der Energie.

Abstract

Wireless sensor networks (WSNs) have similar problems as other portable devices. The energy requirements of hard- and software are much larger than the possible energy that comes from the power source. To achieve long deployment times, which are needed for WSNs, strategies to conserve energy must be developed.

This can be done, through deploying more than one task to each node in the network. To manage all the different applications, a middleware is implemented. This middleware runs these functions in a multi-threading environment. This makes it possible to call each task independently. All these tasks, called agents, are no longer bound to one specific node.
Side by side to this multi-agent system the middleware offers also a power management service to administrate the rare resource *energy*. This controls the agents depending on the state of the network. These agents can be moved from the management from one node to another. This allows using the *global* energy of the network more efficiently. Second task of the power management is to control the duty cycle. That can be changed while the network is running.
The structuring of the network with virtual organisations offers the administrator a simple way to manage the requested quality of service.

The middleware allows the user fast and easy to deploy different tasks (agents) to the different nodes in the network. This deployment can be changed during runtime. Through the possibility to change the duty cycle at runtime, the middleware presents a service to adjust the power consumption to the network load. This allows saving much of the rare energy.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, May 2011 ....

(Reinhard Berlach)

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, im Mai 2011 ....

(Reinhard Berlach)

Danksagung

Diese Diplomarbeit wurde in den Jahren 2010 und 2011 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Herzlich bedanke ich mich bei Herrn Dipl.Ing. Philipp M. Glatz, der mich der langen Zeit dieser Arbeit betreut hat und immer wieder mit Rat und Tat zur Seite stand. Mein dank giltHerrn Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger, der sich bereit erklärt hat diese Arbeit zu begutachten. Auch bei Herrn Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner und Herrn Dipl.-Ing. Dr.techn. Christian Kreiner bedanke ich mich für die aufmunternden und motivierenden Worte beim einen oder andern Kaffee am Gang vor unserm Labor.

Natürlich gilt auch mein Dank meinen Eltern, ohne deren Hilfe und Unterstützung ich niemals so weit gekommen wäre. Auch meinem großen Bruder Clemens Berlach muß ich danken, denn er hat mich dazu inspiriert nach Graz zu kommen und hier zu studieren.
Ebenso dank ich auch meiner liebsten Doris, die mich die langen Jahre auf der Uni so selbstverständlich begleitet hat. Ohne sie wäre ich wahrscheinlich schon in den ersten Jahren gescheitert.
Zu guter Letzt kommen noch die „*Laborratten*“, diese kleine Gruppe an unermüdlichen Diplomanden in den Laboren des ITI, ohne denen die lange Arbeit an diesem Werk wohl unerträglich geworden wäre (und die Betreiber des Kaffeeautomaten um einiges ärmer).
Danke Georg Macher, danke Ralph Gruber, danke Michael Lackner und danke Claudio Schiesl, ohne euch wären die langen Jahre am ITI trostlos und einsam gewesen.

Graz, im Mai 2011 Reinhard Berlach

Contents

List of Figures

List of Tables

List of Sourcecode

Chapter 1

Introduction

This master's thesis is embedded in a project at the Institute of Technical Informatics at the TU-Graz, which is called tiny operating system plug-in with energy estimation (TOSPIE2) [19]. It implements a multi-agent middleware for wireless sensor networks (WSNs) and provides a platform for running energy-efficient virtual organisations (VOs) in a pervasive environment.

1.1 Motivation

WSNs are networks of many small smart sensors, which communicate in an ad-hoc manner. Each of these nodes is battery powered, so energy is one of the resources, that has to be handled carefully. Newer WSNs are using some energy-harvesting system (EHS) to get energy from the environment.

Today it is normal to build one network for one purpose only. But the rapid evolution of microelectronics allows building more general mote. Motes, those are equipped with an array of different sensors for more then one application. So it would be nice to build one Network for many applications.
Each of this application can contain some atomic tasks. This tasks are called agents. More than one agent can run at one node and the agents can be moved by the console. The console is the server side program of the implemented middleware. This program allows the user to inject agents to the network. The middleware has more than one purpose. First it has to coordinate the resources of the node and acts as an arbiter for the virtual machines (VMs). Second it has also to manage the network. In the middleware layer runs the routing protocol and the transport layer of this network. And third the middleware manages the duty cycle of the network and its time synchronisation.
Different applications must not interfere with each other. So we encapsulate each application in a VO. Agents in one VO can communicate with each other, but agents of different VO can only communicate via the server as a proxy. This structuring of the WSN helps the administrator to manage the network.
Since this rapid progress in microelectronics brings faster and more powerful CPUs, it doesn't bring better batteries. So energy management is much more important than management of the sensor-time or CPU-time. This will be done with this middleware.

1.2 Objective and goals

Within the scope of the project TOSPIE2 and the multi-application middleware (MAMA), which is developed earlier at this institute, I build up a multi agent middleware with a power-management console.
Each Task given by an application is split into simple and atomic functions, which are called agents. All agents for this Task grouped together in one VO.
The power-management console is built to maximise the lifetime of the WSN and distributes the workload efficiently over the WSN. This can happen to the migration of some agents or even a whole VO from one part of the WSN to another.

The evaluation of the middleware is built on some measurements and functions given in the third chapter of this master's thesis.

1.3 Structure

In Chapter 2 I present definitions and models of lifetime in WSNs. Inspired by them I look at some approaches to save energy. Three groups of them are discussed in detail in Section 2.2. At Section 2.3 we explore the power consumption on the mica2 node and are looking at the different components in TinyOS and MAMA in aspects of used energy. With the knowledge how to save energy and where it is wasted in the software, we begin to develop some concepts in Chapter 3. First we look at the possibilities in the operating system to reduce the wasted energy. Then I bring up the concept of agents and virtual organisations. After the agents the power management is introduced. There I present the concept of duty cycling as explained in Section 2.2.3 and mobility. Also the modes of the management are shown. Last section the test cases are shown. The Chapter 4 starts with the agents and how the virtual machine is implemented. Then the implementation of the power management and the console is presented. And like in the concept it ends with the test cases. The last two chapters present the result and the conclusion of this middleware. A short discussion of the module test and the benchmarks are given in Chapter 5. An outlook of future work is the last section of this master's thesis.

Chapter 2

Related work

In this chapter I present a detailed power analysis (Section 2.3) of the mica2 mote (Section 2.3.1), some programs (Section 2.3.2) and MAMA (Section 2.3.3).
In the first part (Section 2.2) of this chapter I discuss some techniques to save energy in WSNs. There I present a taxonomy of energy conservation schemes and discuss some of them in detail.

2.1 Lifetime of WSNs

Since WSNs had to work a long time, some times many years, models, to calculate the lifetime of them, are a critical design criterion. Lifetime and energy consumption are directly related to each other since most nodes are battery powered.

2.1.1 Definition of lifetime

How is the lifetime of a network defined? Dietrich and Dressler [15] discuss some definitions. Here a quick overview:

Based on number of nodes alive: the lifetime is only dependent on the number of nodes:

- $T_n^n = \min_{v \in V} T_v$**:** if the first node fails, the network is dead
- T_n^k**:** if k out of n nodes fail, the network fails.

Based on sensor coverage: the failure of nodes reduces the coverage of the WSN or its redundancy.

- **volume- or area-coverage:** each point in the area or volume has to be covered by the WSN
- **target coverage:** only a finite number of points of interests are located in the covered area/volume.
- **barrier coverage** describes the chance, that a mobile object can pass undetected the WSN.
- **α-coverage:** α% of the region is covered by at least one node.

k-coverage: each point of the area is at least covered by k nodes.

Based on connectivity: this definition is dependent of how connectivity is defined.

Nodes that have connection to the basestation (BS).

Data-packets transmitted to the BS.

successful data gathering trips.

Combined form connectivity and coverage: this method to calculate is a combination of the coverage (mostly α-coverage) and connectivity.

Based on application quality of service (QoS) requirements: this is a kind of definitions that sounds very good for real applications but has no use in models. This comes from the lack of a generally accepted definition of QoS.

Defined by Blough and Santi [8]: based on a minimum of three time points: first time is the lost of connectivity, the second time is, when a percentage of nodes died and the third is lost of coverage for a specified part of the network.

These descriptions cover a large area of WSNs and their applications, but there are some parameters, they don't take in to account. [15] also discuss some of them.

Topology changes is something that is not always considered, when designing a WSN. But there can always be one, for example when a node breaks down, then there is a topology change, where some communication links break and the sensor coverage changes. It is like a node moves. The physical mobility of nodes is discussed in [5] to save energy.

Heterogeneity comes in different types. [15] speaks from eight to ten types. A few of them are discussed like different sensors, varying transmission power, different energy reservoirs and so on. But this is only the heterogeneity of the hardware, but there also are different type of applications like cluster heads or base stations. Each kind of heterogeneity brings a much more complex model to calculate the lifetime of a WSN.

Application characteristics is split into three aspects. First it is the application of the WSN. If this application is split into several tasks, which are distributed over the network. Therefore, different nodes have different tasks to do and so they will use a different amount of energy.
Second aspect is the destination of the data. Is there one sink in the middle of the net or on the edge. Are there more then one destination or a mobile sink? All of this is also considered for the energy consumption of the network.
The last aspect is the node activity in terms of sensor measurement, data processing and communication. this can be triggered by an event, by a timer or by an other node. In all cases there will be different energy-consuming actions.

Quality of service is a criteria, that seldom is taken into account. The classical view of the QoS, such as delay, jitter or resource consumption, doesn't help in case of WSN. All of this parameter are End-to-End, but in WSN mostly other parameters are important, such as collective latency[1], coverage, connectivity or event detection ratio. [15] sees a deep relation between QoS and lifetime of a WSN.

Completeness is to consider more than one aspect of a WSN in the lifetime-model. As an example, coverage and connectivity is often considered independently, whereas these measures are influence each other essentially.

2.1.2 Models and formals to calculate lifetime

[11] gives equation (2.1) to calculate the expectation of the lifetime in a network.

$$\mathbb{E}[\mathcal{L}] = \frac{\varepsilon_0 - \mathbb{E}[E_\omega]}{P_c + \lambda \mathbb{E}[E_r]} \tag{2.1}$$

In Equation (2.1) $\mathbb{E}[\mathcal{L}]$ is the expectation of the lifetime of the network. ε_0 is the total non-rechargeable initial energy. $\mathbb{E}[E_\omega]$ is the expected wasted energy. P_c is the constant continuous power consumption over the whole network. λ is the average sensor reporting rate defined as the number of data collections per time unit. $\mathbb{E}[E_r]$ is the expected reporting energy consumed by all sensors in a randomly chosen data collection.

Looking at the formula we see that to extend the lifetime we can manipulate some coefficients. First of all there is ε_0. When we use bigger batteries the network will live longer.
P_c is the energy the network uses all the time. If we can minimise this power drain, we increase the lifetime. This can be done through algorithm and protocols described in Section 2.2.
$\mathbb{E}[E_\omega]$ is the energy of the network, when it dies. To minimise this parameter a protocol should be used that takes into account that some nodes has more energy than other.
λ is given by the applications running on the WSN. So it can't changed by the middleware, but $\mathbb{E}[E_r]$, the reporting energy drain, can be minimised by the used protocols and algorithm to send data to the sink.

Equation (2.1) can be easily extended to include other sources of energy consumption. Equation (2.2) shows how to include network maintenance.

$$\mathbb{E}[\mathcal{L}] = \frac{\varepsilon_0 - \mathbb{E}[E_\omega]}{P_c + \lambda \mathbb{E}[E_r] + \eta \mathbb{E}[E_m]} \tag{2.2}$$

In Equation (2.2) there is η the rate of maintenance in the WSN and $\mathbb{E}[E_m]$ the expected energy drain in one randomly chosen maintenance cycle.

The lifetime calculated with equation (2.1) or (2.2) is a lifetime based on number of nodes alive. Based on the wasted energy $\mathbb{E}[E_\omega]$ it is defined as T_n^n or T_n^k as described in Section 2.1.1.

Dietrich and Dressler bring at the end of their paper [15] a much more complex formula to calculate the lifetime of a WSN.

[1]The collective latency is defined as the difference between the time at which the first packet, related to this event is generated by the source sensor, and the time at which the last packet, related to this event, or the last packet used to make a decision arrives at the sink.

Before we can go into this formula we need some perquisites. They define R as the region of deployment, but it is not relevant what the concrete specification is.
Then there are the sensors. They define $Y = \{y_1, y_2, ..., y_k\}$ as the set of all available sensors in the network. The set of existing nodes is called S^Y. The sensors available at one sensor i are represented by the subset $Y_i \subset Y$. It's important that each sensor is associated to a subset of the set of sensors. There can be more than one sensor at one node or no sensor. The cardinality of S^Y is number of the deployed Sensors (n) as seen in equation (2.3b).

$$S^Y = \{s_1^{Y_1}, \ldots, s_n^{Y_n}\}, Y_i \subset Y \tag{2.3a}$$

$$|S^Y| = n \tag{2.3b}$$

Let $U(t)$ be the set of nodes alive at a time t. In equation (2.4a) $u_i^{Y_i}$ (shortened as u) is the sensor i, equipped with the sensors Y_i, which energy is not used up. The number of all nodes alive at time t is given in equation (2.4b) with $u(t)$.

$$U(t) = \{u | u \in S^Y \wedge u \textbf{ alive at } t\} \tag{2.4a}$$

$$|U(t)| = u(t) \tag{2.4b}$$

From the set of nodes alive $U(t)$ they take out the active nodes and call them $V(t)$. These are all the nodes that are alive and **not** in sleep state.

$$V(t) = \{v | v \in U(t) \wedge v \textbf{ active at } t\} \tag{2.5a}$$

$$|V(t)| = v(t) \tag{2.5b}$$

$$V(t) \subseteq U(t) \tag{2.5c}$$

We see that $V(t)$ must be a subset of $U(t)$ since a node can only be active, when it has enough energy. The cardinality of $V(t)$ is given in equation (2.5b) with $v(t)$.
The base stations, also the set of nodes, that are information sinks, are defined as $B(t) \subset S^Y$. As we see it is a time-dependent set and it is defined out of the set of all Sensors. So it can be that some base stations are sleeping or dead.
The network can be seen as a communication graph. This graph $G(t) = (V(t), E(t))$ is an undirected graph. This assumes that always a communication in both directions is possible. It is notable that only active nodes are in this graph. The edges of this graph $E(t)$ are the communication-ways in the network. The ability to communicate between two arbitrary nodes (m_i, m_j) is expressed as a path between these two nodes in graph $G(t)$. This ability to communicate between two nodes is called $\kappa(t, m_1, m_n)$[2]. This path has the length of $n - 1$ hops.

$$\kappa(t, m_1, m_2) = \begin{cases} \forall i \in \{1, \ldots, n-1\}\colon m_i \in V(t) \wedge (m_i, m_{i+1}) \in E(t) & m_1 \neq m_n \\ 1 & m_1 = m_n \end{cases} \tag{2.6}$$

κ defines the ability to send a message from node m_i to node m_j at time t.
Now we define to targets to a WSN. One is a set of target points P^Y, where each point can

[2]without the loss of generality it is possible to renumber the nodes (m_i, m_j) to (m_1, m_n)

be sensed by one or more Sensors. The formal description of P^Y is found in equation (2.7).

$$P^Y = \{p_1^{Y_1}, \ldots, p_m^{Y_m} | p_i^{Y_i} \in R \wedge Y_i \subset Y \wedge Y_i \neq \emptyset\} \tag{2.7}$$

$$A^y(t) = \bigcup_{\forall v \in V(t)} A_v^y \cap R \ , \ y \in Y \tag{2.8}$$

Equation (2.8) defines the area covered by all sensors of one type y. A_v^y denotes the area covered by sensor y from node v.

Now they introduce two criteria (c_{**}, cl_{**}). Both of them are defined in $[0,1]$. c_{**} is defined as a soft lower bound of full functional network. cl_{**} is the hard lower bound, below that a network is considered as non-functional $(0 \leq cl_{**} \leq c_{**} \leq 1)$. A detailed list of criteria for a WSN is found in [15] at page 18 Table II. For each of this criterion they define two functions (ψ_{**}, ξ_{**}). ψ_{**} indicates how well a criterion is fulfilled and is resulting in a value in the range $[0,1]$. ξ_{**} is a measure of the quality of the fulfilment depending on the upper and lower bound. A value of $\xi_{**} > 1$ means that all criteria are fully fulfilled, $\xi_{**} < 0$ signals that none criteria is fulfilled and in range $[0,1]$ indicates how good the criteria is fulfilled. In equation (2.9) is ξ_{**} defined.

$$\xi_{**} = \begin{cases} \frac{\psi_{**} - cl_{**}}{c_{**} - cl_{**}} & \textbf{if } c_{**} \neq 0 \wedge c_{**} \neq cl_{**} \\ \frac{\psi_{**}}{c_{**}} & \textbf{if } c_{**} \neq 0 \wedge c_{**} = cl_{**} \\ 1 & \textbf{if } c_{**} = 0 \end{cases} \tag{2.9}$$

All the defined criteria from [15] such as *number of alive Nodes*, *Latency* or *area coverage*, defines the function ψ_{**} trough the criteria (c_{**}, cl_{**}).

To define a lifetime from here on we take the minimum of all $\xi_{**}(t)$ and call it simply $\xi(t)$. As defined in equation (2.10) this is the measure how well the network is doing.

$$\xi(t) = \min(\xi_{**}(t)) \tag{2.10}$$

With $\xi(t)$ we can define an order sequence of points in time called T. At each of this points $\xi(t)$ changes from a value below 1 to one over 1 or vice versa. The formal description we find in equation (2.11). From the availability criterion we get the maximal disruption time (Δt_{sd}) and with this time we can define a ratio ψ_{sd} (2.13) between the maximum time of accrued service-disruption sd_{max} and Δt_{sd} .

$$\begin{aligned} T = \{t_i | (\xi(t - \epsilon) \geq 1 \wedge \xi(t) < 1) \vee (\xi(t - \epsilon) < 1 \wedge \xi(t) \geq 1)\} \\ t_i < t_{i+1} \ , \ i \in \mathbb{N}_0 \end{aligned} \tag{2.11}$$

$$sd_{max} = \max((t_{i+1} - t_1) \colon \xi(t) < 1 \ , \ i \in [0, |T| - 1]) \tag{2.12}$$

$$\psi_{sd} = \begin{cases} \frac{\Delta t_{sd}}{sd_{max}} & sd_{max} > 0 \\ 1 & sd_{max} = 0 \end{cases} \tag{2.13}$$

In Equation (2.14) an index is defined of the first point in time, when the disruption of the services is longer than Δt_{sd}. With this index they define all the durations t_i^a when the network is fully operational.

$$e = \begin{cases} \min(i \in [0, |T|-1] \colon \xi(t) < 1) \ , \\ \qquad\qquad (t_{i+1} - t_1) > \Delta t_{sd} & \text{if such } i \text{ exists} \\ |T| & \text{otherwise} \end{cases} \tag{2.14}$$

$$\forall i \in [0, e] \colon t_i^a = \begin{cases} t_{i+1} - t_i & \text{if } \xi(t_i) \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

Now they [15] define the lifetime of a network. Z_a is the time the network is fully operational. Z_e is the time, until the network first lost its services for more then (Δt_{sd}).

$$Z_a = \sum_{i=0}^{e} t_i^a \tag{2.16}$$

$$Z_e = t_e \tag{2.17}$$

Dietrich and Dressler not only define the criteria fully fulfilled but they define a range with these criteria where the network is considered operational but not full. This range is defined when $\xi(t)$ is between 0 and 1. When setting the threshold for $\xi(t)$ in equation (2.11) to (2.17) then we get $T \rightarrow T'$. T' is the sequence of points in time where the network dies $(\xi(t) < 0)$ or comes back to work$(\xi(t) > 0)$.
So we also can do with sd_{max}, ψ_{sd}, e and t_i^a. With all of this new variables we can calculate the time Z'_a and Z'_e. Z'_a is then the duration when the network is fully or partial operational, and Z'_e is the time, when the network dies for the first time longer than (Δt_{sd}).

This kind of view of a WSN can also used to give an administrator a measure how well the WSN do at this time. It is not really usable in the design phase but you can use it in the implementation of the WSN.
It is also possible to consider an EHS in the network.

2.2 Energy conservation

This section brings some schemes of energy conservation, which are used to expend the lifetime of a WSN. [5] defines three groups of power-conservation schemes.

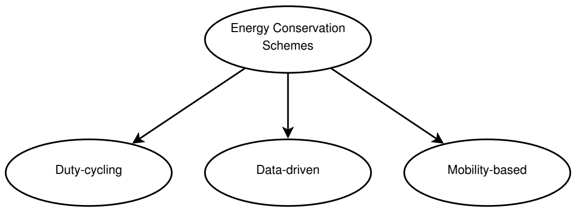


Figure 2.1: Overview of power-conservation schemes [5]

The complete taxonomy from [5] is found at the appendix C on page 94.
As we will see in Section 2.3, to send data over the wireless network will cost most of the energy. So duty cycling is used to determine when and which node is used and turns on its radio. Data-driven schemes try to minimise the sent data or the read data and mobility-based approaches try to minimise the sent messages by introduction mobility.

2.2.1 Data driven techniques

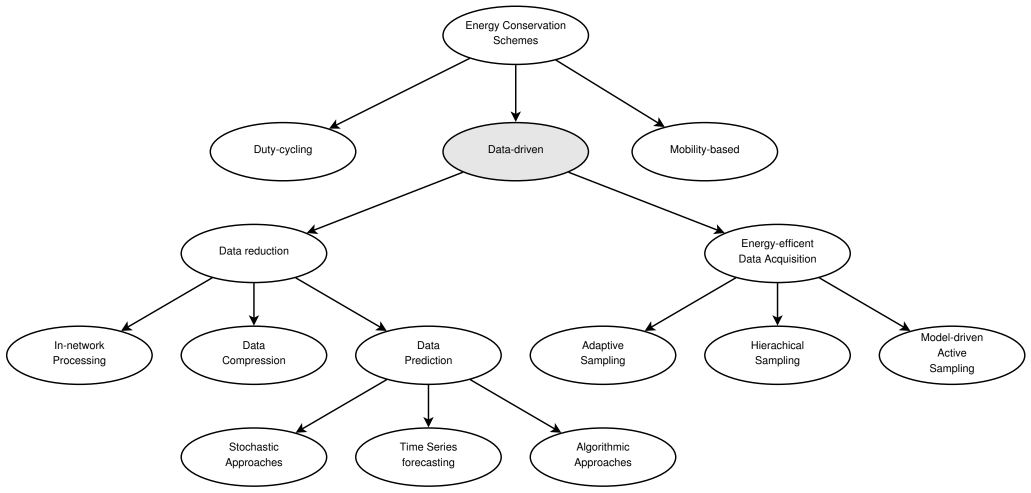


Figure 2.2: Data driven [5]

Data-driven approaches try to minimise the sent data to save energy or to minimise the energy spent to get the data. This can be done through data reduction or energy-efficient acquisition.

Data reduction
This power-conservation technique tries to minimise the data sent over the network. This can be done trough the following methods.

In-network processing uses the ability of the nodes in a WSN to compute the data before sending them over the network. So the amount of data sent over the radio can be reduced.

Data compression tries to compress a series of collected data, from this node or other ones, and send the reduced package of data to the sink. There the data will be restored.

Data prediction: This approach uses signal processing models to predict the data. One model is located at each sensor node and one at the sink (here can be as much models as sensors exists in the network). So can the sink answer queries without sending any data over the network. The model at the sensor is used to ensure the correctness of the model at the sink.

Energy-efficient data acquisition
This technique is used when the energy consumption of the sensor is not negligibly small. This can come from power-hungry transducers or ADCs or even an active sensor like laser ranging. Also a large acquisition time can increase the energy consumed by the sensor.
Therefore, methods to reduce the sampling of the sensor are used.

Adaptive sampling uses similarities of measured samples to reduce the amount of data. This correlation of data can be temporal or spatial.

Hierarchical sampling assumes that nodes are equipped with different sensors. These technique selects dynamical the class of sensor to get a trade-off between energy consumption and accuracy.

Model-driven active sampling: similar to *data prediction*, this approach uses a model to forecast the values with certain accuracy. This model is used to reduce the number of samples and therefore also the transmitted data.

