Delft University of Technology
Master's Thesis in Embedded Systems

Run-Time Reconfiguration in Wireless Sensor Networks

Robin van den Berg

TNO innovation for life

embedded software

TU Delft Delft University of Technology

Run-Time Reconfiguration in Wireless Sensor Networks

THESIS

submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE
in
EMBEDDED SYSTEMS

by
Robin van den Berg

TNO innovation for life
Department of Physics and Electronics
Expertise center Technical Sciences
TNO
Stieltjesweg 1, 2628 CK Delft, The Netherlands

embedded software
Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Run-Time Reconfiguration in Wireless Sensor Networks

Author
Robin van den Berg (robin@vandenbergweb.nl)
Report submission date
15 July, 2011
MSc presentation
22 August, 2011

Abstract

A wireless sensor network (WSN) consists of multiple small and simple computers (nodes), whose performance is tightly linked to its unpredictable deployment environment. It is nearly impossible to design a WSN that performs well in every scenario; instead they are developed for a specific context, with performance rapidly decreasing when environment properties move away from the optimum. Enabling a WSN to adapt to a changing environment could be a solution for this problem. The goal of this thesis project is to develop software for sensor nodes that is able to reconfigure a wireless sensor network at run-time, allowing the sensor network to perform within its requirements under changing conditions.

An approach is proposed in which developers provide knowledge about the WSN's requirements to sensor nodes during design-time, enabling the nodes to reason about their configuration at run-time. A middleware solution is designed that combines the run-time environment data with the knowledge about the requirements and indicates which reconfiguration is required. A proof-of-concept implementation is developed which reasons about reconfiguration within milliseconds. The performed evaluation includes the implementation of a reconfigurable modal analysis application and shows the processing and memory overhead introduced by the middleware remains low. The approach proves to be a solid basis for future developments on reconfigurable sensor networks.

Graduation Committee

Prof.dr. K.G. Langendoen (chair)	Delft University of Technology
Dr.ir. G.N. Gaydadjiev	Delft University of Technology
Dr.ir. Z. Papp (daily supervisor)	TNO
Dr. M. Woehrle (university supervisor)	Delft University of Technology

Preface

Although one never stops learning, the time has finally come for me to finish my education at the TU Delft. At the start of my master programme, I was not particularly interested in the topic of Wireless Sensor Networks. However, as I learned more about Embedded Computer Architectures, Real-Time Systems and Distributed Algorithms and grew interested in how computers can interact with the physical world, I realized that Wireless Sensor Networks combined many aspects of these specializations. As a person who is not very fond of choosing between his different interests, this research area was ideal for my thesis project.

As I would liked to have more experience in industry, I applied for a thesis project with TNO. From their experience with WSN deployments, TNO had learned that run-time changes affect the usability of WSNs. They offered me the opportunity to work on this problem, which I eagerly accepted.

I could not have reached the end of this project without the support of the people around me. First of all, I would like to thank my supervisors at the TU Delft and TNO, Matthias and Zoltan. Although initially I was very shy and reluctant to ask for help, they offered me guidance and tips and convinced me to be less hesitant with asking questions. Furthermore, thanks go out to my colleagues at TNO, especially Arjan and Puneeth, for their suggestions, resources and of course the cookies and fun discussions during the coffee breaks. Last but definitely not least, I would like to express a great deal of gratitude to my parents and to my girlfriend Chanine, who have supported me through the many tough times that came with this project. You mean more to me than I can express.

Delft, The Netherlands
July 15, 2011

Contents

CHAPTER 1

Introduction

Ever since the introduction of automated computing, computer components have decreased in size. Although Moore's law is often explained as an exponential growth in the speed of microprocessors, it also translates into chips with similar performance becoming smaller, cheaper and more power efficient. This phenomenon has gone side-by-side with a decrease in the number of operators per computer: it has in fact now become an increase in the number of computing devices per operator. The last decade, this evolution has resulted in the concept of "smart dust" [18] and the emergence of wireless sensor networks (WSNs).

A WSN consists of multiple small and simple computers called (sensor) nodes. A node contains computing power (a microcontroller or *mcu*), some (wireless) communication capability and one or more sensors. It is typically powered by an off-grid power supply, almost always a battery. This enables them to function in a completely untethered manner, but also requires the nodes to be energy-efficient, since off-grid power supplies generally have a very low energy density.

The size and freedom of the nodes allow them to be deployed in an ad-hoc manner, possibly in hard-to-reach or hostile environments. Typical use-cases for WSNs are environmental monitoring and event detection applications. After deployment, the nodes co-operate to form a network and exchange data to reach a common goal. A gateway computer on the edge of the network, usually referred to as *base station*, can be used to exchange information with the sensor network. A typical WSN setup is shown in Figure 1.1.

The operation of a WSN is tightly linked to its unpredictable deployment environment. Obviously, the environment determines sensor readings on the nodes, but it can also affect other, interlinked properties like communication delay, energy consumption and network topology. The properties of the deployment environment that in some way affect the WSN will be referred to as the *context*. The context for WSNs can vary considerably and some situations can be completely unknown to the developer. It is nearly impossible to design a system that performs well in every case; WSNs are often targeted for a specific context. In this case, performance often rapidly decreases when the differences between the actual and targeted context

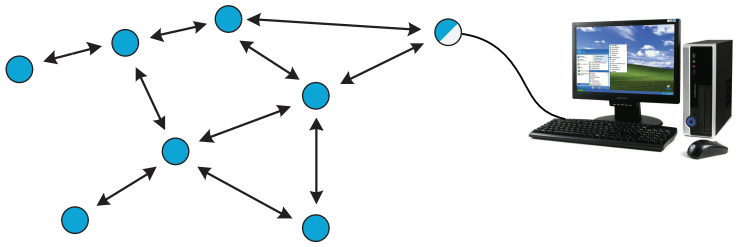


Figure 1.1: An example of a Wireless Sensor Network. Arrows represent communication links, the computer on the right acts as a gateway for network operators.

increase. Adapting a WSN to a changing context is one solution for this problem, providing an acceptable quality of service in a wider range of contexts instead of a high performance in a specific case.

Aside from the deployment context, the use of simple (inexpensive) hardware, wireless transmissions and ad-hoc networking make it very difficult to create dependable WSNs. Next to fault prevention and fault tolerance, fault removal and forecasting are means to ensure a system will meet its specifications during its lifetime [2]. This can require reconfiguring (sub)components if failure of the current system with current or future contexts is anticipated.

Reconfiguration means adapting (sub)components or their arrangement within a system. Although it can be performed by stopping a system, applying changes and starting the new system, in the context of this thesis it implies adapting the WSN or its components at run-time. Reconfiguration requires handles with which (sub)components can be changed and one or more alternatives to change them with. Also, intelligence is required to reason about which alternative to choose, based on performance metrics and system specifications.

1.1 Commercial Interest

TNO is a research company with several offices located primarily in The Netherlands. Its mission statement is *"TNO connects people and knowledge to