2.2.2 Mobility based techniques

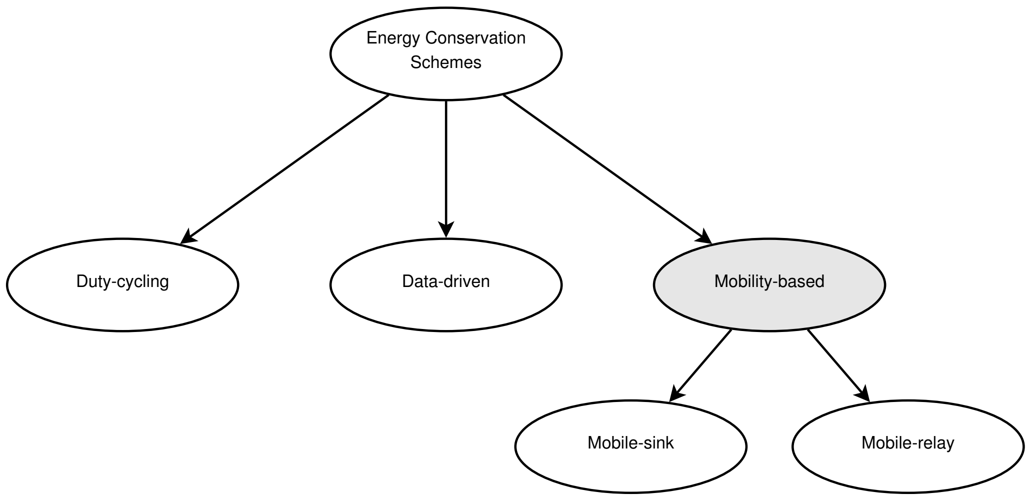


Figure 2.3: Mobility based [5]

Here [5] introduce a new scheme to conserve energy. They use mobility to conserve energy. This is strongly connected to heterogeneity of the network. So they use few special nodes, which have a mobiliser-system onboard.

Mobile sink

This approach uses Data-sinks, which moves around in the network and collects data from the sensor nodes. This can be done direct (single hop) or over a multi-hop path.

Mobile relay

This system uses mobile nodes to function as a mobile relay in the network. These nodes are moving in an unpredictable manner. They collect data from sensor nodes on their route and send these data to a sink, when it comes in range.

2.2.3 Duty cycling schemes

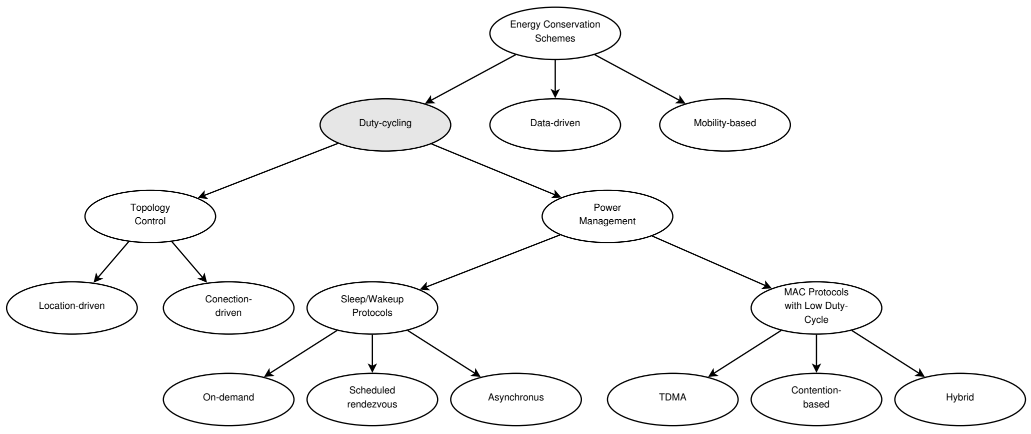


Figure 2.4: Duty cycling [5]

In duty cycling they differs a network wide approach (called topology control) or a single node (called power management).

Topology control
This kind of power-conservation scheme uses the high density of a WSN. Protocols of this kind turn on only the needed nodes, and save so the energy in the network. Therefore, you buy lifetime of the network through more nodes in use.

Location driven approaches use topological information to get the density of the network.

Connection driven approaches use information from the connectivity of the network. This information is used to lay down nodes or to wake them up.

Power management
Power management uses the μC own power-saving capabilities to conserve energy. This technique is used on one node with respect to the network capability. This means that none of these nodes can be excluded from collecting and sending information of the environment. TinyOS [25] uses techniques like this to save energy. As soon as nothing is to do for the mote it puts it μC to the lowest possible power state.

Sleep/wake-up protocols defined for a given component. Normally it will be the radio (or network) sub-module. [5] classifies these protocols in three groups.

- On-demand
- Scheduled rendezvous
- Asynchronous

Medium-access control protocols are used by the radio sub-module to access the radio channel. TinyOS [25] uses a protocol calls B-MAC [28]. X-MAC [9], T-MAC [30] and S-MAC [32] are other MAC protocols. These protocols are sorted in three categories

TDMA: this protocols divide time in periodic frames and each frame a defined number of slots. Each node in the network is assigned to one slot for each frame according to the scheduling algorithm.

Contention based: this protocols let the nodes *discuss* their access to the radio channel. This can happen trough different algorithm.

Hybrid: here the node can switch between time-division-multiple access (TDMA) and contention-based access to the radio channel.

Most of the medium-access control (MAC) protocols used in WSNs are contend based, like B-MAC [28], X-MAC [9], T-MAC [30] and S-MAC [32]. A detailed description of the low-power listening (LPL) from B-MAC is brought in Section 2.3.2.

2.2.4 Discussion of lifetime and power conservation

Before we start with the power analysis, I want to discuss the presented energy-conservation schemes in aspect of the two models of lifetime.
First we look at Equation (2.1). There we have the four parameter $\mathbb{E}[E_\omega]$,P_c,λ and $\mathbb{E}[E_r]$, that can be by the software.
The wasted energy $\mathbb{E}[E_\omega]$ can be reduced with the topology-control technique. This comes from the equally distributed power drain of the nodes. Nodes with lower energy stay sleeping while their neighbours with more energy have to do their work. So the not used energy is very low at the time the network dies.
Power management tries to minimise the continuous power drain P_c. This is achieved by bringing the μC into the low-power sleep state each time nothing is to do, or by the using an energy efficient MAC protocol, that reduces the power consumption of the radio module.
The reporting Energy $\mathbb{E}[E_r]$ can be reduced by data-driven or mobility-based schemes. The data-driven approach reduces the amount of data needed to be sent. Mobile sink or mobile relay reduces the hops to the sink, and therefore the used energy.
λ, the frequency of data gathering, can be changes with the energy-efficient data acquisition.

Looking at the model from Dietrich and Dressler [15] the analysis of used parameters and their connection to the power consumption is a little bit complicated. They never look direct at the power consumption of a mote.
But we can look at the prerequisites of their model. The topology control reduces the number of nodes active without breaking the connectivity to the sink or reducing the covered area. So we can define an optimal topology control as the minimal $V(t)$ so that $\forall v \in V(t), \forall b \in B(t) : \kappa(t, v, b) \wedge \bigcup A_v^y \geq \mathbb{A}$, where $\mathbb{A}$ is the wanted covered area.
For the other schemes are the mathematical expressions not so easy to find. So is a mobile sink defined as working, when the connectivity κ from the sink to each active node guaranteed in a defined time interval. So we can define for each technique, presented in this section, prerequisites or criteria for this model.

2.3 Power analysis

The issue of this section is the analysis of the power-consumption. This happens in three parts.
The first part is the analysis of a Mote; in this case it is the mica2 from crossbow. The second part is concerned with some Programs, implemented in nesC and TinyOS, and the third with MAMA.
The power-analysis is done with the TOSPIE2 measurement system [19].

2.3.1 Power drain of the mote and its components

In Figure 2.5 we see a model of one node. The standard components of a node are a processing unit, some sensors and a transceiver. A power unit is also needed to power up all the systems.
Additional to these units there can be a power generator to charge the power unit, a localizer to get the position of the node or a mobiliser to move the node through the environment.

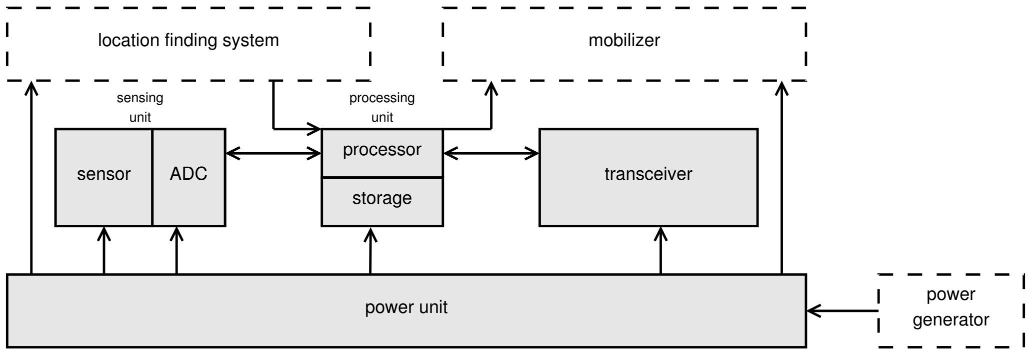


Figure 2.5: Model of a wireless sensor Node [3]

The mica2 mote consists of an ATmega 128 microprocessor from ATMEL as the processing unit and a CC1000 radio chip from Chipcon as the transceiver. This mote has also a 32kB flash-storage for Data. The μC runs at a speed of 8MHz. As sensor system we used the MTS310 board from crossbow with a two axis magnetometer, a two axis accelerometer, a temperature sensor and a light sensor. It also has a microphone and a 4 kHz buzzer. All tests are running with TinyOS as operating system on the mote.

This chapter also verifies the assumption that sending is much more expensive then computing some data.

The ATmega 128 μC

The ATMEL ATmega 128 μC is a low-power CMOS 8bit microcontroller [6]. It is built to be used in low-power systems like WSNs.
Energy conservation comes through the 7 power-save states of the MCU. The power consumption of the different Power-states of an ATmega 128 is seen in Table 2.1.
In TinyOS [25] the module `McuSleepC` provides a mechanism to put the μC into sleep. This module looks which systems are used and puts the μC into the best power-saving

Power state	**Power consumption** from Data sheet[6]
Active	$24mW$
Idle	$11mW$
Standby	$1.8mW$
Power Down	$0.03mW$

Table 2.1: Power consumption of the ATMege128 μC for different power states

mode.
The ATmega 128 has many additional systems such as 4 Timers (2 8-bit and 2 16-bit), a SPI, 2 USART and an 8-channel 10-bit ADC. Not all peripherals running in each power-state. So has the `McuSleepC` to look which one is running and calculates the right state.

Power state	**Radio**(SPI[3])	**Timer**(Milli)	**Serial**(USART)	**ADC**	**Interrupts**[4]
Idle	x	x	x	x	x
ADC NR		x		x	x
ext. Standby		x			x
Standby					x
Power save		x			x
Power down					x

Table 2.2: Power states and components

Active: the μC is working and all peripherals are running (if activated). The MCU accepts all interrupts.

Idle: the μC is inactive but all peripherals are running. The MCU accepts all interrupts and changes into state *Active*.

ADC noise reduce: in this mode the μC can be waked up by Timer0, external interrupts, two-wire interface (TWI, I2C), ADC and the watchdog. It is used to get AD-Conversions with better resolution and lower noise.

Power down: this mode halts all generated clocks, also an external oscillator is stopped. The μC is running in an asynchronous mode. Accepted interrupts are external interrupts and interrupts from the watchdog and the TWI. If getting an external interrupt, the level needs to be hold for at least 2 ms to make sure the μC has changed to state *Active*.

Power save: this mode and power-down mode is the same, except for the external clock source of the Timer0. This timer will keep running and can trigger an interrupt to wake up the μC. The μC will also need 2 ms to change to *active*.

[3]On the mica2 the SPI is only used by the radio chip.
[4]INT4:0 are used on mica2 for external interrupts

Standby: this mode is used, when the system is set into the asynchronous mode but needs to wake up fast. A wake up from standby only takes 6 cycle, because the external oscillator is running but can not trigger any interrupt. Every thing else is like the power down mode.

extended Standby is used instead of the standby mode when using an external oscillator for the timer. Here the timer can trigger interrupts.

Radio module

This section is about the radio-communication module. In the first part we look at the radio chip itself. The second part is about the software implementation.

Chipcon CC1000

> The CC1000 is a true single-chip UHF transceiver for low-power and low-voltage applications. The main parameters are programmable via a serial bus.[12]

Through this programmable control it is possible to turn on or off unused components in the CC1000 chip. An overview over some power states is seen in Table 2.3.

State	Current	Power[5]
Power down	$0.2 - 1\mu A$	$0.6 - 3\mu W$
Crystal oscillator	$30 - 105\mu A$	$90 - 315\mu W$
Bias generator	$860\mu A$	$2.4mW$
Rx	$11.8mA$	$35.4mW$
Tx	$8.6 - 25.4mA$	$25.8 - 76.2mW$

Table 2.3: Power states of the CC1000 chip [12]

TinyOS uses an active message (AM) system [10] and the B-Mac protocol [28] as media access control. When turning on this component the CC1000 chip goes from power down to Rx-mode. When stopping it goes to power down mode.

The mica2-mote with TinyOS uses the CC1000 at $19.2kbps$ with a Manchester encoding. This gives a sensitivity of $-99dBm$ at a $BER = 10^{-3}$.

SPI

For the communication with the CC1000-chip the mica2 uses the SPI.
Including the AM Component in the program for the mote automatically starts the SPI at boot-time, even when the AM is not started. This behaviour prevents the μC to go an other sleep state as `IDLE` (see Table 2.2).

[5]this power is calculated under the assumption that we use 3 V as voltage

The CC1000 chip is used on the mica2 as the SPI-master node and the atm128-μC as the slave for data transmission. To transmit commands the μC is the master and the CC1000 is the slave, but for this is an other bus used.

Sensor board

In the configuration we use, a mica2 node and the MTS300 sensor board, all sensor data are read via the ADC on the atm128-μC. This μC has 8 ADC channels $(0 \ldots 7)$. Channel 7 is used to monitor the voltage of the battery and channel 0 is used to read the RSSI value from the CC1000-chip.
The 52-pin expansion board on the mica2 brings all 8 channels of the ADC out. Channel 2 is used by the microphone, if you want to sample the audio data direct. You also can activate the tone detector, which gives a level-interrupt at `INT3` when detecting a 4 kHz tone.
The sounder is a simple 4 kHz piezoelectric resonator. You can control this buzzer with the output of `PW2`.
Photo and Temperature is both converted by ADC-channel 1. The power of the photodiode is switched with `INT1` and for the thermistor with `INT0`.
The two-axis accelerometer is connected to the ADC-channel 3 and 4 with the μC. The power is controlled by `PW4`. This accelerometer can be use to implement a movement detection or seismic measurement. To enhance the accuracy of the measurement you had to do some simple calibration.
The two-axis magnetometer is used with an instrumentation amplifier for each channel. These amplifiers are sampled by ADC-channel 5 and 6. The power control is line `PW5`. To make this magnetometer useful in some applications you will need an external set/reset circuit to recover from saturation problems.
For more information of the mts300-board and the used sensor-ICs you can look at [14].

2.3.2 Power consumption of TinyOS modules

In this section we will discuss the power consumption of some modules of TinyOS, the timing of them. As some examples for measure measurements is the powerconsumption of building a message (setting all parameter to desired values) and send it into the ether. We also want to know how much energy we need to get one value of each sensor (see Section 2.3.1 and Figure 2.9).

Radio communication in TinyOS

In this section I will discuss the power consumption of the radio communication in TinyOS. In short words I will present the MAC layer used in TinyOS and its duty cycling. We also have a look at the AM implementation.

B-MAC [28] is the MAC protocol implemented in TinyOS. It brings a flexible interface to obtain an ultra low power operation, an effective collision avoidance and a high channel utilisation. In [28] they present an adaptive duty cycling which is not implemented in TinyOS.

Low-power listening is duty cycling in the B-MAC Protocol [28].

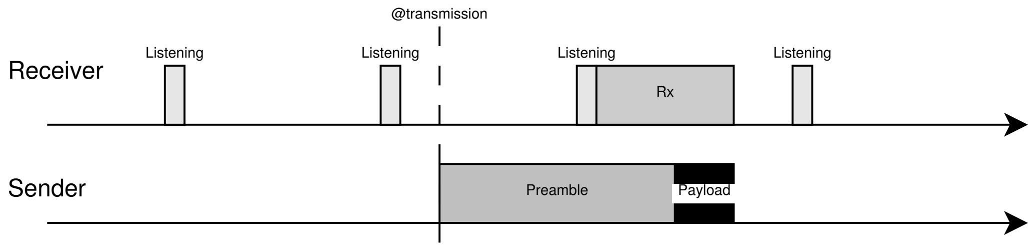


Figure 2.6: Asynchronous low-power listening

Figure 2.6 shows this asynchronous LPL. The duty cycle in the figure is 1 : 10. This means, that the radio listens one time slot and sleeps 10 slots. That the receiver recognises a message, the preamble of this message must be at least as long as the sleeping time; here its 20% longer, also 12 slots.
If a receiver listens to such a preamble, it stays active and waits to the end of the preamble, and receives the payload.
For the sender this means that it must bring up all the energy to send the long preamble and the payload. Figure 2.7 shows this problem.

As we can see in this figure the energy cost of sending the second message is the same

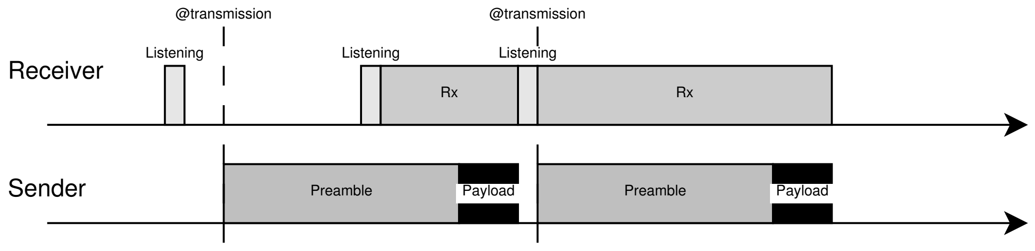


Figure 2.7: Asynchronous low-power listening with two messages sent

as for the first.
The receiver stays in listening mode for a short time, and is listening for a following message.

Active message

> Active message (AM) is a simple, extensible paradigm for message-based communication widely used in parallel and distributed computing systems. [10]

AM is a small and lightweight communication framework that allows a flexible usage of the network resources. Each message contains the name of the user-handler to be invoked on the target. In TinyOS this name is an 8 bit ID (`AM ID`). This allows the user to implement up to 256 different handlers for his messages. TEP 4 [4] form the TinyOS-alliance defines the range from 128 to 255 as the unreserved pool. TEP 116 [4] suggests that each `AM ID` has its own payload-format.

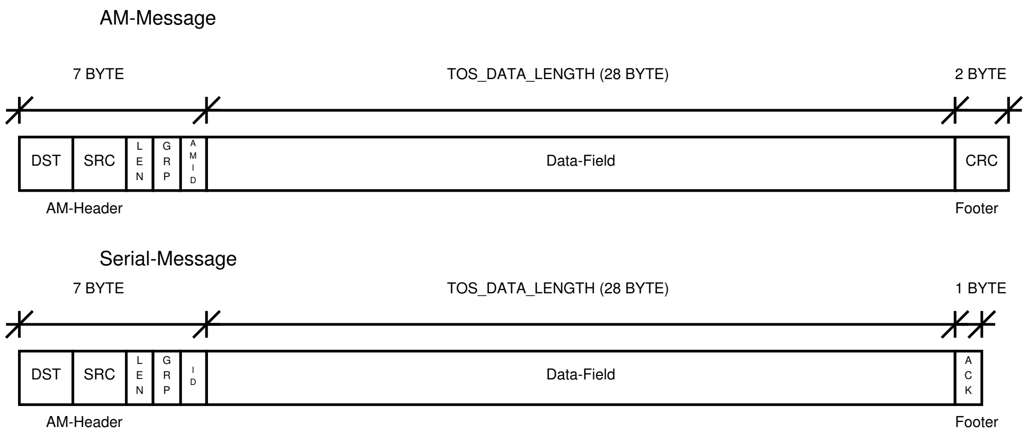


Figure 2.8: Structure of the AM-messages

AM is built on top of the B-MAC-protocol. The AM-modules of TinyOS present a good hardware independent interface to the wireless interfaces of the hardware. How messages are used in TinyOS gives TEP 111 [4]. The AM-System uses AM-messages in their interfaces. In Figure 2.8 we see the structure of the AM-message. Here we see the structure of the 7 byte header. This one is used for a single hop, like the Mac-address in the Ethernet with TCP/IP network. We see when using AM, we get a 9 byte overhead for a maximum of 28 byte of data. This is about 25% overhead on the sent data.

Sensor system

TinyOS uses a 3-Layer Abstraction. This is very useful when using the sensors. Every sensor is accessed by the same interface, called `Read`. This interface is spilt-phase. When reading a Data from a sensor you first call the command `Read.read()` and when the ADC has read the value it signals the event `Read.readDone()`. In Figure 2.9 we see that, from sending the split-phase command to read a sensor until the value is read (event `readDone`), it takes about 9 ms. The actual conversion is only 260 μs at the end (between 6164 and 6166). During this conversion the μC is in the `ADC_NR` state. The long time in between is the time to settle the analog stimulus on the analog-digital converter (ADC).
Figure 2.9 shows short before 6156 the power-up of the analog photo relay. The active μC takes about 25 mW. The high peak (up to $20 mW$) comes from the capacity of the circuit of the photodiode (see [14] page 11, Figure 3-4(b)).
This data are only valid for the photo sensor or the temperature sensor. The magnetic or acceleration sensor takes other times.
In the datasheet of the sensor board [14] we find that it will take 16.5 ms to warm-up in the configuration used on the MTS300-sensor board. TinyOS set it self to sleep for 17 ms.

2.3.3 Middleware

This section shows energy-consumption and timing in MAMA [7]. We look at the timing overhead of the multi-threading system used in MAMA. We also will analyse the overhead of the multi-hop header introduced by the middleware.

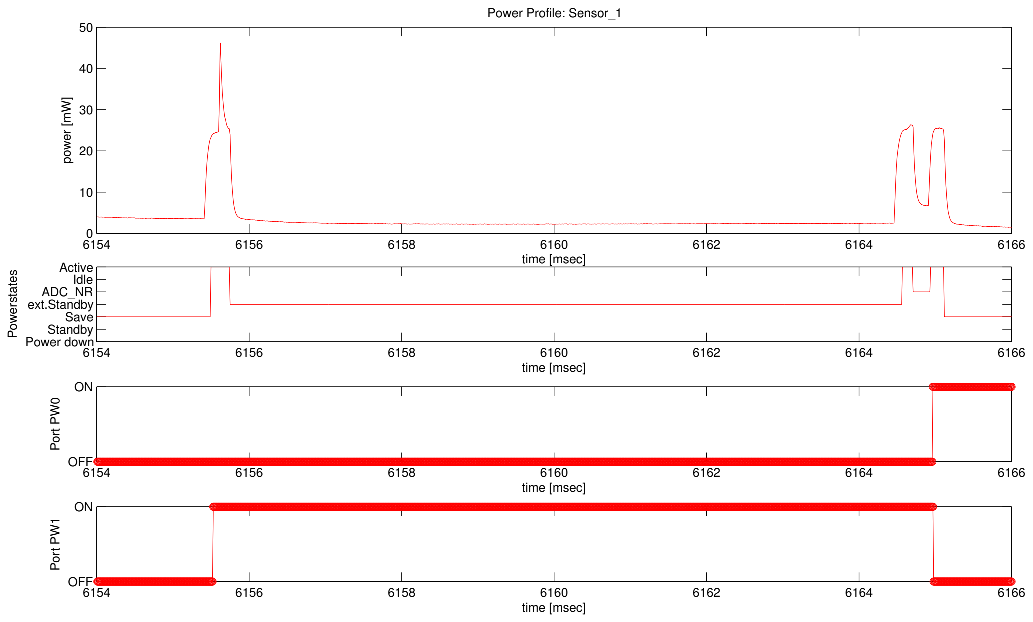


Figure 2.9: read - readDone for photo sensor on the MTS300 sensor board with a mica2

Networking in MAMA

MAMA uses on top of the AM-System its own message-system to build up a multi-hop network. The routing algorithm is implemented in the Router-Module with its own Routing-interface. The utilisation of the network is analysed in [17].

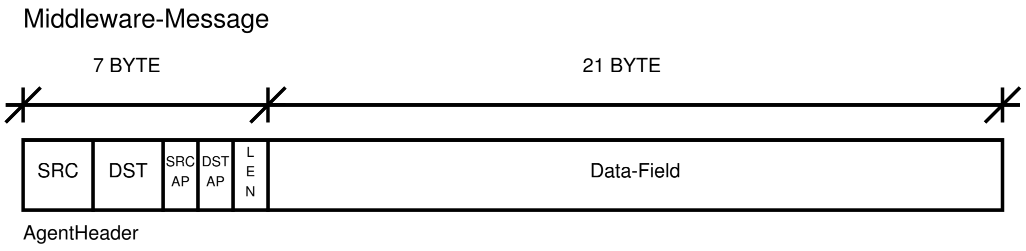


Figure 2.10: Structure of the MAMA messages

In Figure 2.10 we that MAMA use a 7 byte header and has 21 byte maximum for the Data. This gives 25% overhead on the sent data. Over all seen we have 7 byte AM-Header, 2 byte AM-Footer (CRC) and 7 byte MAMA-header. For 21 byte data we get 16 byte overhead. This gives 43% overhead for a message.

Network-Modules

These modules are the connection between the applications and the AM-System.
Here are FIFO-queues used to store the in- and outgoing messages. This makes it possible to implement a transparent duty cycling on the network.

When an application sends a message, it triggers a system call, which puts the message on the sending queue. Here the system call returns.

The sending task takes the first message from the queue. It extracts the header from the message and looks at the destination. If it is for the node, it is put on the receiving queue and the receiving task is triggered.
Is it not for this node, the Router is asked about the next hop to the destination. Then the message is sent via the AM-system.

Sensors in MAMA

Like sending a message MAMA uses system calls to implement the reading of a sensor.
An application wants to read a sensor, so it triggers the system call. For all sensors there is only one systemcall. from the call it will switched to the kernel space and here the split-phase interface is called. When the `readDone` event is triggered the system call comes back and the application can use the read value.

2.4 Virtual organisations

In grid computing, a virtual organisation (VO) refers to a dynamic set of individual and/or institutions defined around a set of resource-sharing rules and conditions. All these virtual organisations share some commonality among them, including common concerns and requirements, but may vary in size, scope, duration, sociology, and structure.

The collaborations involved in Grid computing lead to the emergence of multiple organisations that function as one unit through the use of their shared competencies and resources for the purpose of one or more identified goals [16]. A Grid is defined by [1] through three points:

- coordinates resources that are not subject to centralised control
- using standard, open, general-purpose protocols and interfaces
- deliver nontrivial qualities of service

Is now a WSN a grid? Yes it is. But let's check all the points.
coordinates resources that are not subject to centralised control: Each node coordinates its resources by it self.
using standard, open, general-purpose protocols and interfaces: Most WSNs are using one oft the available , mostly open-source, operating systems.
deliver nontrivial qualities of service: here it comes.

Since a WSN can be seen as a Grid, there can also be some VOs. What rules are significant for a VO in a WSN? VOs are different in size, scope, duration. This means that each VO can include different amount of nodes and/or sensors. They use them in different scope and for/in different time. All members of a VO use the same resources, and share them in a specific way (resource-sharing rules and conditions).

2.4.1 Virtual organisations in MAMA

A VO in MAMA is the set of nodes on that runs specific applications (resource-sharing rules and conditions). An application can consist of one or more set of rules and conditions. These sets we call agents. So a VO consists of one or more agents.
Each of these agents can work on one or more Nodes. Also it is possible that more than one VO and/or agent can run on one Node.

2.5 Summary

At the beginning of this chapter we discuss some definitions and models of lifetime for the WSN. Two models are presented and discussed in detail in the first section. Section 2.2 shows three techniques to conserve energy in a network. Data driven techniques are used to minimise the sent data or to minimise the energy used to gather the data. This is mostly application specific methods so they can't be implemented in the middleware. Mobility based techniques are using motes that can move around in the network. Since moving a node is not something we want to be happening in our network, this approach to save energy isn't available for this middleware.
So only the techniques called duty cycling are to be used.
Comparing them with the models from Section 2.1.2 we see that this approaches are very useful to implement into a middleware. Since we want to reduce the used hardware, and deploy more than one task to each node, we can't use topological controlled services. Therefore, we want further discussions restricted to power management techniques.

In Section 2.3 we see that the components on a mica2 node are build for low power usage and that the operating system TinyOS supports all that features, like low power sleep states, energy efficient MAC-Layer with LPL and low power sensors. The combination of scheduled rendezvous sleep/wake-up protocols and low power contend based MAC-protocols are a worthy approach for this energy aware middleware.
The logical management of nodes through different VOs makes it easier for the user to following theirs agents.

Chapter 3

Design of the middleware and the console

In this chapter we present the concept to move from a multi-application middleware [7] to a Multi-agent middleware with a power-aware scheduler.

In Figure 3.1 we see the overall concept of the Multi-agent middleware. We see the

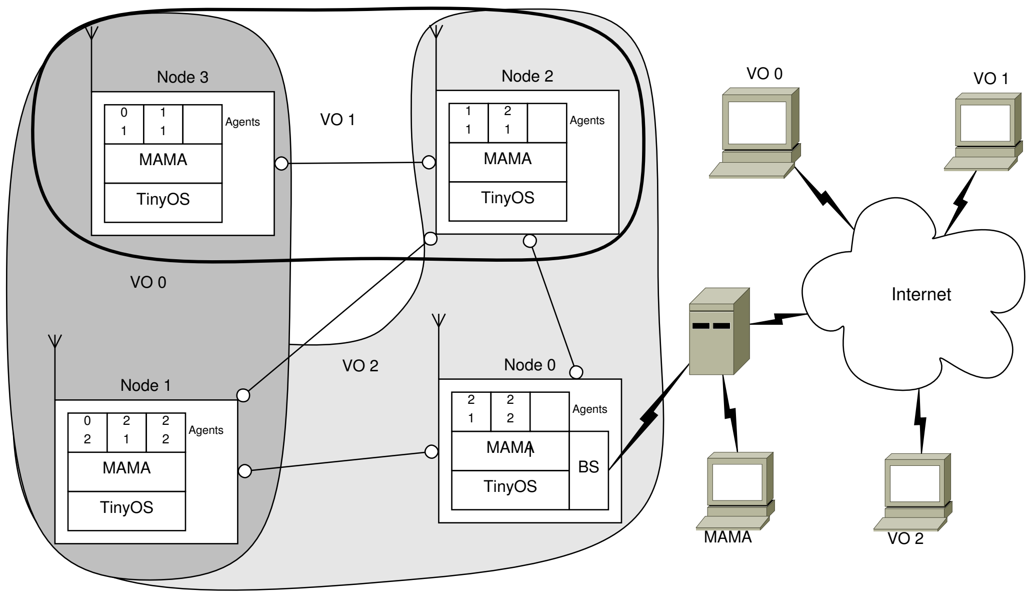


Figure 3.1: Overview of the concept of the middleware

nodes programmed with TinyOS [25] and the middleware. Each node has some slots for agents. Each agent is part of one VO. Node 0 is the BS for this network. It connects the WSN with a server. On this server a controller for each VO has to be implemented. An API for these controllers is provided by the middleware.

In the first part we bring some upgrades to the TinyOS operation system. The second part tells you about algorithm for duty-cycling (see Chapter 2.2.3). Part three tells us how we bring the concept of VOs into the WSN and the Middleware. It also is about the agents and their mobility. The forth section of this chapter brings concepts of test cases and some measures to evaluate the work.

The deep investigation of the power management of TinyOS and its components used on a mica2-node we found some parts of the operating system that we want to change. Some of these things are the SPI system and the LPL implementation of B-MAC [28].

From the 3 main components of energy conservation brought by [5] we choose duty-cycling as the one to implement.
Mobility as discussed in Section 2.2.2 isn't part of MAMA. It also needs a special kind of applications.
As seen in Section 2.3.1 the sensors do not need much energy to sample one Data. So the Data-driven approach (see Section 2.2.1) is not the effective component.

To evaluate the network or better to measure how good it works we take some of the calculations preset in Chapter 2.1.2.

3.1 The mote and TinyOS

In this section we present the changes in the operating system TinyOS and some concepts to save power with the hardware on a mica2 mote.

3.1.1 The mote

The measurement of the power states is seen in Table 3.1. The power from the data sheet for `Standby` or `Power Down` is also valid for the `ext.Standby` or `Power Save`.

Power state	**Power consumption** from Data sheet[6]	**Power consumption** (measurement)	**error** of measurement
Active	$24mW$	$25.4mW$	$\pm0.52mW(2.05\%)$
Idle	$11mW$	$12.1mW$	$\pm0.33mW(2.75\%)$
ADC Noise Reduce		$3.95mW$	$\pm0.17mW(4.18\%)$
extended Standby	$1.8mW$	$0.57mW$	$\pm0.10mW(16.74\%)$
Standby		$0.56mW$	$\pm0.10mW(16.98\%)$
Power Save	$0.03mW$	$0.14mW$	$\pm0.09mW(62.21\%)$
Power Down		$0.14mW$	$\pm0.09mW(62.97\%)$

Table 3.1: Power consumption of the mica2 for different power states of the μC

The millisecond timer of TinyOS uses the Timer0 with the external oscillator. If this component is used the μC can only put to power-save state if the next event is some milliseconds in the future, else the μC goes to the extended standby.

3.1.2 Communicating with the radio-chip

The serial-peripheral interface (SPI) is reserved on the mica2 for the radio chip (CC1000). Since the activation of the SPI prevents the μC to sleep (other state as `IDLE` see in Table 2.2), we had to change the SPI-components in TinyOS to turn of the interface if it is not used.

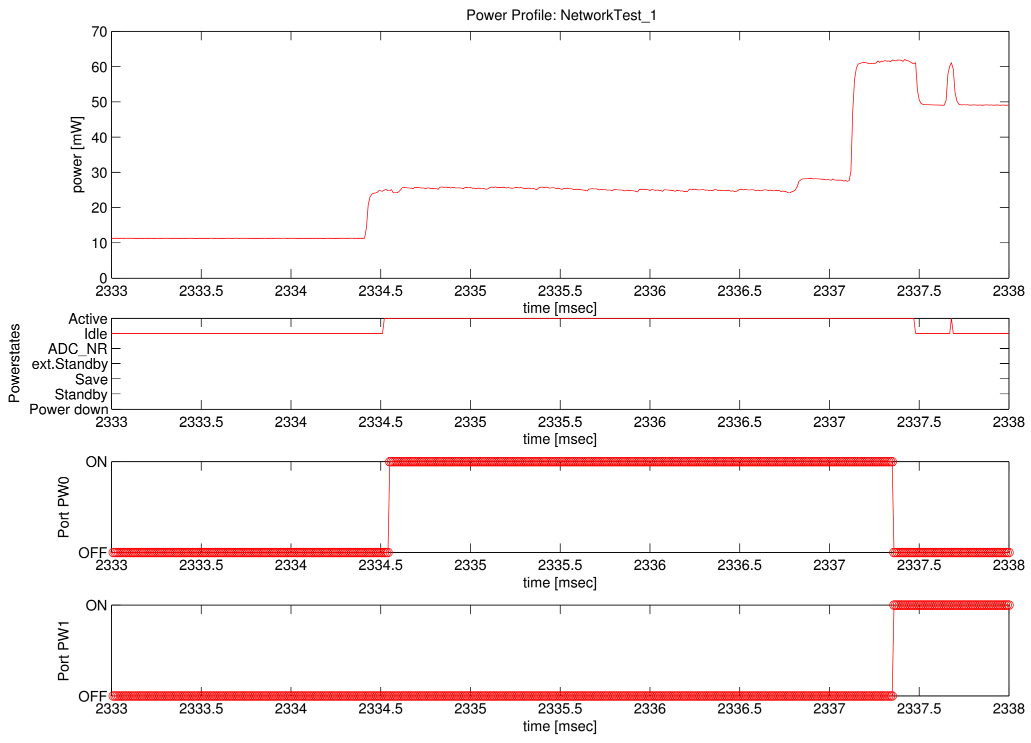


Figure 3.2: Power-up the AM-component

Powering up the radio chip and the AM-Component takes about 3 ms. [12] defines a start up time from power down to Rx in about 2.5 ms, which correlates with the measured power up time in Figure 3.2.

At the first start of the radio module it takes about 4 seconds to adjust the noise-level readings.
As seen in Figure 3.2 the μC is only in `IDLE` while waiting to activate the AM-Component. At the end of the chart in Figure 3.2 there is a power peak and the μC is a short time active. This comes from the pulse-check in the CSMA module of the B-MAC implementation from TinyOS. This peak repeats every 400 μs. This is the minimum time for the preamble in the B-Mac implementation in TinyOS.

3.1.3 Low power listening

At the first start after initializing the active message Module, the radio-chip stays 4s online to level up the noise-floor measurement, as seen in Figure 3.3. In this time the LPL can't send the radio-chip to sleep mode. The time a receiver stands active is 1.6ms and sleeps for 256ms in Figure 3.3. This is a duty-cycle of 1.6 : 256 = 1 : 160.

$$preamlelength[byte] = 2.4 * sleeptime[ms] + 22 \tag{3.1}$$

Equation (3.1) shows how the length of the preamble is calculated in TinyOS. The 22 bytes added at the end is for the CC1000 chip to get a threshold to identify the `HIGH` and `LOW`. The 2.4 is the bytes per millisecond inclusive about 20% to make sure a receiver gets it.

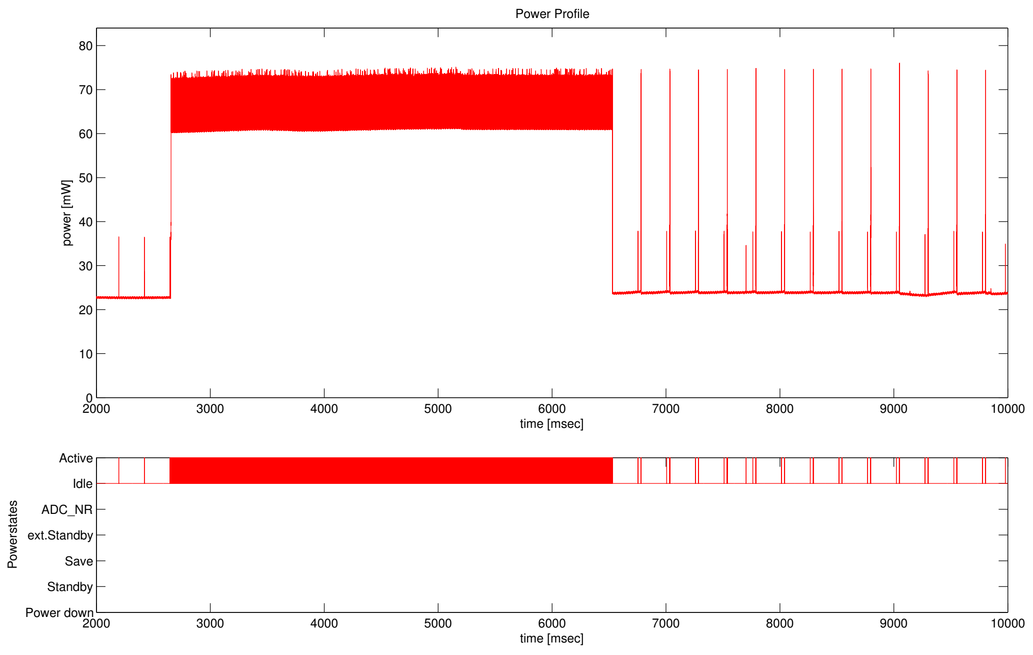


Figure 3.3: Power trace of asynchronous LPL

After receiving a message the LPL waits $30ms$ for the next message. If none comes it begins with its normal duty-cycle.

The problem described at the end of Section 2.3.2 and in Figure 2.7. There the sender doesn't know that the receiver listen for the next message. When the sender knows this, the preamble length can be reduced for every other message. In Figure 3.4 we see such an approach. Also here will not only saved energy but also much time for sending all the data.

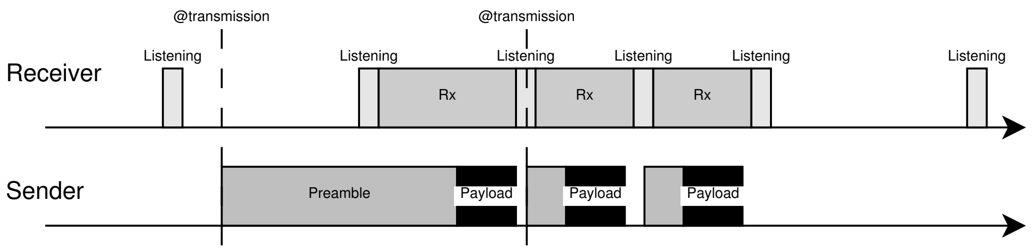


Figure 3.4: Asynchronous LPL with three messages sent

Since TinyOS has no automatism for this, the middleware had to implement something to increase power saving through sending more than one message.

3.2 Agents and virtual organisation

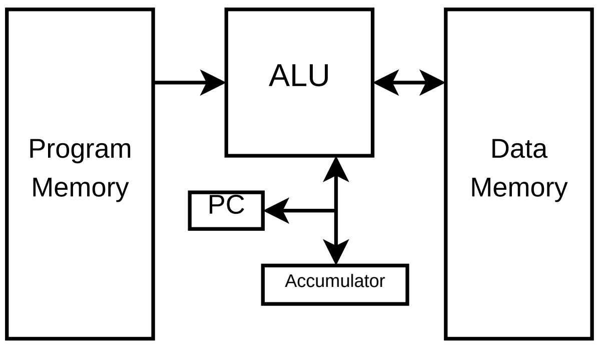
After the discussion of the benefits of the hardware we will discus in this section the System of agent and VO.

The virtual organisation is in the MultiAgent-MiddlewAre (MAMA) a logic clustering of agents and nodes in the network. As seen in Figure 3.1 a VO can consist of one or more agents and span over more than one node. The nodes do not need a direct communication link, but they must been able to exchange messages.

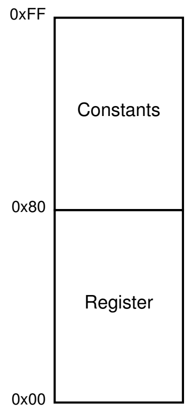
3.2.1 Abstraction of an agent

The agents can be seen as a register/accumulator-machine. They execute some byte code in a VM [13]. Each command consists of one byte instruction and up to 3 *memory address* for the operands.
The VM for each agent is just an application for MAMA. The great difference of applications in the multi-application middleware and an agent here is, that an application knows it behaviour at compile time, an agents gets it at run time.

Model of Computation: As we see in Figure 3.5a an agent has a separate Memory for Data and Program. The program memory is a read only memory. The ALU fetches an instruction from the program memory and all the addresses of the needed operands. Then it will execute the instruction and stores the result into the given operand. The communication with the data memory, the program counter (PC) and the accumulator is bidirectional.



(a) Model of computation



(b) Memory map

Figure 3.5: Memory map and model of computation of an agent in MAMA

Data memory of an agent: This 8 bit *memory address* defined as the MSB as identifier for constants (Parameter of the agent) or variables (register of the agent) and the other 7 bits used to point to the right field. As seen in Figure 3.5b is the upper half of the memory space used by the Constants of the agent and the lower half for registers.
Each of the Constant or Register are 16 bit broad. This is enough space to take an AM address or a data read from a sensor.

Instruction of agent: The instructions of the agents are directly mapped to the system calls used by applications in MAMA or they are instructions used for a control-flow of the agent. There are also instructions to manipulate variables.
Nearly each instruction can be called with or without accumulator. A detailed description of the instructions comes now:

1. Control flow

   HALT: stops the agent.

   JMP: jumps to position OP_1 in program.
   $PC = OP_1$[1]

   SLEEP: puts the agent for OP_1 milliseconds to sleep.

   LOAD: loads the given variable to the accumulator.
   $A = M\{OP_1\}$

   STORE: stores the accumulator in the given variable.
   $M\{OP_1\} = A$

2. ALU

   INKR: increments the variable given with the address.
   $M\{OP_1\} = M\{OP_1\} + 1$

   INKRA: increments the accumulator.
   $A = A + 1$

   ADD: $M\{OP_1\} = M\{OP_2\} + M\{OP_3\}$

   ADDA: $A = A + M\{OP_1\}$

3. System calls

   SEND: Sends $M\{OP_1\}$ to $M\{OP_2\}$.

   SENDA: Sends A to $M\{OP_1\}$.

   RECV: receives a value and stores it in $M\{OP_1\}$.

   RECVA: receives a value and stores it in the accumulator A.

   READ: Reads the sensor given with OP_1 to $M\{OP_2\}$.

   READA: Reads the sensor given with OP_1 to A.

   SETLED: Displays the value of $M\{OP_1\}$ on the LEDs (3bit wide).

   SETLEDA: Displays the value of A on the LEDs (3bit wide).

As we see the instruction with the accumulator need less space in Program Memory than the other. In Table 3.2 we see a summary of all instructions. In summary we choose this system because some simple agents can be written with little program-size and in more complex programs we can save some space. Now we look at some easy calculations and how they implemented in the two systems. $A+B = C$ is build with a register-machine with one instruction `ADD C A B` and needs 4 byte program space. The accumulator machine needs for the same operation also 4 bytes of program space but two instructions `LOAD A; ADDA B` and the result is in the accumulator. When it is needed to be stored in one register this

[1]A. . . accumulator
PC. . . program counter
OP_i. . . the i-th operand
$M\{i\}$. . . data memory on position i

[2]The LOAD and STORE instruction does not make any sense without an accumulator

Instruction	Accumulator	Register
`HALT`	0	0
`JMP`	1	1
`SLEEP`	1	1
`LOAD`[2]	0	-
`STORE`[2]	0	-
`INKR`	0	1
`ADD`	1	3
`SEND`	1	2
`RECV`	0	1
`READ`	1	2
`SETLED`	0	1

Table 3.2: Operands per instructions with and without accumulator

machine needs a third operation.
This simple operation does not show the advantage of the accumulator operations. But when send the result of the calculation to another agent D. Then the register machine had to use the `SEND C D` instruction with 3 byte. With the accumulator machine we only need the `SENDA D` instruction with 2 bytes and so we save one byte in the program.
For some examples see Section 3.4 later in this thesis.

3.2.2 Messaging

The message system of the agents is an extension of the system of MAMA seen in Section 2.3.3. We had to alter the header a bit to bring the ids of agents and VOs in the concept. In Figure 3.6 we see the message types sent by the multiagent middleware.

In Figure 3.6 we see the envelope for the messages used by the middleware. It consists of a 7 byte header. The difference from the header MAMA uses is that instead of the application ID of source and destination here we use a 4 bit ID of the virtual organisation and a 4 bit ID of the agent in this VO. The field `TTL` is a time-to live. It is counted in hops. The networking system counts every resend also as a hop. Then we have the length field. This is the same as it is used in MAMA.

Then we see in Figure 3.6 three types of payload. The Data is used by an agent to send a value, read by a sensor or somehow calculated, over the network. It has the same size as a register or a constant in the memory. The last one is a command sent from the power console to a node.

Infield programming

The idea behind the agents is that they are inserted in the network on runtime [23]. But they are not a whole program to be sent like in Deluge [33] or a compiled object which needed to be linked on the node like in [27] and [22].
It must be possible to put the whole agent in only one middleware message. As seen in Figure 3.6 an agent has 2 Parameter each 2byte long and 16 byte of instructions. This

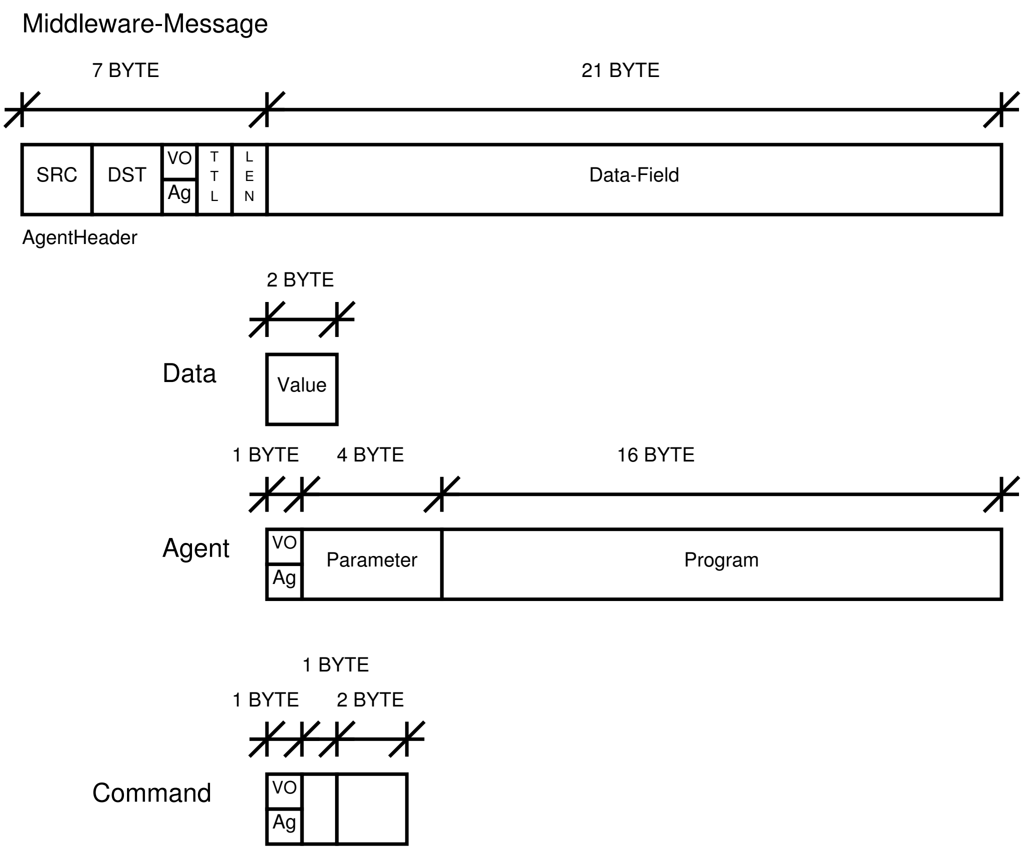


Figure 3.6: Structure of MAMA messages

16 byte are approximately 7-8 instruction with it operands. This is enough for a simple *sense and send* application or *Blink2Radio* application.

Agent to agent communication

The agent to agent communication is build with the `SEND` and `RECV` instruction. This two operations uses only one value to communicate between two agents in the same VO. This is a small disadvantage of the MultiAgentMiddlewAre.
This value is enveloped by a middleware message seen in Figure 3.6. It is the first of the three message types in this figure. It is called *Data* in this figure.

Commanding an agent

Commands to an agent or better to the middleware to interact with an agent is build with the message type from Figure 3.6 called *Command*. It contains the identification of the agent (VO and agent), then the command it self and if necessary a parameter. This operand is used for the mobility commands (see Section 3.2.3) to give the destination.

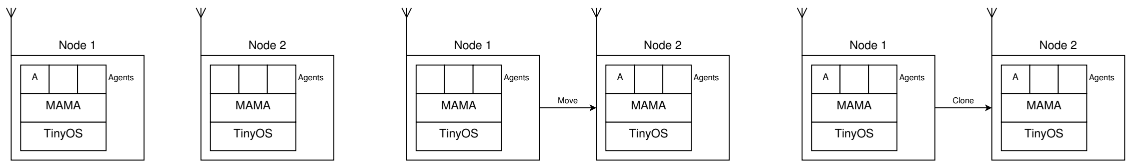
To identify all types of Messages used in the middleware the agent identification in the Header is used. The VO with the ID `0xF` is the middleware it self. The agent-ID is then the identifier of the message type, Command or a new agent.

3.2.3 Mobility and mobile agents

Each agent has the ability to move from one node to another. This comes in two different types.

Move takes the *memory* of an agent with it. Here has the node to be halted and then the agent (program and parameter) *and* the *memory* has to be moved to the designated node. There the agent runs at the same position as on the old node.
We see in Figure 3.7b how the network evolves from 3.7a with the move command.

Clone is the ability to move a copy of the agent to another node. Here for the agent copies only its program and maybe its parameters, and sends it to the designated node. There the new agent will start.
We see in Figure 3.7c how the network evolves from 3.7a with the clone command.



(a) Before move or clone an agent (b) After move agent A to node 2 (c) After clone agent A to node 2

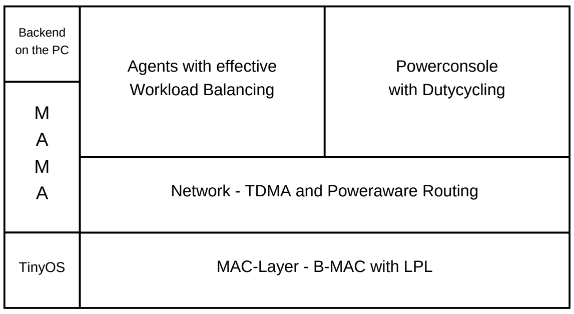
Figure 3.7: The mobility of agents

This abilities of the node can be used to prevent a node to use up all its power. a node, which power is low, can send an agent to another node in its neighbourhood. This can be one of the power saving features of the power management.

3.3 Power management

Power management is the heart of this middleware. Since we have battery powered nodes or nodes with an EHS, we need to build up a service that will handle this limited resource.

The power management is build up in layers as seen in Figure 3.8.
The lowest layer is the MAC-Protocol. Here we use the standard B-MAC implementation in TinyOS with the low-power listening. On top of this MAC-Layer we use a Power-aware-routing protocol, which ensures that transport a message from the source to the sink will be energy efficient. Also we will use a TDMA mechanism to access the radio channel to prevent collisions on it. Each time slot of the TDMA will be 5 to 10


Backend on the PC
Agents with effective Workload Balancing
Powerconsole with Dutycycling
M A M A
Network - TDMA and Poweraware Routing
TinyOS
MAC-Layer - B-MAC with LPL
Figure 3.8: Levels of Powermanagement

messages long. On top of the messaging system is built the agents and the workload balancing. Side by side with them, there will work the power console with the duty cycling and the agent system with the workload balancing.

Every node in the network sends periodical data about its energy state. This data the power management collects and calculates the fitness of the network (see 2.1.2).
A node i can have three different states.

Connected: $\kappa(t, i, bs) = 1 \wedge i$ **active at** t

Alive: $\kappa(t, i, bs) \neq 1 \wedge i$ **active at** t

Dead: $\kappa(t, i, bs) \neq 1 \wedge \neg(i$ **active at** $t)$

To determine these three types we use the equations (2.5a) and (2.6). Since we only can obtain the state of a node, when it is connected to the basestation (Node ID 0) and sending its energy state, we just look at three points in time for each node. The time $t_0(i)$ is the time, when the last message from node i arrived. $t_0(i) + T_C$ is the point in time when we consider node i as alive but not anymore connected to the basestation. T_C is the timeout of the next energ ystate to receive. The next point in time is $t_0(i) + T_A$ where $T_A > T_C$. T_A is the timeout the power management waits to consider a node alive.
when duty cycling is active and the nodes sends each cycle its energy state, then $T_C \geq n{*}T_D$ and $T_A \geq m{*}T_D$ where T_D is the duty cycle and n and m says how many duty-cycle we wait until the node is considered disconnected or dead.

3.3.1 Duty cycling

Discussed in [5] and in Section 2.2.3 we use this approach to implement the power aware scheduler. Since the abstraction of the applications is at the node we mostly use the power management to save energy.

Duty cycling we can use for more than one aspect in the middleware. First we can use it on the network. Beside the LPL from B-MAC, we can turn on and off the AM-modules in TinyOS, so that we can not send or receive any messages and save the energy to hold the receiver online.
Second we can turn on and off the VM for the agents. This helps to preserve energy, while the node does not need to execute the agents, send its data or read any sensors.
But to turn of any agent is only valid, when none of the neighbours can take this agent. So this is part of the workload balancing.

Network

Duty cycling on the network means we had to manage the times when a node is listening to the network or sending on the network.
[21] brings some pattern to build a duty cycling. They assume that a message can only do one hop per cycle, because they speak of duty cycling in the MAC-protocol (like the LPL in B-MAC).
They just assume that the nodes are all tracing the listening time of each neighbour. So

the node can send a message toward to a sink when they are listening or toward to the network when the other nodes are listening.

In this middleware we just use a full synchronised network with duty-cycling. This means that all nodes are up at one time and listening to the channel and sending their messages. Therefore, a synchronising mechanism is needed and also a system to distribute the information about the duty cycle thought the network.
This system is similar to S-MAC [32], but is not build in the MAC-Layer. To access the physical medium we will use the B-MAC [28] Protocol which is already implemented in TinyOS and used as its standard MAC-Protocol.

Time synchronisation

Since the network is needed to be synchronised, so that data can flow from a source to the sink, we need some Protocol to synchronise the nodes.
Since this problem is not one of our main concerns, we just implement a very simple system. We will send each Cycle a message, that contains a time in the future when the UP-time will end and the information about the actual duty-cycle (UP time, DOWN time and LPL-sleep time).

A possible other solution for this problem is to integrate the flooding time synchronisation protocol (FTSP) [26] from TinyOS into the middleware. This will provide a time synchronisation with a variance below $1ms$, which is much more than needed in this middleware.
An other advantage of this will be time stamping of sensor data and messages when needed.

Low power listening

Low power listening is a duty-cycling on the MAC-Layer. An introduction of this system is brought in Section 2.3.2. Here it is only mentioned to be complete with all duty-cycling approaches in the Middleware.

Finally, I use in this middleware only a duty cycling of the network. Since the agents are very small programs, they won't consume much energy, and the mote will sleep most of the time. Also we will test the LPL with this duty cycling.

3.3.2 Mobility

Mobility is also a concept to keep the network alive. But we don't want to move the nodes like [5]. More we want the agents to move from one node to another, if it is better for the network.
Here we use the ability of the agents to move through the network. So can an agent move from one node to another, if on its host the energy is low, or it would be efficient, in terms of power consumption, to work on the other node.

3.3.3 Modes of the power console

The power console is the controlling unit of the Middleware running on the PC called MAMA in Figure 3.1. Here, an admin can view the status of the network and can control the different nodes.
This control-unit can run in an autonomous mode and a manual mode.
The auto mode the power console controls the behaviour of the power management trough some rules.
In the manual mode an administrator can send agents from one node to another or shut them down. He can even change the duty-cycle of the network.

3.4 Testing

This section shows some VOs with their agents. These VOs are used to test the middleware and the power management.

3.4.1 Routing

For the test cases we just implement a simple power-aware routing.
This routing protocol builds up its routing tables trough flooding from a BS. To get the *best* route back to it will take the hops and a weighed average of the energy on the route.

$$E_{R_i} = (1 - \frac{1}{x}) * E_i + \frac{1}{x} * E_{R_{i-1}} \tag{3.2}$$

Equation (3.2) shows how this weighed average of the energy is calculated. E_{R_i} is the Energy of the Route at node i and E_i is the Energy of the node i itself. The factor x is to determine the ratio between the actual energy of a node and the energy of the Route to this node. The Energy of the Route at node 0 (the BS) E_{R_0} is equal to E_0 (the energy of this node).

3.4.2 Base station

The basestation is a module of MAMA, and is little changed to support the VOs and agents.
Its only duty is to enable the communication of the middleware with a gateway and so with the Internet.
Since the USART-Interface on the mica2-node prevents this mote to go to sleep (as seen Table 2.2). This node has to be connected to a power source, to make sure the bidirectional serial communication from node to gateway is functional all the time.

An important detail to this basestation is, that it will only route messages addressed to the node to the gateway.

3.4.3 Agents

Agents are very small programs that will perform one atomic task. Many of them are clustered to a VO. They work with the KISS-principle.

The *Sense and Send* application has only one agent. This one will get a value from a sensor and sends it to the BS. In Figure 3.9 we see the automate of the *Sense* application. This is a just strait forward agent. There are no branches and only one jump to build the infinite loop to read the sensor data.

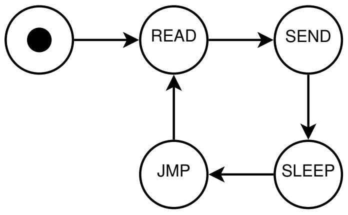
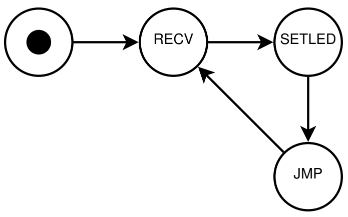
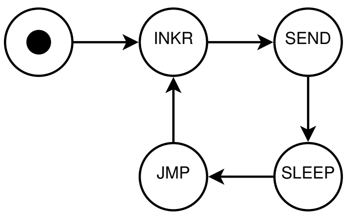


Figure 3.9: *Sense and send* application



(a) Receiver



(b) Sender

Figure 3.10: The *Blink2Radio*-application

For the *Blink2Radio* there are two agents, a sender and a receiver. The receiver gets the counter from the network and displays it on the LEDs. The sender increments the counter and sends it to a receiver over the network. In Figure 3.10 we see the automates for the sender and the receiver of this application. The sender (Figure 3.10b) is similar to the *sense and send*-agent, except that it does not need to read a value from a sensor, only to increment a variable. The receiver (Figure 3.10a) waits for a message from the sender and then displays it on the LEDs.

3.4.4 Virtual organisations

The VO is a logical grouping of agents, which perform together a task. Since every agent has its own small task to perform, we group them together in one VO to perform the function of the application.

As an example we use the *Blink2Radio* application. Here we need a sender and a receiver to perform the task. Each pair of sender and receiver builds their own VO.

The *Sense and send* application only has one task to do. But this task is done on several nodes simultaneously.

An extension to the *Sense and Send* application would be an agent, which averages the values of one or more *Sense*-agents.

3.4.5 Console

As console we call the system which runs on the PCs. Each VO has one to control its agents and analyse the gathered data. And also the middleware has one to control the power management and the network.

In the tests we will move and clone agents. We also test the service of the power management to change the duty-cycle during runtime.

3.4.6 Benchmarks

In most publications about WSNs the network is build for one purpose only. So the benchmarks they use are specific to the requirements the application running on the network needs. This middleware is much more complex. It gives the opportunity to use one WSN for more than one application. So the benchmarks used to qualify this middleware must be more general than others.

Some papers try to compare some parts of the software running on a WSN. Like [31] compares two different operating system (OS), or like [20], which compares different MAC-Protocols. But there is non that compares different middlewares, or the whole system. Maybe it is because you can't compare it. Like the problems in [31] to compare different OS.

The publications of different MAC-Protocols ([9],[28],[30],[32]) always brings some metrics to compare their new system with some old. Some of this metrics I will use in Section 5 to show the benefits of MAMA.

Metrics

[31] counts the time the mote is in a specific state and compares so 2 different OS. We will do some thing similar to show the performance of the power management. We also know the energy drain in each state, and so I can build on the basic of this knowledge a power profile of the middleware. Also is it possible to measure this power trace with the equipment presented by [18] and [19]. Power traces like this I have presented in Section 3.1.
So one of the metrics I will use to determine the behaviour of the middleware is their power consumption and the ratio of the different power states.

Other metrics are for the network. Here we will get the behaviour of the messaging system of the middleware. A stress-test of this system is done in [17]. For this tests is used the old middleware [7], but in this extension of the middleware, the part of the networking system is nearly untouched.

To show the advantage of this middleware we will present two metrics. On will be the energy consumption of the motes. It will be compared to different duty cycles and sleep times of the LPL. Then to show how good the implemented network system works we will compare the end-to-end delay of the messages with the duty cycle and the LPL-sleep time.
We also will explore the effect of injecting agents to the network. How they will change the delay when producing more traffic. Or how much more energy they will consume while running.

An other test case will be the testing of the workload balancing. Here the console counts the messages received from each node. If the difference between the minimum and the maximum of this count exceeds a threshold, an agent will be moved from the node with the maximum count to the node with the minimum count. So we can ensure that each node is has an equally workload.

Lifetime of WSNs

The impact of the duty cycle to the lifetime of a WSN is discussed in the previous chapter. In our test cases we will provide a measured proof of these ideas. I also will implement a possibility to calculate the condition of the network. This will be done with the models for Dietrich and Dressler [15].

3.5 Summary

This chapter of my master's thesis spoke about the mica2-mote and TinyOS. In Section 3.1 I explain the power analysis of the mote and the implementation of TinyOS and its features. In Section 3.2 I discuss the concept of the agents. I explain the messaging-system and the mobility of them.
Section 3.3 brings an overview of the strategies to conserve the energy on the WSN.
In the last section I spoke about the testing of the system and the metrics I want to use for the middleware.

Chapter 4

Implementation of the middleware and the console

In this chapter takes the concept and design from Chapter 3 and shows how that is built into TinyOS [25].

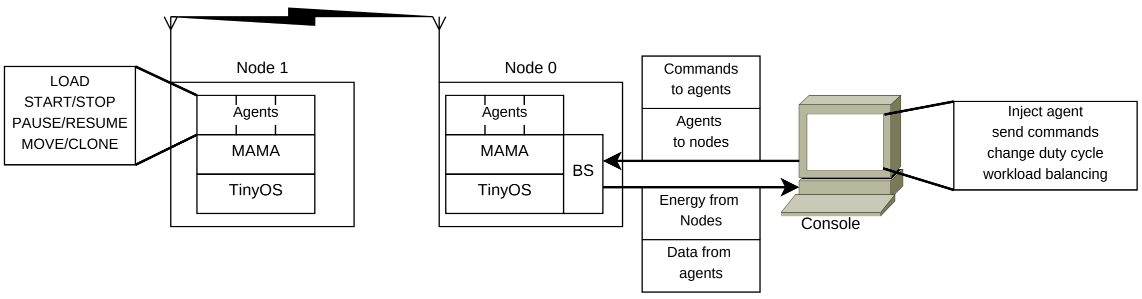


Figure 4.1: Overview of the implementation of the middleware

In Figure 4.1 we see an overview of the implementation of the middleware. This figure is a section of Figure 3.1. We see two motes and the server. The console runs at the server. From there it controls the duty cycle of network via the BS. The agents are running on top of the middleware. The middleware has to manage the agents and their VMs. The middleware presents services to access all resources at the mote to the VM. It also manages the network and the routing. The duty cycling is executed at the mote in the middleware but managed from the console.

The first part shows the agent. It presents the implementation of the instructions, how an agent is loaded into the system, the command dispatching and discusses the mobility. The second section is about the power management. First Part brings the implementation of the time synchronisation. Second section presents the duty cycling and how it is published in the network. The last section of the power management discusses the Workload balancer.
Section 3 describes the implementation of the server side of the middleware. In a few words it shows the how the system is implemented and where a user program has to be

plugged in.
The last section shows how the test cases discussed in Section 3.4 are implemented and what an environment is used to run the tests.

4.1 The agent and the virtual machine

The agent is a small program running on this Middleware. It consists of a sequence of Instructions on the memory of this Node. it is also possible to run more than one agent on a Node.

4.1.1 Execution of the instructions

The instruction set of the agents is just an `enum` defined in a header. The definition you see in Sourcecode B.1. The details description of the instructions can be seen in Section 3.2.1. The instructions can be split into 3 parts.
The first part are `HALT`, `JMP` and `SLEEP`. These are the one to control the agent itself. `LOAD` and `STORE` are to get or put a value to the accumulator.
`INKR(A)` and `ADD(A)` are the mathematical instructions.
`READ(A)`, `SEND(A)`, `RECV(A)` and `SETLED(A)` are calls to system calls of the middleware.

In appendix B at page 90 and 91 you will find the code for the agent. It shows how all of the instructions are executed.
Sourcecode 4.1 is an example how an executions looks like. The fetch is nothing else as to

```
57          case INKR:
58            (*memory(ag->prog[mem.pc]))++;
59            mem.pc+=1;
60            break;
```

Sourcecode 4.1: The INKR-instruction in the agent

get the op-code from the program-array at position `PC` (you see this in Sourcecode B.2 in line 42). After that the `PC` stands at the first operand. The `INKR`-instruction increments the variable. We see the memory function gives us the pointer to the right variable and then it will be incremented. In line 59 the number of operands is added to the `PC`.
Sourcecode 4.2 is an example of the system calls. This is the `READ`-operation. This requests

```
71          case READ:{
72            uint16_t val;
73            if(call SensorRead.read(ag->prog[mem.pc], &val)==SUCCESS)
74              *memory(ag->prog[mem.pc+1]) = val;
75            mem.pc += 2;
76            }break;
```

Sourcecode 4.2: The READ-instruction in the agent

a value from the Sensor given by operand 1 and stores it in the memory at position of operand 2. In line 73 we see the call to the middleware. Line 74 is the storage instruction and in line 75 the number of operands is added to the PC, and it points to the next instruction.

4.1.2 Load

The load is invoked when a agent arrives. It is signalled by the value of `0x0` in the agent field. This means that the payload of the middleware message is an agent.

```
161   void createAgent(agent_t *msg){
162     uint8_t  ID = findAgent(getID(msg));
163     agent_t* agent = NULL;
164      if (ID == MAX_AGENTS){
165        ID = findFreeAgent();
166        agent = call Pool.get();
167        if (ID == MAX_AGENTS){
168          call Pool.put(agent);
169          return;
170        }
171      }else{
172        if(agentStatus[ID] == CREATE || agentStatus[ID] == STOPPED)
173          agent = progs[ID];
174        else{
175          return;
176        }
177      }
178     *agent = *msg;
179     signal AgentRegister.createAgent[ID](agent);
180     agents[ID] = getID(agent);
181     agentStatus[ID] = CREATE;
182     progs[ID] = agent;
183   }
```

Sourcecode 4.3: System: load an agent

The first thing to do is to look if an agent with this ID is already loaded (Sourcecode 4.3 line 162). If there is one the middleware sets the pointer to the program (line 173). If there is none the system looks for a free agent (line 165) and gets a new slot for the program (line 166). In line 167 the agents will be copied and in the next one it will set to the right slot. In the lines 180 to 182 all the status-variables are set.
The created agent just stores its program and constants out of the sent and sets it internal status.

4.1.3 Commands

Commands are coming from the console via the network. Commands are identified by the agent-field (`0x1`) in the header of the middleware message. The VO-field is set to `0xf` to signal that it is forwarded to the middleware instead to an agent. Sourcecode 4.4 shows

```
void commandToAgent(command_t* com){
  uint8_t ID = getID((agent_t*)com);
   switch (com->com) {
     case STOP:
       break;
     case START:
       break;
     case PAUSE:
       break;
     case RESUME:
       break;
     case CLONE:
       break;
     case MOVE:
       break;
     default:
       break;
   }
}
```

Sourcecode 4.4: The command dispatcher

the code for the command dispatcher. In the first two lines it looks for the right agent. Then the commands will be executed on the agent.

Start

After an agent is loaded, it awaits the start command. When it comes, the agent creates the thread with the VM. This thread executes the function seen in Sourcecode B.2 and B.3.

In Sourcecode 4.6 we see the agent side of this implementation. Line 136 is where the thread is created. When this creation is finished TinyOS signals the asynchronous event `justCreated()`. Here we post the start task and this will call the `startDone()` command. Sourcecode 4.5 shows the part of the command dispatcher that starts an agent. It first checks if the agent is loaded and then calls the `startAgent` event. Sourcecode 4.7 shows the `startDone`. Here the middleware only traces the status of the agent.

Pause and Resume

The `PAUSE`- and `RESUME`-command are far easier to implement. We see this implementation in Sourcecode 4.8.
To pause an agent we just have to look if it waits for a system call. If it does so, then we signal it that we want to pause the agent. If not we can pause the agent directly.
To resume we do the same only the other way around. If the agent is waiting for the system call we do not have to do anything. If it is not waiting we just resume the agent.
The command dispatcher just looks when pausing an agent if it is a running one, or when it will resume one it looks if it is paused. You see this in Sourcecode 4.9. The dispatcher also sets the variables to trace the status of the agent.

```
        case START:
          if(agentStatus[loc_id] == CREATE){
              signal AgentRegister.startAgent[loc_id]();
          }
          break;
```

Sourcecode 4.5: System: command dispatcher starts an agent

```
event error_t AgentRegister.startAgent(){
return call DynamicThread.create(&thread, startThread, NULL,
    TOSTHREAD_MAIN_STACK_SIZE);
}

task void start(){
  if(state == CREATE || state == STOPPED){
    call AgentRegister.startDone();
    state = RUN;
  }
}

async event void ThreadNotification.justCreated[thread_id_t
    thread_id](){
  if(thread_id == thread)
    post start();
}
```

Sourcecode 4.6: Agent: start an agent

```
command error_t AgentRegister.startDone[uint8_t appID](){
  agentStatus[appID] = RUN;
  return SUCCESS;
}
```

Sourcecode 4.7: System: agent has started

```
event error_t AgentRegister.pauseAgent(){
  thread_t* t = call ThreadScheduler.threadInfo(thread);
  if(t->state == TOSTHREAD_STATE_SUSPENDED){
    t->syscall->state = SYSCALL_PAUSE;
    return SUCCESS;
  }else
    return call DynamicThread.pause(&thread);
}
event error_t AgentRegister.resumeAgent(){
  thread_t* t = call ThreadScheduler.threadInfo(thread);
  if(t->state != TOSTHREAD_STATE_SUSPENDED)
    return call DynamicThread.resume(&thread);
  return SUCCESS;
}
```

Sourcecode 4.8: Agent: pause and resume

```
      case PAUSE:
        if(agentStatus[loc_id] == RUN){
            signal AgentRegister.pauseAgent[loc_id]();
            agentStatus[loc_id] = PAUSED;
        }
        break;
      case RESUME:
        if(agentStatus[loc_id] == PAUSED){
            signal AgentRegister.resumeAgent[loc_id]();
            agentStatus[loc_id] = RUN;
        }
```

Sourcecode 4.9: System: command dispatcher pause and resumes an agent

Stop

Only a running agent can be stopped. In Sourcecode 4.10 the command dispatcher checks if the called agent is running, and then calls the `STOP`-command. When the `stopDone` is signalled the middleware set the status-variables and puts the slot for this agent back to the pool.
On the agent side of this command (Sourcecode 4.11) there is more to do. If the agent is waiting for a system call to return, we just signal it that we want to end the agent. When the system call returns, it will automatically destroy the thread. In case of an receive-system call we had to cancel this, because it is not sure this call ever comes back.

```
      case STOP:
          if(agentStatus[loc_id] == RUN){
            signal AgentRegister.stopAgent[loc_id]();
          }
        break;
```

Sourcecode 4.10: System: command dispatcher stops an agent

```
event error_t AgentRegister.stopAgent(){
  thread_t* t = call ThreadScheduler.threadInfo(thread);
  if(t->state == TOSTHREAD_STATE_SUSPENDED){
    t->syscall->state = SYSCALL_DESTROY;
    if (t->syscall->syscall_ptr == SYSCALL_WAIT_ON_EVENT)
      call MiddlewareRadioReceive.chancel();
    return SUCCESS;
  }else
    return call DynamicThread.destroy(&thread);
}

task void stop(){
  if (state == RUN){
    call AgentRegister.stopDone();
    state = STOPPED;
  }
}

async event void ThreadNotification.aboutToDestroy[thread_id_t
    thread_id](){
  if(thread_id == thread)
    post stop();
}
```

Sourcecode 4.11: Agent: stop an agent

```
command error_t AgentRegister.stopDone[uint8_t appID](){
  agentStatus[appID] = STOPPED;
  call Pool.put(progs[appID]);
  progs[appID] = NULL;
  return SUCCESS;
}
```

Sourcecode 4.12: System: agent has been stopped

4.1.4 Mobility of an agent

The ability of an agent to move through the network and to be programmed via the network is for the middleware nearly the same problem.
To inject an agent from a BS to a designated node is like a clone command. Since an agent fits into a single radio message we don't have to worry about fragmentation of the program.

Clone

Cloning an agent from one node to another is just like copying the program and its parameter to the other node. In Sourcecode 4.13 we see at line 310 that the program

```
302   void cloneAgent(uint8_t ID, am_addr_t addy){
303     RadioMsg_t msg;
304     RadioMsgHeader_t* header = call RadioPacket.getHeader(&msg);
305     uint32_t time;
306     if (call GlobalTime.getGlobalTime(&time) != SUCCESS)
307       return;
308     if(agentStatus[ID] == STOPPED || agentStatus[ID] == REG)
309       return;
310     memcpy(call RadioPacket.getPayload(&msg), progs[ID],
            sizeof(agent_t));
311     call RadioHeader.setDstAddress(header, addy);
312     call RadioHeader.setSrcAddress(header, call RadioHeader.address());
313     call RadioHeader.setSrcAppID(header, 0xF0);
314     call RadioHeader.setlength(header, sizeof(agent_t));
315     header->Time = time;
316     call SystemSend.send(&msg);
317     agenttostart = ID;
318     address = addy;
319     call Timer.startOneShot(256);
320   }
```

Sourcecode 4.13: System: clone an agent

```
276   event void Timer.fired(){
277     sendCommand(address, (agents[agenttostart]&VO_MASK)>>4,
            agents[agenttostart]&AG_MASK, START, 0);
278     if(agentStatus[agenttostart] == PAUSED){
279       signal AgentRegister.resumeAgent[agenttostart]();
280       signal AgentRegister.stopAgent[agenttostart]();
281     }
282   }
```

Sourcecode 4.14: System: start an agent after move or clone

and its parameter are copied. The value `0xF0` in the header-field indicates that this

message is for the middleware (VO `0xF`) and it is a program that should be injected in the node (agent `0x0`) (line 313). In line 316 the program is given to the network subsystem so that the program can be sent to its destination. In the last line of Sourcecode 4.13 (line 319) there is a timer started. This timer is needed to secure that the program arrives before the start command is given. This part is seen in Sourcecode 4.14.
In the `fired`-event is just send the start command to the destination of the agent. Since the same timer is used to start after clone or move, the system has to stop the paused agent when moving.

Move

Moving an agent is just the same as cloning. The agent itself has to be copied and sent via the network to its destination, but moving involves also the transfer of the memory of the agent. This is seen in Sourcecode 4.15. From line 277 to line 292 it is the same as the

```
276  void moveAgent(uint8_t ID, am_addr_t addy){
277    RadioMsg_t msg;
278    RadioMsgHeader_t* header = call RadioPacket.getHeader(&msg);
279    memory_t *mem = signal AgentRegister.getMemory[ID]();
280    uint32_t time;
281    if (call GlobalTime.getGlobalTime(&time) != SUCCESS)
282      return;
283    if(agentStatus[ID] == STOPPED || agentStatus[ID] == REG)
284      return;
285    signal AgentRegister.pauseAgent[ID]();
286    memcpy(call RadioPacket.getPayload(&msg), progs[ID],
         sizeof(agent_t));
287    call RadioHeader.setDstAddress(header, addy);
288    call RadioHeader.setSrcAddress(header, call RadioHeader.address());
289    call RadioHeader.setSrcAppID(header, 0xF0);
290    call RadioHeader.setlength(header, sizeof(agent_t));
291    header->Time = time;
292    call SystemSend.send(&msg);
293    memcpy(call RadioPacket.getPayload(&msg), mem, sizeof(memory_t));
294    mem = call RadioPacket.getPayload(&msg);
295    mem->pc--;
296    call RadioHeader.setSrcAppID(header, 0xF2);
297    call RadioHeader.setlength(header, sizeof(memory_t));
298    call SystemSend.send(&msg);
299    agenttostart = ID;
300    address = addy;
301    call Timer.startOneShot(256);
302  }
```

Sourcecode 4.15: System: move an agent

cloning of an agent.
In line 293 the memory is copied. The parameter 0xF2 in line 296 indicates that the message is the memory of an agent.

The last line starts the Timer to start the agent of the other node. Also a difference between move and clone is, when moving an agent, it will pause while it process and then stopped. While cloning it can work further, because this won't change the program which will be sent.

4.2 Power management

The power management is split into two parts. First of it is the duty cycling and the time synchronisation. Second part is the mobility and the decision where and when an agent is moved.

4.2.1 Time synchronisation

To synchronise the duty-cycles of the nodes we need to implement some protocol. Here we just implemented a simple protocol but in the first functional tests we have seen that we need a better solution.
We just send the information about the duty-cycle and the remaining time until the next Down-time out. This Data will flood through the network. This is seen in Sourcecode 4.17. Additional a sequence-counter is send with the data so that every node only updates the newest values.

One of the main problems in this simple synchronising is, to estimate the time from calling the `send`-command to the actual receiving on the node. When there is no traffic on the channel this time is about $20ms$. But it is not predictable when other nodes are sending. This brings some problem when synchronising the Nodes.

This can be prevented, when using the FTSP [26], which is implemented in TinyOS. This provides a service to the middleware to get a synchronised global time trough the whole network.
But the implementation of FTSP is not direct compatible with this middleware. First of the problems are that FTSP is not compatible with duty cycling on the network as MAMA uses it.
Second FTSP can not change its update cycle as it will be needed when using an adaptive duty cycle.
This problems are overcome with some slightly changes of the sending algorithm of the FTSP implementation in TinyOS. Now FTSP sends only in the right sending slot, and it is fully compatible with the adaptive duty cycling of the power management. Once each time slot each node sends tries to send a FTSP message. This message is only send when the node it self is time-synchronous with the root node (which is the basestation).

4.2.2 Duty cycling

For the duty cycling there is only the need of a timer, a task and a variable to trace the status. As we see in Sourcecode 4.16 the task `start_stop` does nothing else as starting a new Timer and starting or stopping the AM-Modules. The `Timer.fired` event only posts this task.

```
47  task void start_stop(){
48    uint32_t glob_time = 0;
49    current_slot = 0;
50    if(call GlobalTime.getGlobalTime(&glob_time) != SUCCESS)
51      glob_time = 0;
52    p_msg.start_time = glob_time;
53    calculateOntime();
54    if (status == STOPPED){
55      if(call AMControl.start() == EALREADY)
56        signal AMControl.startDone(SUCCESS);
57      if(p_msg.up_time && p_msg.down_time){
58        call Timer.startOneShot(p_msg.up_time*1024+p_msg.hops*EARLY);
59      }
60
61    }
62    if (status == RUN){
63      if(p_msg.up_time && p_msg.down_time){
64        call Timer.startOneShot(p_msg.down_time*1024-p_msg.hops*EARLY);
65        if(call AMControl.stop() == EALREADY)
66          signal AMControl.stopDone(SUCCESS);
67      }else{
68        if(isBS){
69        call EnergyRead.read();
70        call
            LowPowerListening.setLocalWakeupInterval(p_msg.LPL_sleeptime);
71        call
            Timer.startOneShot(1024*((p_msg.up_time)?p_msg.up_time:1));
72     }
73    }
```

Sourcecode 4.16: Duty cycling in MAMA

If duty cycling is deactivated by setting the up-time or the down-time to 0, the synchronisation process is called every up-time second. The up-time is also used to calculate the slots for the TDMA.
The duty-cycle needs to be published through out the network every DC. Sourcecode 4.17 shows the task, that is used to send out the DC. As we have seen in Sourcecode 4.16(line 52) at he beginning of each DC-Task, the `start_time` will be set. This time will also be sent out into the network to synchronise the duty cycle of each node.

Time slicing

To prevent the system while testing from to much collisions and resends i have a time-slicing implemented. This is necessary because the radio channel is a shared medium.
This is needed to ensure that only one node at a time is sending messages via the radio channel. The `Routing`-, `PowerConsole`-, `FTSP`- and `Network` module are only sending its messages, when the mote is allowed to send its messages. An agent sends its messages via the `Network` module. This module stores the messages in a queue and sends them when it

```
task void send(){
  if(TDMA_status == TDMA_SEND){
    PowerMsg_t* msg = call AMSend.getPayload(&am_msg,
        sizeof(PowerMsg_t));
    msg->cnt = p_msg.cnt;
    msg->up_time = p_msg.up_time;
    msg->down_time = p_msg.down_time;
    msg->LPL_sleeptime = p_msg.LPL_sleeptime;
    msg->start_time = p_msg.start_time;
    msg->hops = p_msg.hops+1;
    call LowPowerListening.setRemoteWakeupInterval(&am_msg,
        p_msg.LPL_sleeptime);
    call AMSend.send(AM_BROADCAST_ADDR, &am_msg, sizeof(PowerMsg_t));
  }
}
```

Sourcecode 4.17: Publishing the duty cycle in MAMA

is allowed to. So an agent is not blocked, when sending a message if the network is down or in receiving mode.

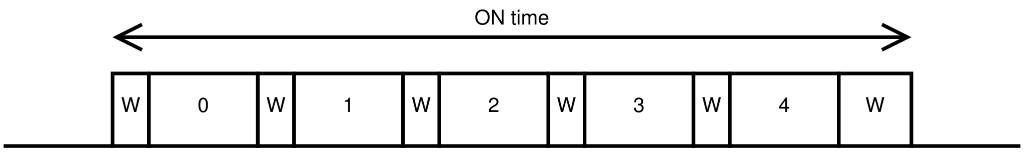


Figure 4.2: Schema of time slicing

The TDMA-technique is implemented in the `PowerConsole` module. The slots of each node and the total number of slots are given by the topology of the network. The total number of slots is the maximum number of neighbours plus 1 for the node itself. The assignment of the time slots should be so that all nodes are free to send (see Figure 4.4). Between the sending-slots are always some little time. This is needed to compensate the imperfect time synchronisation. Also is a short waiting at the beginning and the end of each duty-cycle planed. These waiting times are denoted as W in Figure 4.2 and Equation 4.1.

$$slottime = \frac{\textbf{ON Time} - W * (\#Slots + 1)}{\#Slots} \tag{4.1}$$

Equation 4.1 shows how the time for one slot is calculated on each node. This slot time is calculated on the beginning of each duty-cycle. This has to be done, because the **ON Time** can be changed.

The TDMA is started from the `start_stop`-task, where also starts the timer for the duty cycling. Sourcecode 4.18 shows parts of the time-slicing system. The task `TDMA_slot` is posted when the timer fires. This task just looks at the TDMA-status and then starts the next send slot or receive/waiting-slot. If it is the right slot for this mote (indicated by `SLOT`), the `startSlot` or `stopSlot` event is signalled. Each Module which wants to send has this two events implemented. When `startSlot` is signalled they begin to send and

```
task void TDMA_slot(){
  if(current_slot < SLOTS*N){
    if(TDMA_status == TDMA_RECEIVE){
      TDMA_status = TDMA_SEND;
      call TDMATimer.startOneShot(ontime);
      if((current_slot%SLOTS) == SLOT){
        signal PowerConsole.startSlot();
      }
      return;
    }
    if(TDMA_status == TDMA_SEND){
      TDMA_status = TDMA_RECEIVE;
      call TDMATimer.startOneShot(WAIT);
      if(current_slot%SLOTS == SLOT){
        signal PowerConsole.stopSlot();
      }
      current_slot++;
      if(current_slot == SLOTS*N)
        call TDMATimer.stop();
      return;
    }
  }
}

event void TDMATimer.fired(){
  post TDMA_slot();
}
```

Sourcecode 4.18: Time slicing in MAMA

when `stopSlot` is signalled the stop the sending procedure.

To use the Radio-channel in a more flexible and adaptive way, there also can be implemented a random back off system [24] or like in the `IEEE 802.3 CSMA/CD` standard [2] a Exponential back off. Both systems are not deterministic.

4.2.3 Workload balancing

At present I implement only the manual mode of the workload balancing. So the server side of the middleware decides when an agent is send to another node.
Therefore, a command message is sent to the node to move or clone the agent to another mote. This command message is interpreted by the command dispatcher as seen in Section 4.1.3.
When this command has to be sent can be decided by the console or the user with the help of the energy messages. These messages are sent by the `PowerConsole`-Module at the beginning of each duty cycle.
The console (see Section 4.3) also can be used to calculate the adaptive duty-cycle.

The idea for the automatic or adaptive mode of the workload balancing was that node decides themselves where and when to move an agent. This algorithm is not implemented and is part of further work and research.
But a simple system for workload balancing is implemented for the test cases. The console counts the messages it receives from the nodes and moves the agent from the node with the most messages to the node with the least messages.

4.3 Console

The console is the *server*-part of this middleware. It is implemented in Java with the interfaces TinyOS provides.

4.3.1 Middleware

The middleware class implements the communication between the VOs and TinyOS. This is just the same task as it has on the mote.
The Interface `UI` brings the services to inject an agent to the network and to send commands to agents in the network. It also allows changing the duty cycle. This can be done in an adaptive way or a static.

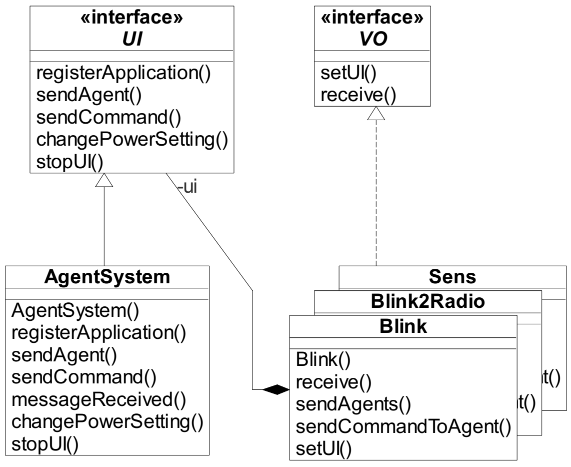


Figure 4.3: Console: class diagram

This class also holds an offset to calculate the date of the time stamp of a message. This offset is calculated of the `timesync` messages the BS sends on the beginning of each duty cycle to the PC.
This class has also the function to change the duty cycle, if the user requests it. It can also track the fitness of the network as discussed in Section 2.1.2.

To sum up, I'd like to say that the console has implemented the API to send agents and commands to the different nodes, to receive date for each VO registered in the system. It also calculates the offset between the global time in the network and the time at the server. The console also collects the energy information each node sends at the beginning of each duty cycle.

4.3.2 Interfaces

Each VO needs a class which implements the `VO`-Interface. In the class diagram (Figure 4.3) we see the `Blink`, `Blink2Radio` and `Sense` agent.
This interface provides the system with the needed methods. The `receive`-method is called from the `AgentSystem`-Class when a message for this VO is received. The `setUI`-method is called from the `registerApplication` in the `UI`-interface. It is need to tell the VO the correct implementation of the `UI`-interface.
Via the `UI`-interface each VO can send an agent or a command to its agents, or change the power settings like duty cycle or LPL sleep time.

4.3.3 The program

In the current version of this part of the middleware all interfaces and methods are given, but no algorithm for an automatism to move, clone, pause, start or stop an agent are implemented. Most of the time, the main-routine has to be changed when you want to measure something new.
Only for the test case workload balancing is a simple algorithm programmed, will automatically move an agent from the node with the highest workload to the node with the lowest workload.

4.4 Test programs

This section explains the environment of the tests discussed in Section 3.4. We also will present the implemented programs and how they are running and what parts of the implemented middleware they will test.
The last part here will show how we will measure the metrics presented in Section 3.4.6.

4.4.1 Environment

To test the middleware we use the mica2 node, the TOSPIE2-messurement system [18] and the Avrora simulator [29].

The Avrora simulator brings a cycle accurate simulation of the middleware. With the real-time monitor it simulates, as the name of the monitor says, in real time. Here we can use debug messages and other information about the motes, which are not available in the real hardware.

The measurement system and the mica2-Nodes are used to get the power traces, as seen in the previous chapters. With this system we can get timing information and energy information about the middleware and the agents, which can be used in design later.

Figure 4.4 shows the topology of the simulation. With this grid we will test the mobility of the nodes and the power managemt. We also use this topology to get the benchmark tests.

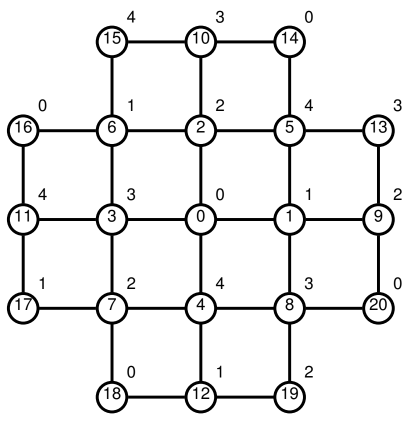


Figure 4.4: Topology of the test cases

4.4.2 Programs

The simplest agent to test the middleware is a `BLINK`-application. This application just counts up and displays it at the LEDs. Then it waits the time given by Parameter 0. In Sourcecode 4.19 we see the implementation of this agent.
The other agent in Sourcecode 4.19 is the `SENSE`-application. This agent has to read the Sensor X and sends this value to the BS given by parameter 1. Then it waits the time given by parameter 0.

```
INKRA
SETLEDA
SLEEP P0
JMP 0
```

(1) Blink

```
READA X
SENDA P1
SLEEP P0
JMP 0
```

(2) Sense

Sourcecode 4.19: The *Blink*- and *Sense*-agent

```
INKRA
SENDA P0
SLEEP P1
JMP 0
```

(3) Sender

```
RECVA
SETLEDA
JMP 0
```

(4) Receiver

Sourcecode 4.20: The Agents of Blink2Radio with accumulator

In Sourcecode 4.20 we see the sender and the receiver of the `BLINK2RADIO`-VO. This two agents are injected via the console to the network. After starting them the sender begins transmitting its counter-value to the receiver. The receiver displays the received value on the LEDs.

These agents are used to test the implementation of the VM and most of its features. The `BLINK`-agent tests the injection of agents in the network. With this small agent we can see without using any debug-system, if an agent is running and working or not.
The `SENSE`-agent is build to test the sensor-system and the network. It also can be used to get the delay of the network, when time stamping of messages is activated.
The `BLINK2RADIO`-VO consists of two agents, which are testing the communication within a VO. The VO are similar to the `BLINK`-agents, but split into two agents. One agent, the sender, counts up and the other, the receiver, displays the counter.

4.4.3 Benchmarks

After testing the functionality of the middleware with the test-programs like `BLINK2RADIO` or `SENS` we want to show some benchmarks make the middleware comparable. As discussed in Section 3.4 I want to show to metrics with this middleware. First we want to measure the delay of a message from its source to the destination. Second I want to show that using the power management of the middleware can really save energy.

Delay measurement

To calculate the end-to-end delay of messages I implement a time stamping for all middleware messages. With the help of this time stamps I can calculate the delay of each message.
This time stamp is also used by the networking module to sort out the expired messages, also messages they are taking to long to get to their destination.
The delay of each message is written to a file and analysed with a matlab script.

We will measure the delay of each message. Since I use a TDMA-policy to access the radio channel, and messages can be sent from an agent at any time, we will see not only the delay of transmission but also the delay of the TDMA and duty cycling.
So I expect delays up to one duty cycle at the basestation.

4.4.4 Energy measurement

The used power of the motes is measured with two different systems. First I will use the Avrora simulator [29]. To get the energy information out of this simulator, at our institute is a monitor developed that will bring that information and all the states of the system out.
With this I can determine the state of each component of the mote. We only will use the state of the μC, the radio and the LEDs. Other systems like the flash-memory or the sensors are not relevant or much to short to be for any use. This monitor also calculates the drain current of the mote and so we can calculate the power used at the moment.
The second part of the energy measurement is taken with the mica2-motes. To measure the power consumption of the real hardware we will use the measurement system from [18]. This system brings the power traces shown in the previous chapters. We will compare them with the results from the simulator, and discuss it.

4.4.5 Workload balancing

To show the function of the workload balancing the console has to count all incoming middleware messages for each node.
The first of this test runs with 2 nodes (the BS and one extra). An agent will be injected to the node and starts running. After the difference of message count will exceed the threshold the agent will be moved from one node to the other.
The second test case will be with 3 nodes (BS and two others). Here agents will be injected to the nodes 1 and 2. The console counts the messages it receives from the nodes and when the difference of the maximum of this counts and the minimum exceeds the threshold, the agent from the node with the maximum is moved to the node with the minimum.

4.5 Summary

In this chapter I present the implementation of the Multi-Agent MiddlewAre. The first section speaks about the virtual machine and the agent. We discuss the execution model of the VM and the commands, which can be sent to an agent. Also part of this section is the mobility of the agent.
The second section describes the power management. Features of this are the time synchronisation and the duty cycling. In the section duty cycling I also mention a static TDMA-technique to prevent collisions at the radio channel. The Section 4.3 presents the server side implementation of MAMA. It uses the JAVA-library from TinyOS to communicate with the motes and represents interfaces for the VO and agents. The section called test programs tells about the implementation of the test cases presented in Chapter 3.4. I bring up the implementation of this test cases and how they will be analysed.

Chapter 5

Results

This chapter shows the results of the implementation and changes. First, I show the effect of the changes in TinyOS [25]. The next section is how the agent works. Then the power management comes and its implicit energy conservation. Finally, the last section brings the outcome of the benchmark-tests.

The environment from Section 4.4.1 is used in all tests with more than one agent. This are the Sections 5.2, 5.3 and 5.4.
The Sections 5.2 and 5.3 are simple functional tests. There we only want to see if the modules are working as they should.

5.1 TinyOS

This section shows the outcome of the changes in the power management of TinyOS. The most important one is the possibility to turn off the SPI, when shut down the radio.

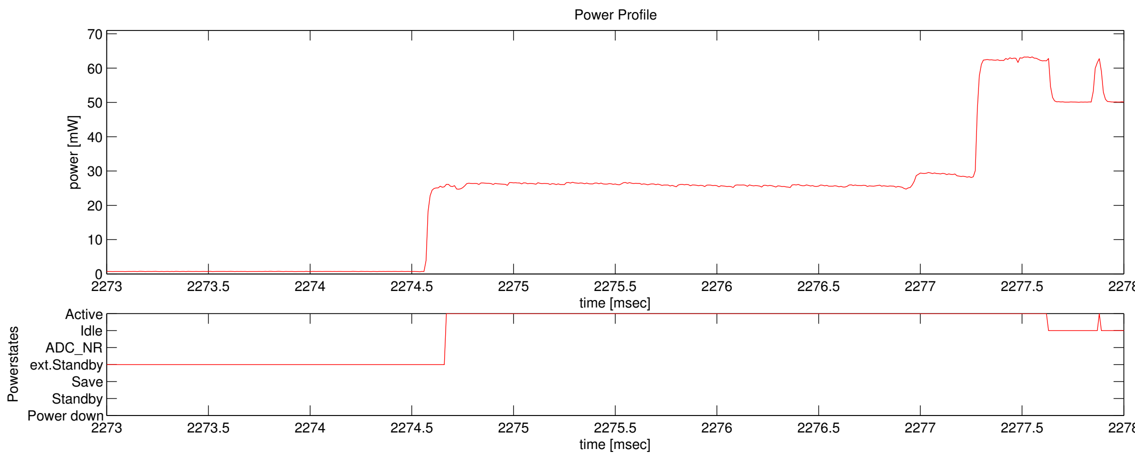


Figure 5.1: Power UP the AM component with SPI deactivated

Figure 5.1 shows the power-on of the radio module. This looks like the same as in Figure 3.2. The little difference is the beginning. Here the mote comes from the state

`ext. STANDBY` and not from `IDLE` as before.
The real energy conservation comes when using low-power listening. This we see in Figure 5.2. There we see that the mote needs nearly null energy between the on-times of the LPL. The duty-cycle of LPL is in this case 1:200.

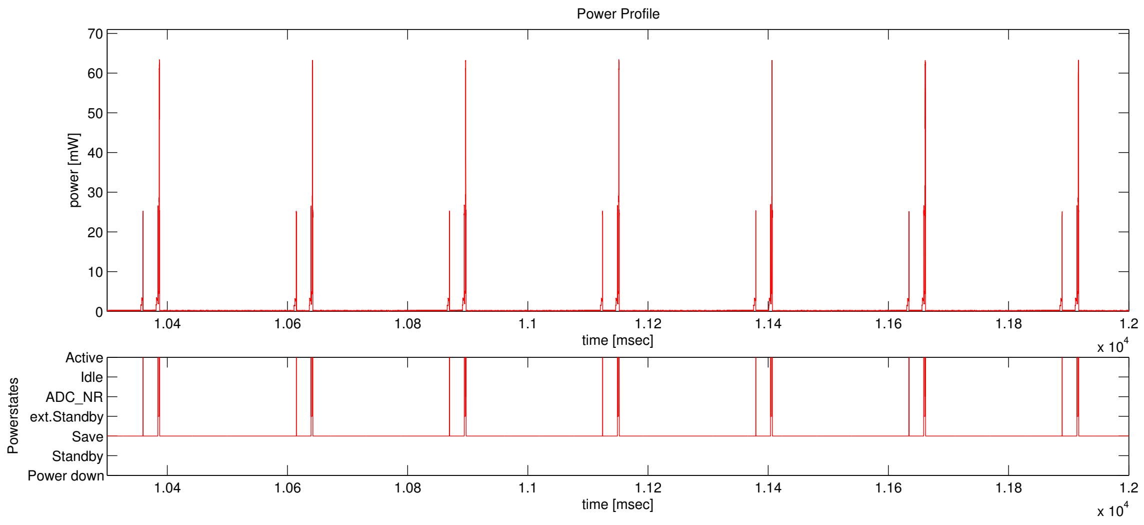


Figure 5.2: Power trace of asynchronous LPL with SPI deactivated

The Table 5.1 brings a summery of the power traces. The power values shown in the table are averages over 20 seconds.

Program	SPI always on	SPI can be deactivated
`NetworkTest`	ca. 30mW	ca. 21mW
`NetworkTest` with LPL	ca. 21mW	ca. 11mW

Table 5.1: Network tests.

The test programme called `NetworkTest` in Table 5.1 is a simple TinyOS application. The programme has only a timer running, that fires every 2 seconds. Each time it fires the programme toggles the state of the `AM` module. In second row, called *with LPL* we have used the same programme but started at with the timer also the LPL with 250ms sleep time.
As we see in the table is the effect of saving energy with LPL or by turning of the SPI the same. But by using LPL we have a much longer sending period as without. Using LPL and turning off the SPI we will safe two third of the energy in the mote.

5.2 Agents

Agents are the heart of the middleware. To test this little programs and the virtual machine, where they are running, was a difficult object.

In the first functional tests are tested the VM. Then is tested the injection of agents and the command dispatcher. And the last functional tests are the `Clone` and `Move`-commands.

5.2.1 Virtual machine

The VM is tested with static implemented agents. These agents are loaded into the VM at the boot and then started. Here I used a simple `BLINK`-agent.

The next stage of these tests is the tests of all system calls from the VM. Therefor I use two different VO. First one is the `BLINK2RADIO`. Here I test the send and receive Systemcall. This VO also tests the possibility to use more than one agent on one mote. The `SENS`-agent tests the sensor subsystem and the correct functionality of the basestation.

The `BLINK`-agent and the `BLINK2RADIO`-VO are used with a visual inspection with the simulator and on the real hardware. The `SENS`-agent is used to test only with the basestation.
For the `BLINK`-agent we want to see the LEDs blinking. They display the current value of the counter.
The `BLINK2RADIO`-VO does the same thing as `BLINK` but uses two agents, one to count up and send the counter value and one agent to receive the value and display it on the LEDs. When the LEDs at the *receiver* blink, we know that the network system and the system calls are working.
The `SENS`-agent takes a value from the sensor subsystem and sends it via the network to the BS. To display this value we use the console implementation for the server side with the `SENSE`-VO. This program displays the value at the screen on the PC.

These three simple tests ensure the ability of the middleware to work with agents and their VM, and test also all implemented subsystems.

5.2.2 Agent and command injection

This test case is worked out in three steps. First we tested the injection of an agent to the BS. Therefore an agent is sent via the serial-communication interface to the mote. There the basestation-module receives it and hands it to the Network-module. From here it is the same as if an agent is received via the Radio-Interface. The agent is loaded into the System-module, where it will be copied and then loaded into a VM. Then it waits for the command to start.
Second step is the tests via the radio-interface. Here it is injected via the serial-interface to the BS and from there via the radio-channel to the designated mote in one hop. There it is copied to the VM and waits for the start command.
The last step is the testing with a multi-hop routing. Here travels the agent via the network to its destination and is there loaded into a free VM.
This three test cases not only Tests the agent injection but also the commands to an agent. Not only is the start-command tested. Also in all three cases the pause/resume-command and the stop-command.

To control the functionality of the injection of the agents and the command dispatcher and the commands I used the visual inspection as before. For the `BLINK` and `BLINK2RADIO` VO I can see the results direct on the motes or in the simulator. For the agent I used the BS and the implementation of the console and the `SENS` VO at the PC. There I get the taken sensor data on the screen.

5.2.3 Mobility

Provided that the Network is synchronised and the routing-table is built, an agent can be moved or cloned though the network.
Mobility is just an extension to the agent and command injection. Since we know from the test of the last section, that we can inject an agent to the network, we know that it is possible to clone the agent. It is just the same procedure.
Little bit more complex is moving an agent. Here not only has a Program to be sent, also the memory of the agent has to move. Here it has to be looked at the execution of the VM. The normal memory of the agent can be transferred without any problem, but at the special register like the `ProgramCounter` or the Accumulator we had to act with caution. But this is all done in the VM itself. The injection of this memory into the agent works at it should be.

This is tested with the `BLINK2RADIO`-VO. Here is the Sender injected to the node 1 and the receiver to the BS, so we can receive the sent Counter on the PC. Then when it runs, node 1 gets the command to move the agent to node 2. Here we see that the counter runs normal further from its value as it wars before the move-command.

5.3 Power management

From the previous sections we see that the console can inject agents to the Network, receive data from the agents and send commands to them to start, stop, pause, resume, move or clone.
Now we look at the power-management system of MAMA. This includes the time synchronisation and the duty cycling. Workload balancing is part of the power management but not part of the tests here, because only the static system is implemented.

5.3.1 Time synchronisation

First we look at the time synchronisation. This is done with the modified flooding time synchronisation protocol. At the beginning of each TDMA-slot a `TimeSync` message is sent. After 4 successful received messages the FTSP considers a node as synchronised with the root node, and begins itself to send sync-messages at each TDMA-slot. So it is possible to synchronise the whole network. Does a mote not receive a `TimeSync` message over four duty cycle; the `TimeSync` module considers the node out of sync. Then the mote stays online until it receives enough `TimeSync` messages to resynchronise.
With the successful synchronisation of the node begins also the duty-cycling.

5.3.2 Duty cycling

When `TimeSync` module considers a node synchronised and it receives the published duty cycle, the node will begin with it.
From the published duty-cycle the node gets the global start time of this cycle. In the message also are all data for the cycle (up-time, down-time, LPL-sleep time). Here it starts the timer with the remaining time.

With the basestation module at the BS-mote the power console can inject a new duty-cycle into the network.

TDMA

As part of the duty-cycling mechanism the time-slicing or TDMA is also only active when the node is synchronised. For On times larger than 1 second we just multiply the #slots with the ON time in milliseconds. So each node can send more than once per duty-cycle.

$$slottime = \frac{\textbf{ON Time} * 1024 - W * (\textbf{ON Time} * \#Slots + 1)}{\textbf{ON Time} * \#Slots} \qquad (5.1)$$

In equation 5.1 **ON Time** is given in seconds, slottime is calculatet in binary milliseconds[1] on the mica2-node. $\textbf{ON Time} * 1024$ is the **ON Time** of the duty cycle in binary milliseconds. $\textbf{ON Time} * \#Slots$ is the number of slots in one duty cycle. And this number increased by one is the number of waits W. This formula allows the nodes to repeat the time slicing all second independent of the duty-cycle (or even if no DC is given).
No DC is given when the ON-time or the OFF-time is set to 0.

These three processes are dependent on each other. The LEDs are used as a visual inspection of these tests. The yellow LED is active when a mote considers itself as out of sync. The red LED shows the duty cycle. It is on when a mote has activated the network and off when deactivated. The green LED displays the TDMA slot of this node, when on it is allowed to send.
In the network shown in Figure 4.4 we can see when starting the network how the nodes are synchronised. The first node is the BS. Then the nodes 1-4 come, which are only one hop away from the root node. Then will be the nodes 5-8 turn off their yellow LED, then the nodes 9-12 and so on. The nodes 5-12 are all 2 hops away from the root node. But the nodes 5-8 are faster synchronised as the nodes 9-12 because they got `TimeSync` messages from 2 different nodes.

[1] TinyOS splits a second in 1024 parts and calls it binary milliseconds.

5.4 Measurement

No we come to the measured results of this master's thesis. First we just complete the description of the environment. A first look at the testing environment we got in Section 4.4.1.
Then we present the benchmark tests, like the delay. The last part of this section is about the effects we can get with the power management.

5.4.1 Environment

In Figure 4.4 at Section 4.4.1 we see the topology of the test system. In the middle of the grid is the basestation with the address 0. The number on the right upper of each node gives the assigned slot. In the grid we have a maximum count of neighbours of 4 so we need 5 time-slots [0 - 4].

5.4.2 Memory usage

TinyOS doesn't have a dynamic memory management. Each variable and pools of variables must be known at compile time. This allows the compiler to optimise the memory footprint of the programs.

This behaviour coerces us into using pools of memory instead of dynamic memory as used in `C`. One of this pools are used in the networking system to manage the in- and outgoing messages. An other pool is used for the agents. Also is a such a pool used for the stack of the VMs. This stack pool has the restriction that its size has to be a power of 2.

VMs	**RAM**	**ROM**	**stack size**
1	2289	38980	256
2	2911	42116	512
3	4045	43982	1024

Table 5.2: Memory footprint of the implemented middleware in byte

From our functional tests we know that a VM needs a minimum stack space of 200 byte. So we know that we need at least 200 byte for one agent, 400 byte for two and 600 byte for 3 agents. In Table 5.2 I bring the memory footprint of the middleware with one, two and three VMs. The presented memory usage is only the usage of all global variables and all memory pools used in the middleware. Auto variables[2] take place at the stack, and so we need at least 200 byte additionally to the presented value as stack for the kernel. So we see that the value of 3 agents (4045 byte) exceeds the 4 kbyte memory of the mica2 node.

[2]Auto variables are declared inside of functions.

5.4.3 Benchmarks

This section brings all the measurement. First we will discuss the Delay of the middleware. Then we look at the energy consumption and the powertraces of the middleware.
The delay- and energy test are running in the environment described in Section 4.4.1 and in the section before.
As spoken in Section 3.3 duty cycling will work at the network. So we turn on and off the network modules and save so energy, but we will see that this system will invoke a higher delay.

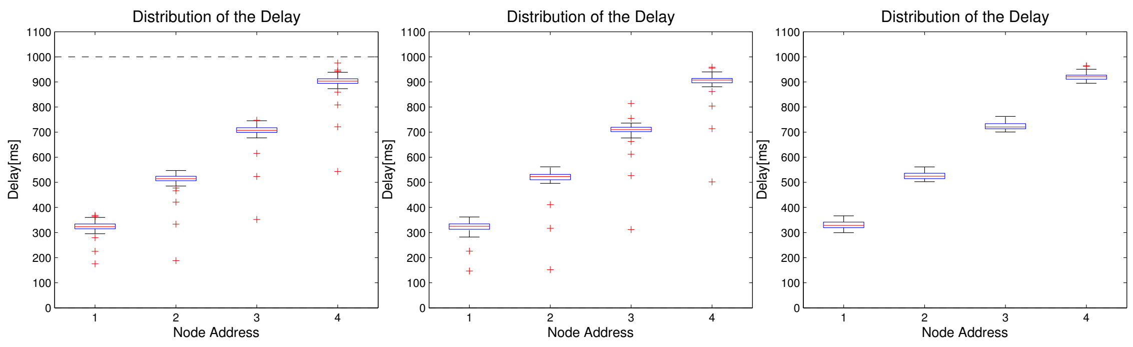
Delay

The delay of messages is measured on each message received by the basestation. It is the difference of the time given in the header (send-time) and the time the message is received on the PC. Therefore, the BS sends at the beginning of each duty-cycle a message to synchronise with the PC.

The tested duty cycles are 1:0, 1:1 and 2:1. This duty cycles are given in seconds and its **ON** : **OFF**. The first test case is for one second is the radio module on and never turned of. This is called *no DC*. The slots are calculated with the one second on-time.
The next duty cycle 1 : 1 means that the radio is online for one second and than it will turns off. Also there will be one sending slot per node per duty cycle.
The last test runs with a duty cycle 2 : 1. The two seconds on time allows the motes to send twice per duty cycle.

without active agent

These tests are simulated with the avrora-simulator [29] and measured with mica2-Nodes. Figures 5.3, 5.4 and 5.5 are from the simulations.



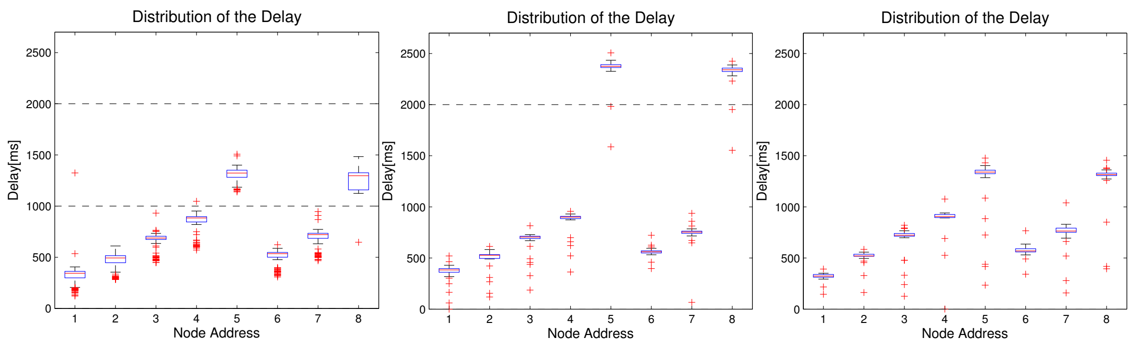
(a) No duty cycle (b) DC = 1:1 (c) DC = 2:1

Figure 5.3: Distribution of the delay on 4 nodes with different duty cycles

Figure 5.3 shows tests with only 4 nodes. We use the nodes 1-4 from Figure 4.4. These are the nodes, which are only a single hop away from the BS.
The pictures in Figure 5.3 show the distribution of the delays from the different nodes. The plots at the figures are `boxplots` from matlab. Figure 5.3a shows the test case with

no duty cycle. When we put the ON-time into the Equation 5.1, we get a slot time of about 174 ms. With the wait time of 20ms this is nearly 200 ms. So we can say that the slot 0 is from 0 to 200 ms, slot 1 from 200 to 400 ms and so on. The dashed line at 1000ms at Figure 5.3a is the end of the duty cycle.
We see in the Figure 5.3a that the messages of node 1 arrive with not more delay than we would expect from the time slots. 50% of the messages arrive at the PC with less than 320 ms delay. 50% of the messages arrive with a delay between 310 and 340 ms. For node 2 is the median at 510 ms and the 25th and 75th percentiles at about 500 and 530 ms.
Comparing the other plots we see that the delay and their distribution is the nearly the same.
We see that the delay (and its distribution) is for this test case independent of the duty cycle. Each message comes independently of the duty cycle in the same time. But we have to consider that in test case 2 and 3 that we will have less messages. In the second test case we have a duty cycle of 1 : 1 where every 2 seconds a message is sent to the BS. And in the third test case there is a duty cycle of 2 : 1. Here the node sends every three seconds a message to the BS. BUT the messages will take the same time to arrive at the PC.

Figure 5.4 presents the results of the tests with 8 motes. A look at Figure 4.4 shows that the nodes with the IDs 5-8 are 2 hops away from the BS. Looking at the motes with IDs 1-4 we see that their delay is just like the tests before. Only the variance of the delay is little higher as in case. This comes from the higher density of the messages. The *inner* nodes have to forward the messages from the *outer* nodes to the BS.

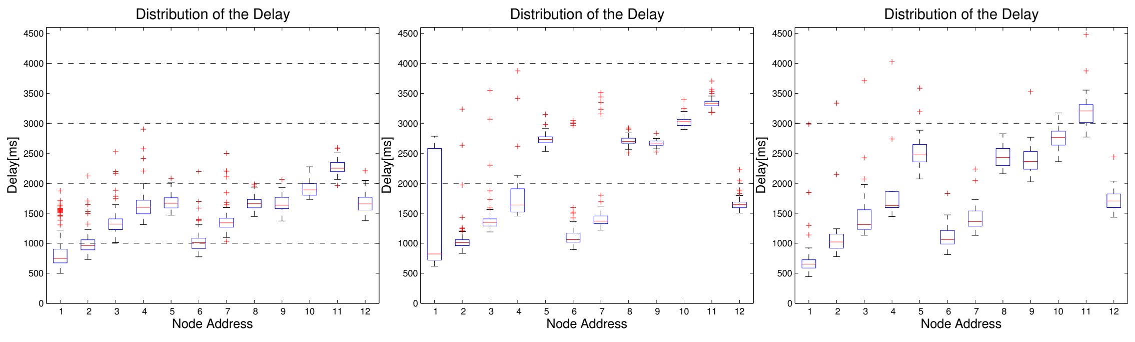


(a) No duty cycle (b) DC = 1:1 (c) DC = 2:1

Figure 5.4: Distribution of the delay on 8 nodes with different duty cycles

First we will look at Figure 5.4a. This is the plot from the test with the 8 nodes and no duty cycle. We see that the first 4 nodes (IDs 1-4) have little higher variance of the delay than in the test case before. Interesting is also the variance of the nodes 5 and 8 and their delay. For both nodes is true that they have more than one second delay. A quick look at Figure 4.4 shows that these nodes have only neighbours with an earlier sending slot. So, when sending a message from node 5 to the BS this message will go to node 1. Node 5 sends in slot 4, the last slot in the duty cycle. In the next cycle node 1 has to send the message from node 5 and its own message from the actual cycle. From there comes the

higher delay of node 5 and the higher variance at node 1 and 5.
Now we look at Figure 5.4b. This is the test with a duty cycle of 1 : 1. Here we have the same behaviour of the delay as in the test before. Only that the messages from node 5 have to wait at node 1 one DOWN time. This is the time between 1000 and 2000 ms. Above the dashed line in Figure 5.4b there comes the messages from the nodes 5 and 8.
The look at Figure 5.4c shows the same as Figure 5.4a, but with less variance at all nodes. In principle we have the same, but we have two TDMA cycles per duty cycle in this test case. So messages from node 5 will have to wait at node 1 to the next TDMA cycle, but node 1 has not to send its own messages at this sending slot. So we have less traffic at node 1 and therefore less variance in the delay.



(a) No duty cycle (b) DC = 1:1 (c) DC = 2:1

Figure 5.5: Distribution of the delay on 12 nodes with different duty cycles

In Figure 5.5 we see the *boxplots* of the simulations with 12 nodes. In this simulation we have simulated all nodes with a distance of 2 hops from the BS.
The first thing that catches our eye is the higher delay at all nodes, especial in the first test with no duty cycle. We can see the same structure in the plots. So we can assume the same behaviour in the network. But what causes the much higher delay at all nodes? The answer to this question is very simple. It is the basestation. The higher traffic in the network causes some jams in the BS. The high frequency of incoming messages at the BS, holds up the forwarding to the serial interface. So the BS has to wait until the network sleeps and than can forward the messages to the PC. This also causes the much higher variance at all nodes.
Beside this delay we have the same behaviour as in the tests with 4 or 8 nodes. With all this experiments we can extract a formula to calculate the maximum delay of a message. First we have seen that a message needs a maximum of `hops` TDMA cycles to reach the BS. From the tests with 12 nodes we saw that in environments with high traffic density we need the sleep time of the duty cycle to forward the messages to the server. From this experiments follows that we can expect that a messages receives in $\frac{\textbf{hops}}{\textbf{ON Time}}$ duty cycles, where **hops** are the count of hops to the BS and **ON Time** the on time of the duty cycle in seconds.

In the Apendix C we have the time traces of these delay measurements. Figure C.1, C.2 and C.4 shows the plots of the delay over the time for 4,8 and 12 nodes with a duty cycle of 2 : 1. There we see that the delay is time independent. Figure C.3 shows the time trace of the Figure 5.4b.

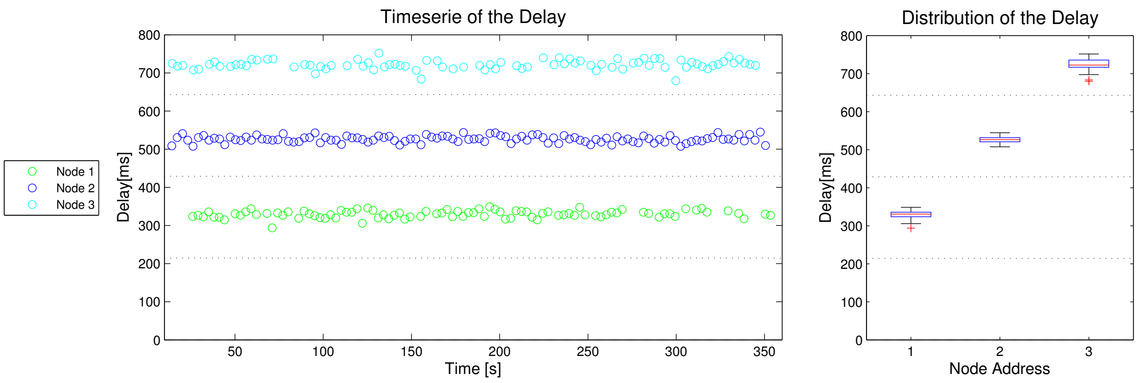


Figure 5.6: Delay of the mote grid with 3 nodes (DC = 2:1) measured on mica2-nodes

The results from the simulations are a little worse than the results from the real hardware. This comes from the Avrora-simulator. Simulating the middleware in Avrora has the disadvantage that acknowledgements not always are recognised or even sent. This brings MAMA to resend the packet and so causes much more traffic.
The test case on the mica2 nodes results in the expected picture. We see in Figure 5.6 that all messages came in their time slot. The delay on the real hardware is nearly the same as of the simulation. The right plot in Figure 5.6 has to be compared with the plot in Figure 5.3c.

with low-power listening
Next step in testing the delay would be the activation of the LPL. I have planed to do the same tests as without LPL and agents.
Starting the first tests with a LPL sleep time of 10 ms. After nearly the same time the nodes are synchronised. Now we start the console and look at the received messages. But there never comes one.
Running tests with more than 500 duty cycles we only got 1 or 2 messages at the BS. This is a package-drop rate of over 99%. In the tests before we got a package-drop rate of about 2-5%.
Analysing the messages sent in the simulation we saw that the nodes are sending the `Timesync` messages, the `Routing` messages and publishing the duty cycle as we expected. But they never send any `Data` messages via the radio channel.
This analysis and the fact that nodes can synchronise provides us with the knowledge that MAMA and LPL should work. But there is some problem with the TDMA mechanism, which prevents the network module to send `Data` messages via the radio channel.

May be it will work with another arbiter of the *resource* radio channel. This other arbiter policies can be the random back off policy [24] or a exponential back off policy [2]. They can maybe also provide a more flexible access to the radio channel.
Since the nodes can be synchronised with the BS, we will analyse the power consumption

of the middleware even with low-power listening.

with active agent

Now we come to the impact of the injecting of an agent.
The first test case is given by a 4 node network with a duty cycle of 2 : 1 and the agent is injected at 20 seconds from the beginning of the simulation.

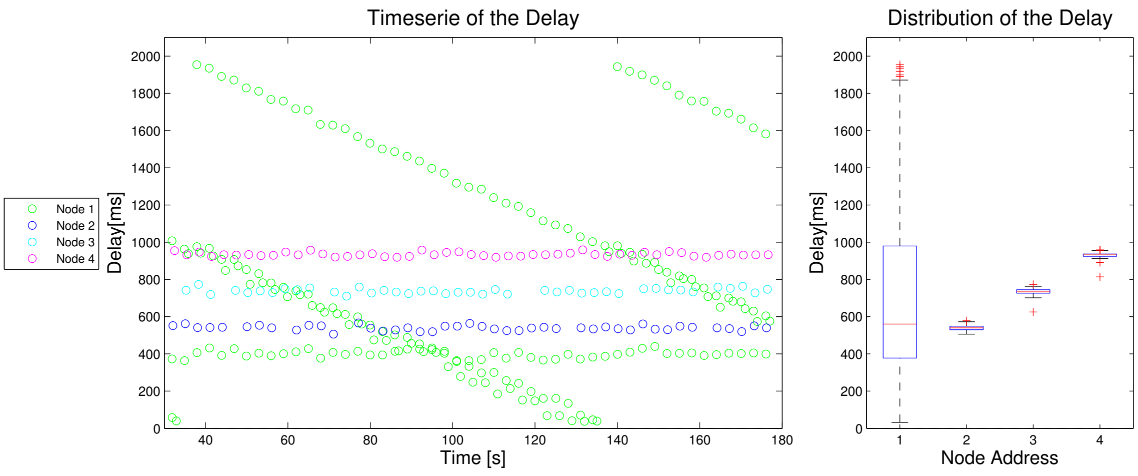


Figure 5.7: Delay of the mote grid with 4 nodes (DC = 2:1) and Sense-Agent working on Node 1

Figure 5.7 shows the diagram for a single-hop network with an agent working at node 1. This agent gets every second the value from the temperature sensor and reports it to the BS. We see in this figure the energy messages come in like without an agent (compare Figure C.1).
The green diagonal lines are the messages from the agent. This behaviour of the delay comes from the manner how an agent works.

Let's start the analysis of this behaviour with a look at the timing in the node 1. We take the time a duty cycle starts as time 0. At this time the mote activates the radio module and starts the TDMA-mechanism. We know that node 1 uses the slot 1. So the node can send its messages from time 200 ms to 400 ms and from 1200 ms to 1400 ms. Lets say that also at time 0 the agent starts working. The first command of the `Sense` agent is to capture a value from the sensor. This will take about 10 ms. So the agent gives its data at time 10 ms to the network module of the middleware, and than it waits 1 second. So it will awake at time 1010 ms.
At the beginning of each duty cycle the power console samples the voltage of the battery and sends it to the BS. At 200 ms when the node is allowed to send, the `Energy` message has a delay of 200 ms and the `Data` message from the agent has a delay of 190 ms. The network module works with a queue. First it sends the `Energy` message, which will take about 20 ms and than it sends the `Data` message. This one has now a delay of 210 ms. Then the node waits until the agent awakes. This happens at 1010 ms. Now the agent samples again the sensor and at 1020 ms it will give the new data to the network module and sleeps until 2020 ms. The network module will send the `Data` message at 1200 ms.

This message has now a delay of 180 ms. Then the node sleeps until 2000 ms. There the power console tells the radio module that the on time has expired and waits until 3000 ms when the duty cycle begins again. At 2020 ms the agent awakes and gets a date from the sensor. This will be given at 2030 ms to the network module. Then the agent sleeps again until 3030 ms. The message at the network module has to wait until 3200 ms, when the node is allowed to send the next time. Here the message gets a delay of 1170 ms.
In the next duty cycle the first message from the agent comes at time 3040 ms, which is 40 ms after starting the duty cycle. In the cycle before it wars only 10 ms after the beginning. This message will have only a delay of 200 ms, because before this message there will be sent the `Energy` message and the `Data` message from the sample before. The next `Data` message will come at 4050 ms and will send at 4200 ms (= delay of 150 ms). This decrease will go on until the agent sends it messages in the sending slot. There it will have a delay of nearly 0. In the next duty cycles the message comes after the sending slot and it have to wait until the next, which will be 1 second, if it was the first slot in the duty cycle, or 2 seconds, if it was the second slot.

We see in this picture that the maximal delay is 2 seconds. This happens when the agent sends its message shortly after finishing the TDMA-slot. Then the message lays one TDMA-round and one `OFF`-time at the queue before it is sent to the BS.

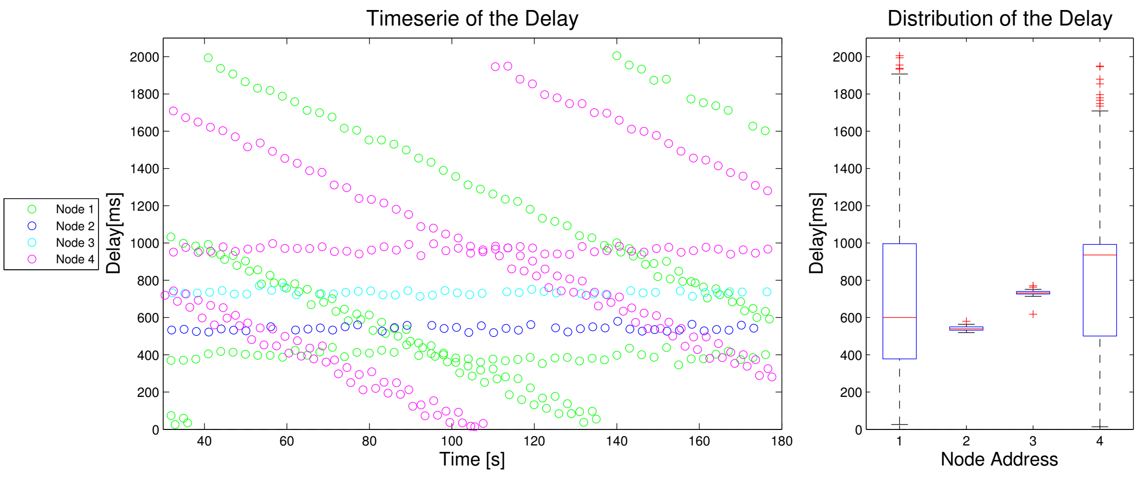


Figure 5.8: Delay of the mote grid with 4 nodes (DC = 2:1) and Sense-Agent working on Node 1 and 4

As we see in Figure 5.8 the activation of a second agent on another node has no impact at the delay of one message. This is one of the benefits of a TDMA system.
In this test case the second agent is cloned from the first one 1 second after it is started. Also this event isn't seen in the time series in Figure 5.8. But we see at the beginning of the measurement that at node 1 are for the injection of the fist agent.

Summary
To sum up, we see that the TDMA-mechanism provides a good system to grant access to a shared medium like the radio channel. This makes it also possible to calculate the time when a message should arrive. A big problem is the distribution of the time slots.

As we see in all the measurements the delay is as we expect and the messages will all come with a little variance. Also the package drop is below 10% in all networks even the big ones and with agents applied in the network.
The disadvantage of the TDMA is that the low-power listening does not work. This would be a good opportunity to save much more energy as we see in the next section.

Energy

Here we show some power profiles taken with the measurement system from [18] or with the Avrora-simulator [29].

The first power profile I bring up is from a measurement with no duty-cycling. This one is the maximum power consumption this middleware can have.
In Figure 5.9 we see this power trace of an mote. For better overview we only plot the states of the μC (and here only the states a mote can go in), the radio states and the LEDs to signal specific events. The red LED shows the duty cycle. Is it on then the node is in the `ON`-time and vice versa. The green LED indicates the TDMA-slot.

In Figure 5.9 we see at the first 6 seconds the synchronisation of the mote. Therefore it stays on and only listens to the radio and receives all messages send from the root-node. After the synchronisation the node begins with the duty-cycling and the TDMA. Looking at the green LED and the state of the radio we see that in each time-slot the node sends messages. These are the time-sync messages and the routing messages.
In this experiment I use a DC of 1 : 0. As we know from previous chapters this means that duty-cycling is deactivated, and the nodes stay online all the time. This setting produces the smallest delay but costs most energy. Each message arrives in maximal 3 seconds in our test-network (maximal 3 hops).

In Figure 5.10 we see at the beginning about 10 seconds the synchronisation. Then we see each transition of the mote. According to the duty-cycle the state-transitions of the radio are good seen. We also see that the Radio only sends date when it is allowed to.

In the experiment which is used to take the measurement in Figure 5.10 we used a duty-cycle of 2 : 1. This means that we save about 30% of energy compared to no duty-cycling and we can grantee that a message is delivered in our test-network in maximal 4 seconds.

LPL	duty-cycle					
	1 : 0 (100%)	2 : 1 (66%)	1 : 1 (50%)	1 : 2 (33%)	1 : 4 (20%)	2 : 18 (10%)
0 ms	40.48mW 100%	27.68mW 68%	21.52mW 53%	15.07mW 37%	9.92mW 25%	5.99mW 15%
20 ms	14.94mW 37%		10.20mW 25%	7.60mW 19%	5.28mW 15%	

Table 5.3: Energy consumption with different LPL-sleep times and duty-cycles

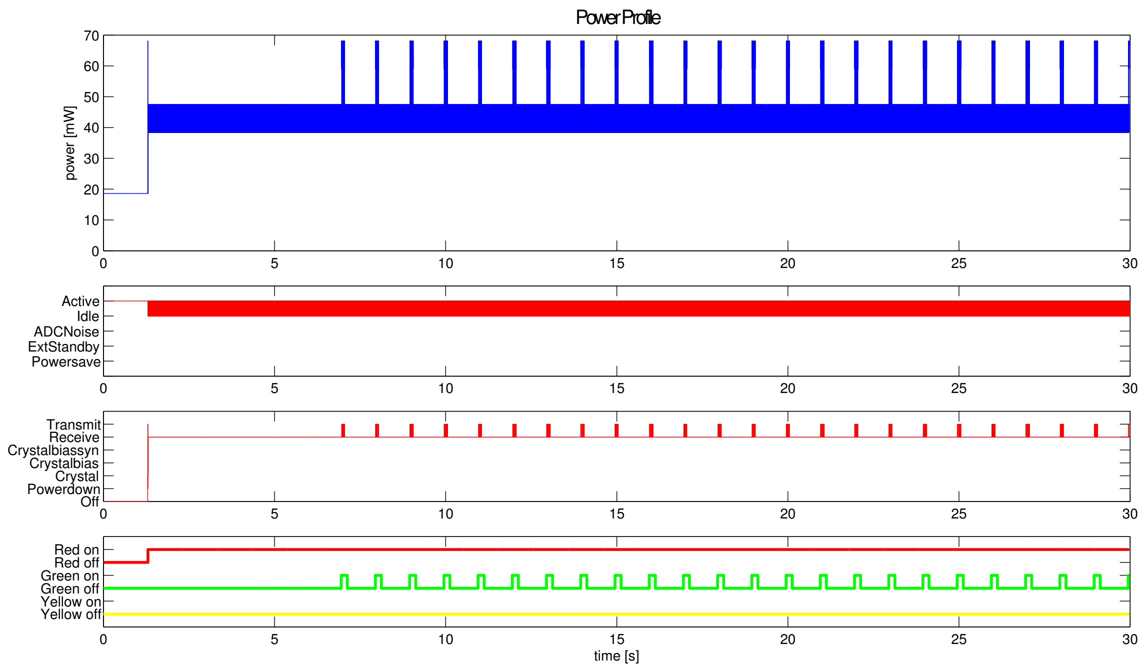


Figure 5.9: Trace of energy-states of the component on a mica2 node

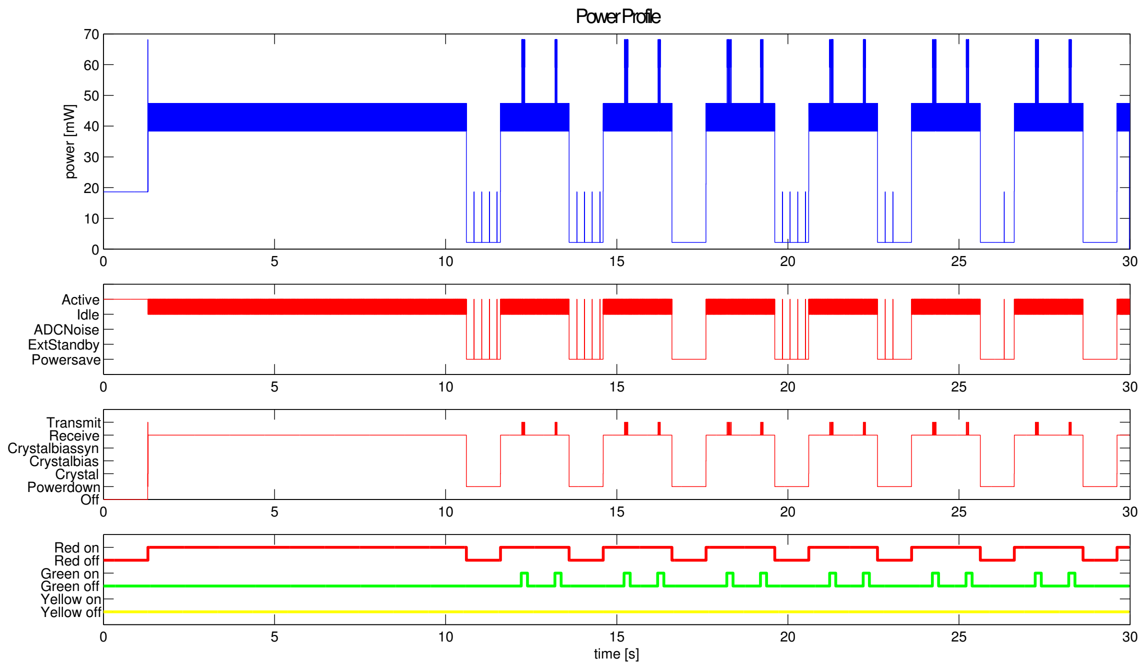


Figure 5.10: Trace of energy-states of the component on a mica2 node

Table 5.3 show a summary of the power-measurements. We see that using duty-cycling has the effect as thought. The power-consumption is about the percentage of the `ON-time`. When using LPL we see that even this amount of power-saving is doubled. But as we have seen in the previous part of the benchmarks, the LPL prevents the middleware to work. So the advantage of saving energy is only academic.
Table 5.3 shows only the power used by the middleware when running without an agent and is used as a measure for the energy drain of the middleware itsef. This includes all the routing (inclusive the sending of its messages), publishing the duty cycle and synchronising the motes.
Table 5.4 shows the power consumption of the middlware with the `SENS`-VO at the node. The gathering cycle of the agent is given by the duty cycle. We measured the energy drain with one, two and three data gathering cycle per duty cycle.

Data	duty-cycle					
per cycle	1 : 0 (100%)	2 : 1 (66%)	1 : 1 (50%)	1 : 2 (33%)	1 : 4 (20%)	2 : 18 (10%)
1	41.04mW	27.84mW	21.81mW	15.26mW	10.02mW	6.03mW
2	41.23mW	28.00mW	21.86mW	15.34mW	10.07mW	6.04mW
3	41.23mW	28.16mW	21.85mW	15.35mW	10.09mW	6.07mW
5	41.23mW	28.33mW	21.91mW	15.38mW	10.10mW	6.08mW
10	41.28mW	28.40mW	22.02mW	15.43mW	10.20mW	6.12mW

Table 5.4: Energy consumption with different duty-cycles and agents running

We also see in Table 5.4 that an agent doesn't need much power. In case of 100% duty cycle just 1.5% more or in absolute numbers $0.55mW$. This includes the acquisition of data and sending them to the BS.
Here we see that running an agent at a mote doesn't need much energy. More energy is wasted in the system messages like routing and publishing the duty cycle. Even if the mote samples up to 10 sensor samples per duty cycle. This produces up to 10 messages at the node. But this needs not much more energy. Part of the additional energy comes also from the multi threading system of TinyOS. This wakes the mote every 5ms to switch the threads.
From the results with 10 data gatherings per duty cycle we see that the TDMA mechanism doesn't work well with this much traffic. We observe that there only can be delivered 7 messages per time slot. If there come more messages they will be stored in the queue and have to wait to the next duty cycle. This will increase the delay and there can be messages be dropped if they take longer than two duty cycle to reach the BS.
The experiment with the duty cycle 2 : 1 shows that it will be better to have more time slots. This enables the networking system to send more messages per duty cycle and reduces the delay and the package drop.
In Figure 5.11 we see a state trace of a mote with an running agent. It is the experiment with a duty cycle 1 : 4 and 10 data gathering per duty cycle. While the Network is `ON` we don't see the acquisition of the data. During the `DOWN` time we see the peaks in the μC state. The thicker ones are from the data collection and the smaller ones are calculations in the middleware or thread switches. We also see that an agent doesn't need

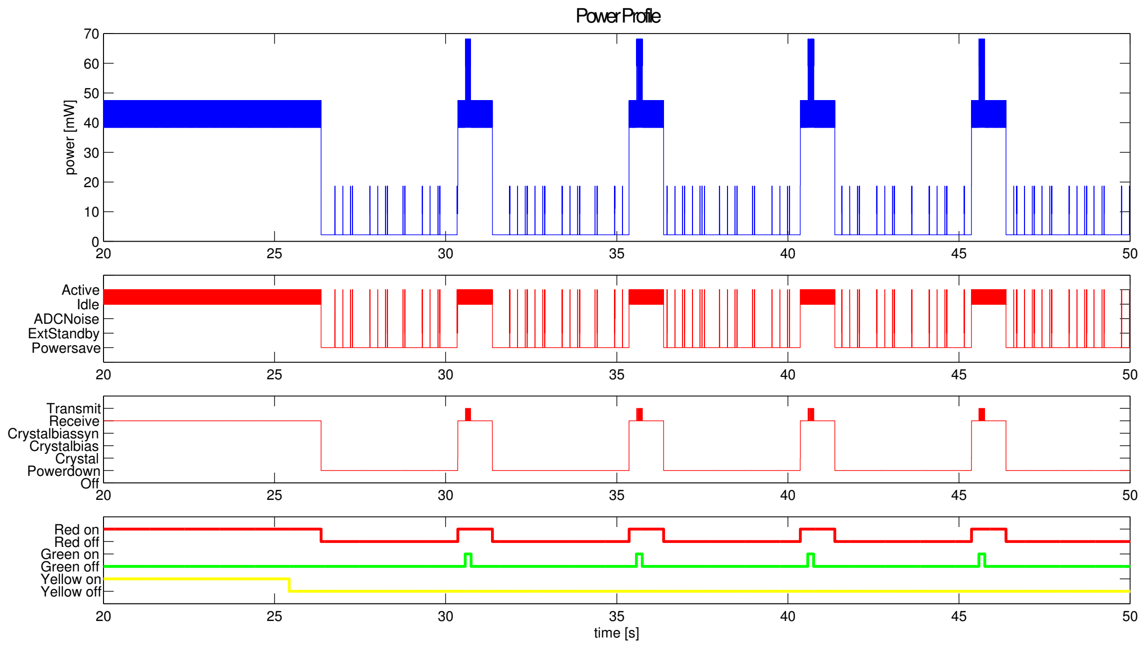


Figure 5.11: Trace of energy-states of the component on a mica2 node with the Sense agent running

much energy as also seen in Table 5.4.

Energy and delay

In Table 5.4 some of the cells are highlighted in grey. These are the tests where an agent samples its sensor each second. With this requirement from the user we can determine a trade off between energy and delay.
With a duty cycle of 1 : 0 the node needs about 41 mW and its messages have a maximum delay of $1*\mathbf{hops}$ seconds. At the second column we see the duty cycle 2 : 1. There we need about 28 mW power and get a maximum delay of $\lceil\frac{3*\mathbf{hops}}{2}\rceil$ seconds. For the duty cycle 1 : 1 the mote needs little less than 22 mW power and gets a maximum delay of $2*\mathbf{hops}$ seconds. Duty cycle 1 : 2 implies a power usage of about 15 mW and a maximum delay of $3*\mathbf{hops}$ seconds. For a duty cycle of 1 : 4 we got 10 mW power consumption and a delay of maximal $5*\mathbf{hops}$ seconds.

5.4.4 Workload balancing

The workload balancing is designed to grant that every node has the same workload. In the two test cases we will show, that this is possible, in a simple system.
In our test cases we will count the messages that we will receive from the nodes at the PC. Each message can be resent two times before it will be discarded. The Server will move an agent from the node if the count of messages between the node with the highest count and the node with the lowest count exceeds 20 messages. In the plots at Figure 5.12, 5.13 and 5.14 we plot at the y axis this count of messages. So the move command comes when the distance of the highest and the lowest line is more than 20. This distance is checked

by the `AgentSystem` class in the server side implementation every 100 ms.
In all tests we use the `Sense` agent with different reporting cycles. The middleware will send only one message per duty cycle (and its two resend if necessary).

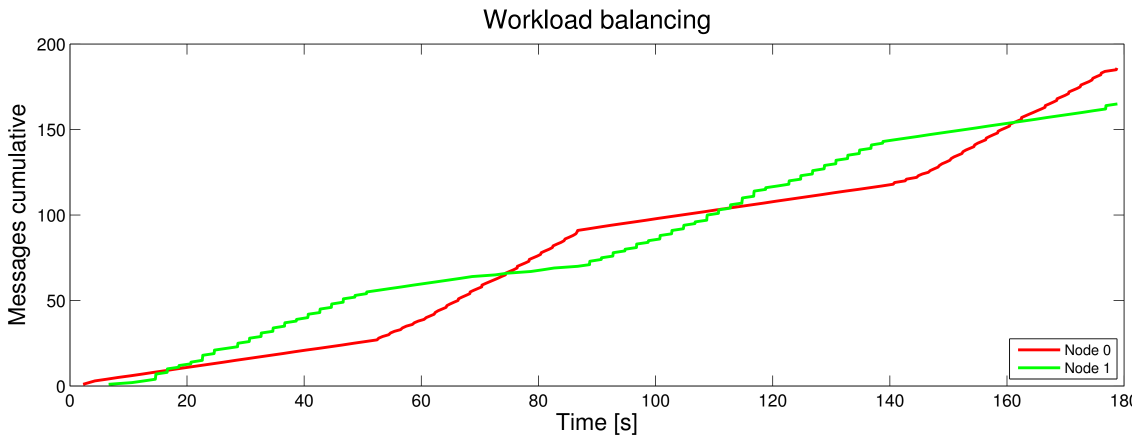


Figure 5.12: Time trace of the first test case

In Figure 5.12 we see the trace of the first test case. Here I used no duty cycle ($DC = 1 : 0$).
The agent we inject in the network will send every second a message to the BS.
At second 5 node 1 is synchronised and starts sending its energy information. A few seconds later (second 12) the agents is injected and starts working. This can be seen by the rising of the green line. From here the messages sent by node 1 are twice the messages from node 0. After about 20 seconds the threshold is reached and the workload balancing moves the agent from node 1 to node 0. As we see the red line starts rising and the green line keeps it level. The moving of the agents takes less than one duty cycle in this direction.
At second 75 the workload of the nodes is equal. At second 95 Node 0 has sent enough messages so that the threshold is reached and the agent is moved back to node 1 by the workload balancing. This move will take as the one before about one duty cycle. At 115 seconds the workload is again equal and 20 seconds later the threshold is reached a third time.
On average the workload of both nodes is the same as we can see in Figure 5.12.

Figure 5.13 shows the second test. Here we uses a duty cycle of 1 : 1. The agent at node 2 has 2 data gathering per duty cycle and the one at node 1 has 5 data gathering per duty cycle. So the node 0 sends 1 message, the node 1 6 messages and the node 2 3 messages per duty cycle.
We expect that the agent moves from node 1 to node 0 and then the agent moves from node 2 to node 1 and next the agent moves from node 0 to node 2 and so on.
In Figure 5.13 we see that node 0 and node 1 have behaviour like in the first test case. The agents from node 1 moves to node 0, which has no agent in the beginning, and than moves back.
But we see that the agent at node 2 never migrates to another mote. The workload of this node is always between the workload of node 0 and node 1. So this agent is never

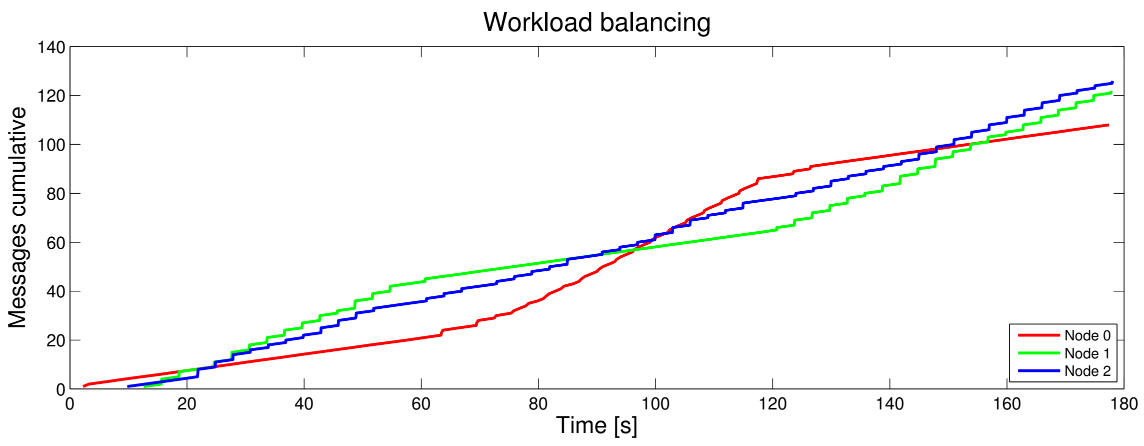


Figure 5.13: Time trace of the second test case

considered to get moved away from its node.

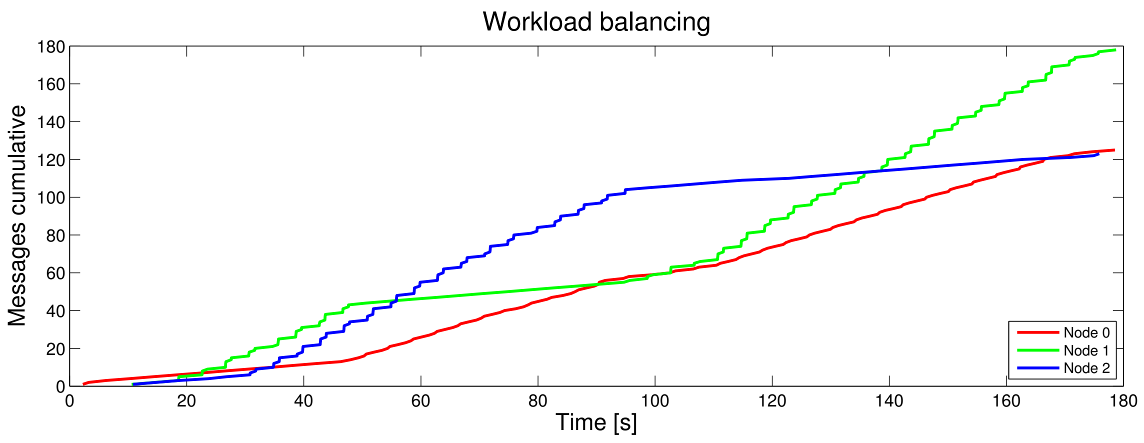


Figure 5.14: Time trace of the second test case

Figure 5.13 shows the desired behaviour of the workload balancing as discussed before. At second 20 is the agent injected to node 1. This agent is the slower one, with only 3 data gathering per duty cycle. At second 30 the server injects the agent to node 2. This one has 6 data gathering per duty cycle. Node 0 starts with no agents.
As we see in the figure, at second 45 the threshold exceeds the limit of 20 messages between the message count of node 1 and node 0. From here the node 1 has no agents and sends only the energy report back to the BS. Node 0 now reports 3 times per duty cycle the temperature. At second 95 the node 0 and node 1 have sent the similar amount of messages. Also at this time the limit between node 1 and node 2 exceeds the threshold. Now the fast agent migrates from node 2 to node 1. Now node 2 only reports its energy back to the BS and node 1 reports 6 times per duty cycle the temperature. At the end of this plot the node 0 and node 2 have equal workload. The node 1's workload exceeds the limit to the workload of the other nodes. So it would be time to move the agent back to

node 2.

5.5 Summary

In Section 5.1 I present the impact of the changes at the operating system TinyOS. There I show how a little change in the system will help to save up to 30% of the energy.
In the Sections 5.2 and 5.3 we discuss the functionality of the middleware and how I have tested it. In Section 5.2 I discuss the agents and the VM. Here I present the results of the module tests for the VM and the agents.
In Section 5.3 I discuss the power management. In short words I discuss the time synchronisation and the duty cycling.
The Section 5.4 brings the measurements. A quick review of the environment starts this Section. It continues with the benchmarks. Here we discuss the delay of the middleware and the failure with the LPL. The energy measurements show that the implementation brings a good opportunity to save much energy. At the end of this Section I bring up the relation of energy conservation and the delay.
In the last part of this chapter I present the results of the workload balancing and how it works.

Chapter 6

Conclusion and future work

In this chapter I will give a résumé of the results from the previous chapter and combine it with the ideas from Chapter 2. Also I will give a short outlook to future work and extensions for the presented middleware.

6.1 Conclusion

This section give a short summary over the thesis and gives in the last part a review of the completeness of the implementation of the middleware.
We will start with the agents, go further to the network implementation and finish with the power console and the workload balancing.

6.1.1 Agents

Agents are a good and flexible extension of the TinyOS framework. These little programs make it possible to change the behaviour of the nodes in the network in runtime.
The mobility of the agents provides a flexible system to deploy tasks to motes in the field. This system provides also a good system to spread agents over the network. We had only to deploy the agents to one node and then clone it to its neighbours.

6.1.2 Networking

And so we come to the implementation of the networking modules. Here we have to discuss three themes. First comes MAC protocol and its low power duty cycling.

The B-MAC [28] works very good for programs written in TinyOS, which will not very often send messages. But in this middleware with its good time synchronisation we encounters very often the *hidden node problem*. Two motes try to send at the same time via the radio channel. Both can't sens the message from the other mote but both want to send to the same destination they can reach. When this occurs the two messages collide at the destination and both are unreadable. This is the occasion why I have implemented the TDMA with its small time slots.
Another thing to discuss is the LPL. While testing the networking module the LPL prevents the motes to receive any message. Even when I have implemented the time slotting

only the system messages, like `TimeSync` messages, `Power` messages or `Routing` messages, are received at the destinations. Only very few data messages even when not injecting any agents reached the BS. But the motes get synchronised. Further investigation of this matter will help this middleware to save more energy.

Second I will discuss the TDMA mechanism as a arbiter for the network. The static assignment of the time slots is very good for testing purposes but not practical in real world environments. The focus here should be looking for better arbiter to access the resource *radio channel*. Maybe this can be done with other MAC-protocols.

Third we will have a short look at the routing. Here I have done nearly nothing. Power and energy aware routing protocols are big points in the scientific community that works with WSNs. Impact of this protocols to the long-term energy conservation should be one of the next points on the list of research.

6.1.3 Power console and workload balancing

In the previous chapter we have seen the power consumption of the motes with and without agents. We see that agents which samples 10 times per second the a sensor only draws about $0.80mW$ in average over several seconds.
When using duty cycling we can reduce this drain down to $0.13mW$. This happens at a duty cycle of 10% and sampling every 2 seconds.

We see that the agents draw very little energy. Most of the used energy is consumed by the middleware itself. But this energy can be reduced with the duty cycle. In Table 5.3 we see that the power consumption of middleware is about the same percentage as the duty cycle. This is as we have it expected.

The workload balancing works much better as expected. It spreads the workload equally over the network as we see in the results. This also works even with this simple algorithm I use in the test cases. Here maybe a distributed mechanism could be implemented in middleware to determine where and when an agent has to move.

6.1.4 Completeness of the implementation

This part speaks about the modules and services i discussed in the concept but they are not implemented in the middleware.

First of all is the problem with the LPL. As mentioned before this has to be investigated and maybe there will be a solution in the future.

Second I spoke in the concept about some adaptive algorithm to control the duty cycle. The API is just implemented to have something like that done but it is not programed. So there has to be done some study of literature and research. Than the middleware can have a workload aware adaptive duty cycling.

Third I have mentioned an adaptive workload balancing. A simple static method to distribute the workload over all the nodes is implemented. But with a little research someone will find some better mechanism to spread the workload.
At the end I will bring a small hint to my successor, who will work with this middleware. Workload balancing can be thought of a kind of topology controlled duty cycling.

Also some long term measurements I want to take. But through the lack of time and equipment that has been cancelled from schedule.

6.2 Future work

In this last section I will speak of things that are missing in the middleware and not mentioned in the concept. Also I will bring some topics of future research for the middleware.

Something that is never spoken of in this work is security. Before this system can be implemented to some real life assignments, there should be a mechanism to make sure that:

- only agents will be injected that are authorised,
- only VOs will have access to the network that are authorised and
- agents that are injected into the network are not overwriting some other agent.

Other security issues have to be implemented as well, like encrypting data, authorising for new agents and nodes and so on.

Other thing should be the implementation of other access methods to the network instead of TDMA. Some other systems are described in concept. Also can the usage of other MAC protocols, like X-MAC [9],S-MAC [32] or T-MAC [30], be a good idea to increase the utilisation of the radio channel. Also integration of some power- or energy-aware routing protocols can bring a improvement of the power consumption.

For the agents a mechanism of version control will be a nice feature. So the application developer can update all his agents without losing their memory. Also a better dissemination protocol for larger agents can be a step forward do a widely use of this middleware.
Other computational models for agents or better memory management will improve their maintainability.

Appendix A

Abbreviations

ADC	analog-digital converter
AM	active message
BS	basestation
EHS	energy-harvesting system
FIFO	first in - first out
FTSP	flooding time synchronisation protocol
KISS	keep it small and simple
LPL	low-power listening
MAC	medium-access control
MAMA	multi-application middleware
OS	operating system
QoS	quality of service
SPI	serial-peripheral interface
TDMA	time-division-multiple access
TOSPIE2	tiny operating system plug-in with energy estimation
USART	universal synchronous/asynchronous receiver transmitter
VO	virtual organisation
VM	virtual machine
WSN	wireless sensor network

Appendix B

Code

```
enum{
    HALT = 0,  // Ends the Program
    JMP,       // Jump A byte back
    INKR,      // Inkrement VarA
    SETLED,    // display VarA to the leds
    SLEEP,     // Sleep for A ms
    SEND,      // Sends VarA to VarB
    RECV,      // Receives VarA
    READ,      // Reads Sensor A to VarB
    ADD,       // Var A = Var B + Var C
    // Instruction with accumulator
    LOAD,      // Loads VarA to Accumulator
    STORE,     // Stores the Accumulator to Var A
    INKRA,     // Increments the Accumulator
    SETLEDA,   // Displays the Accumulator to the leds
    SENDA,     // Sends the Accumulator to Var A
    RECVA,     // Stores the received value in the Accumulator
    READA,     // Reads Sensor A to the Accumulator
    ADDA,      // Accu = Accu + Var A
};
```

Sourcecode B.1: Instructionset of the agent

```
  while(1){
    switch (ag->prog[mem.pc++]) {
     case HALT:
       return;
       break;
     case JMP:
       mem.pc = ag->prog[mem.pc];
       break;
     case SLEEP:
       call System.sleep(*memory(ag->prog[mem.pc]));
       mem.pc+=1;
       break;
     case SETLED:
       call Leds.set(*memory(ag->prog[mem.pc]));
       mem.pc+=1;
       break;
     case INKR:
       (*memory(ag->prog[mem.pc]))++;
       mem.pc+=1;
       break;
     case SEND:{
       data.value = (*memory(ag->prog[mem.pc]));
       error = call
           MiddlewareRadioSend.send(*memory(ag->prog[mem.pc+1]), 0,
           &data, sizeof(data_t));
       mem.pc += 2; // go two bytes;
       }break;
     case RECV:
       if(call MiddlewareRadioReceive.receive(&msg, 0)==SUCCESS)
         *memory(ag->prog[mem.pc]) = ((data_t*)call
             RadioPacket.getPayload(&msg))->value;
       mem.pc+=1;
       break;
     case READ:{
       uint16_t val;
       if(call SensorRead.read(ag->prog[mem.pc], &val)==SUCCESS)
         *memory(ag->prog[mem.pc+1]) = val;
       mem.pc += 2;
       }break;
     case ADD:{
       (*memory(ag->prog[mem.pc])) = (*memory(ag->prog[mem.pc+1]))
           + (*memory(ag->prog[mem.pc+2]));
       mem.pc += 3;
     }break;
```

Sourcecode B.2: The agent-virtual machine with register

```
        case LOAD:
          mem.accu = (*memory(ag->prog[mem.pc]));
          mem.pc += 1;
          break;
        case STORE:
          (*memory(ag->prog[mem.pc])) = mem.accu;
          mem.pc += 1;
          break;
        case SETLEDA:
          call Leds.set(mem.accu);
          break;
        case INKRA:
          mem.accu++;
          break;
        case SENDA:{
          error_t error;
          data.value = mem.accu;
          error=call MiddlewareRadioSend.send(*memory(ag->prog[mem.pc]), 0, &data, sizeof(data_t));
          mem.pc += 1;
          }break;
        case RECVA:
          if(call MiddlewareRadioReceive.receive(&msg, 0)==SUCCESS)
            mem.accu = ((data_t*)call RadioPacket.getPayload(&msg))->value;
          break;
        case READA:{
          uint16_t val;
          if(call SensorRead.read(ag->prog[mem.pc], &val)==SUCCESS)
            mem.accu = val;
          mem.pc += 1;
          }break;
        case ADDA:
          mem.accu += (*memory(ag->prog[mem.pc]));
          mem.pc += 1;
          break;
        default:
          return;
          break;
      }
    }
```

Sourcecode B.3: The agent-virtual machine with accumulator

Appendix C

Pictures

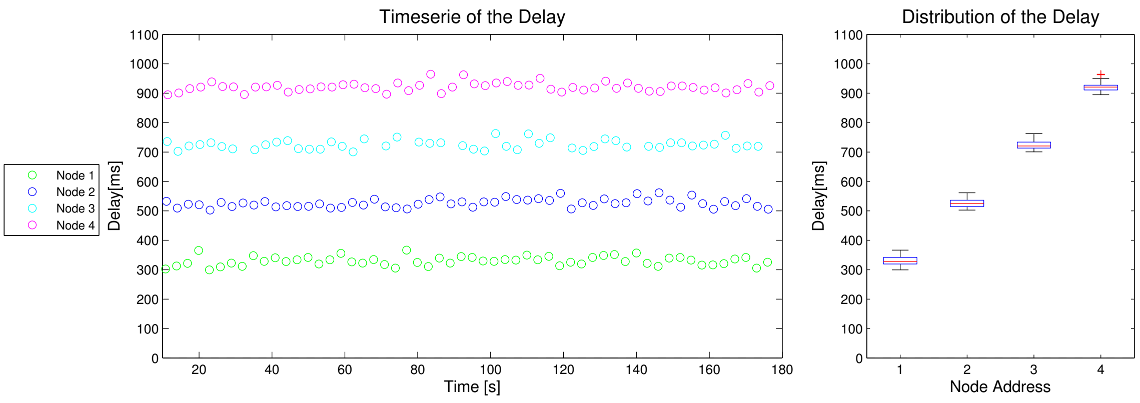


Figure C.1: Delay of the mote grid with 4 nodes (DC = 2:1)

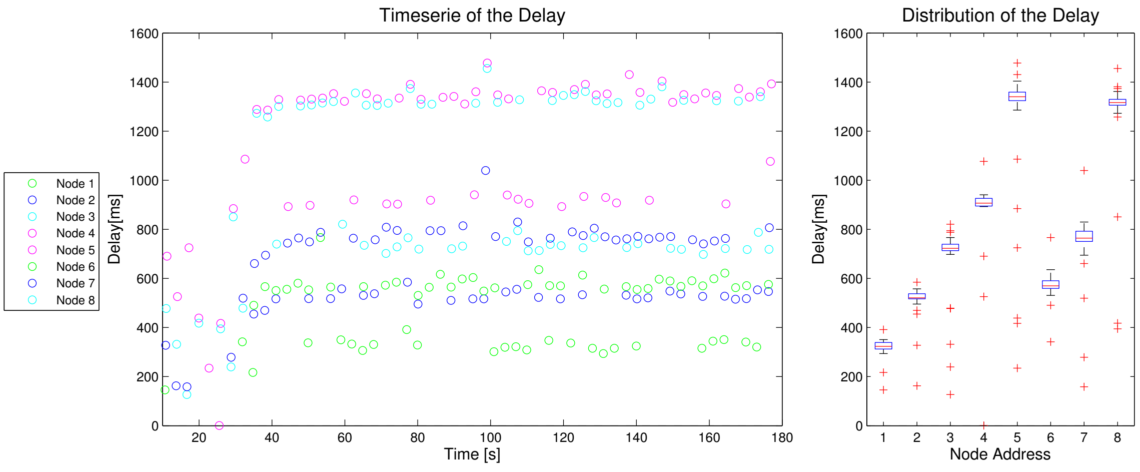


Figure C.2: Delay of the mote grid with 8 nodes (DC = 2:1)

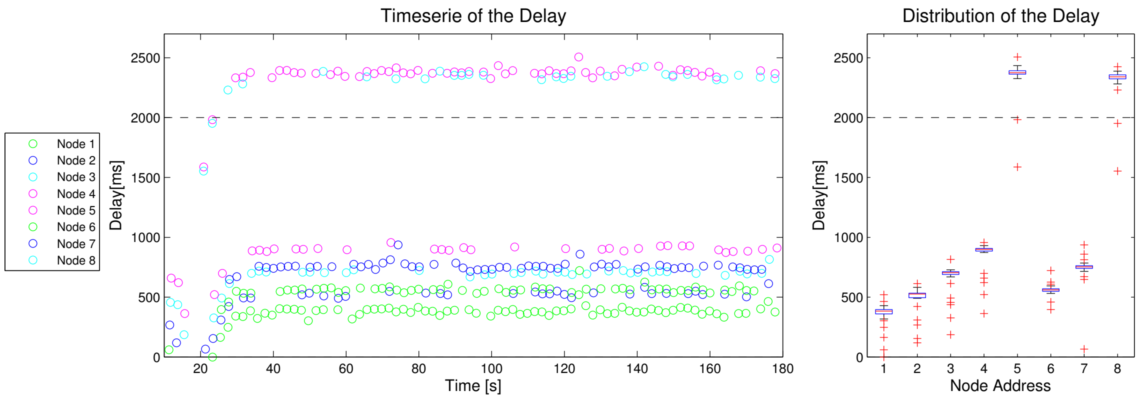


Figure C.3: Delay of the mote grid with 8 nodes (DC = 1:1)

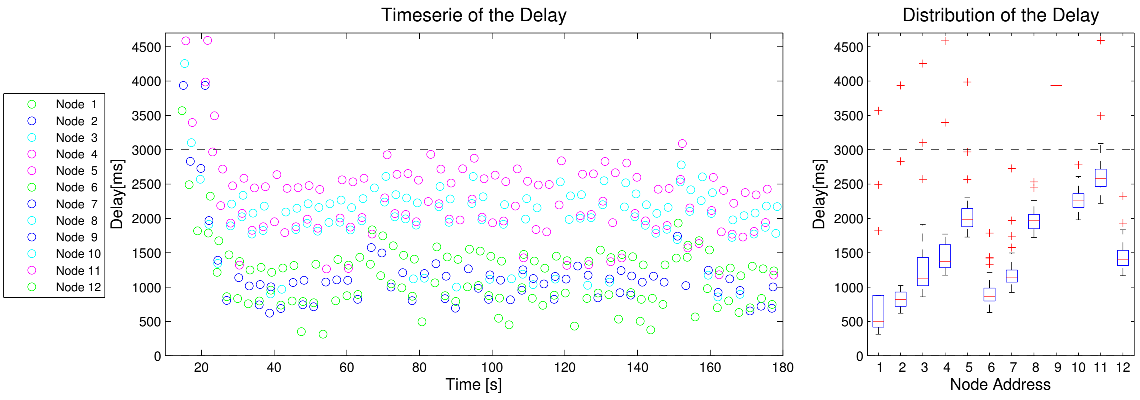


Figure C.4: Delay of the mote grid with 12 nodes (DC = 2:1)

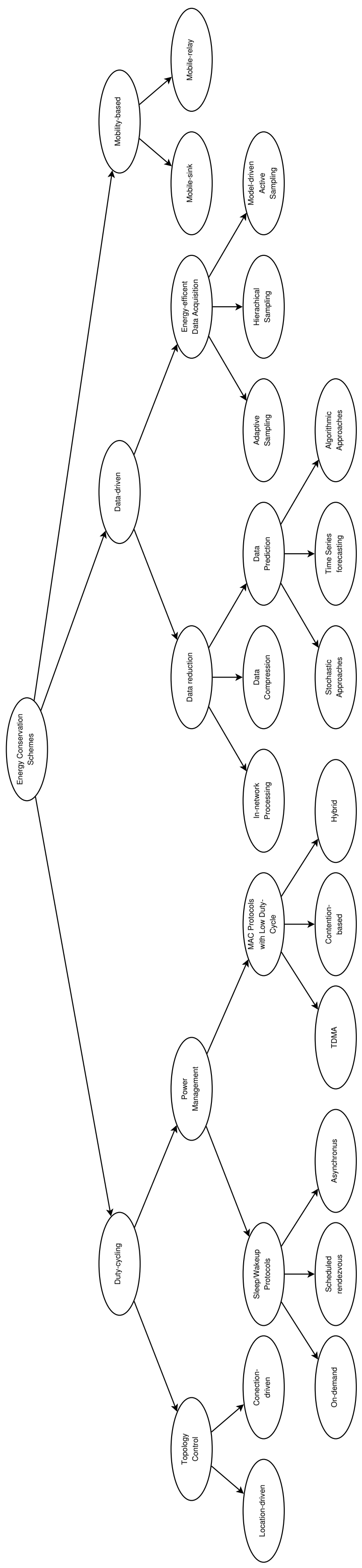


Figure C.5: Taxonomy of energy conservation [5]

Bibliography

[1] What is the grid? a three point checklist. *Grid Today*, 1(6):22–25, 2002.

[2] Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications, June 2010.

[3] I. F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. 40:102–114, Aug. 2002.

[4] TinyOS Alliance. Tinyos extension proposal. online at http://www.tinyos.net/tinyos-2.x/doc/.

[5] Giuseppe Anastasi, Marco Conti, Mario Di Francesco, and Andrea Passarella. Energy conservation in wireless sensor networks: A survey. *Ad Hoc Netw.*, 7(3):537–568, 2009.

[6] ATMEL. *ATmega 128, 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash*, June 2007.

[7] Reinhard Berlach. Multiapplication middleware for wireless sensor networks. Project thesis. Institut für Technische Informatik, TU-Graz, May 2010.

[8] Douglas M. Blough and Paolo Santi. Investigating upper bounds on network lifetime extension for cell-based energy conservation techniques in stationary ad hoc networks. In *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 183–192, New York, NY, USA, 2002. ACM.

[9] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 307–320, New York, NY, USA, 2006. ACM.

[10] Philip Buonadonna, Jason Hill, and David Culler. Active message communication for tiny networked sensors, 2001.

[11] Yunxia Chen and Qing Zhao. On the lifetime of wireless sensor networks. *Communications Letters, IEEE*, 9(11):976 – 978, nov. 2005.

[12] Chipcon Products from Texas Instruments Inc. *CC1000 Single Chip Very Low Power RF Transceiver*, January 2009.

[13] N. Costa, A. Pereira, and C. Serodio. Virtual machines applied to wsn's: The state-of-the-art and classification. In *Proc. Second Int. Conf. Systems and Networks Communications ICSNC 2007*, 2007.

[14] Crossbow Technology, Inc., San Jose, CA 95134. *MTS/MDA Sensor Board Users Manual*, June 2007.

[15] Isabel Dietrich and Falko Dressler. On the lifetime of wireless sensor networks. *ACM Trans. Sen. Netw.*, 5(1):1–39, 2009.

[16] I. Foster. The anatomy of the grid: enabling scalable virtual organizations. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 6 –7, 2001.

[17] Philipp M. Glatz, Leander B. Hörmann, Christian Steger, and Reinhold Weiss. MAMA: Multi-Application MiddlewAre for Efficient Wireless Sensor Networks. In *Proceedings of the 18th International Conference on Telecommunications*, pages 1–8, Ayia Napa, Cyprus, 2011.

[18] Philipp M. Glatz, Philipp Meyer, Alex Janek, Thomas Trathnigg, Christian Steger, and Reinhold Weiss. A measurement platform for energy harvesting and software characterization in WSNs. In *IFIP/IEEE WD*, 2008.

[19] Philipp M. Glatz, Christian Steger, and Reinhold Weiss. Tospie2: tiny operating system plug-in for energy estimation. In *IPSN '10: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 410–411, New York, NY, USA, 2010. ACM.

[20] G. P. Halkes, T. van Dam, and K. G. Langendoen. Comparing energy-saving mac protocols for wireless sensor networks. *Mob. Netw. Appl.*, 10(5):783–791, 2005.

[21] Abtin Keshavarzian, Huang Lee, and Lakshmi Venkatraman. Wakeup scheduling in wireless sensor networks. In *MobiHoc '06: Proceedings of the 7th ACM international symposium on Mobile ad hoc networking and computing*, pages 322–333, New York, NY, USA, 2006. ACM.

[22] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140, New York, NY, USA, 2009. ACM.

[23] Mauri Kuorilehto, Marko Hännikäinen, and Timo D. Hämäläinen. A survey of application distribution in wireless sensor networks. *EURASIP J. Wirel. Commun. Netw.*, 2005:774–788, October 2005.

[24] Ji Woong Lee and Jean Walrand. Zerocollision random backoff algorithm. Technical Report UCB/EECS-2007-63, EECS Department, University of California, Berkeley, May 2007.

[25] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004.

[26] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 39–49, New York, NY, USA, 2004. ACM.

[27] Razavan Musaloiu-E., Chieh-Jan Mike Liang, and Andreas Terzis. A modular approach for wsn applications. Technical report, Johns Hopkins University, Sept. 2008.

[28] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM.

[29] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[30] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 171–180, New York, NY, USA, 2003. ACM.

[31] Mohamed K. Watfa and Mohamed Moubarak. Building performance measurement tools for wireless sensor network operating systems. In *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '09, pages 599–604, New York, NY, USA, 2009. ACM.

[32] Wei Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1567 – 1576 vol.3, 2002.

[33] Youtao Zhang, Jun Yang, and Weijia Li. Towards energy-efficient code dissemination in wireless sensor networks. pages 1 –5, apr. 2008.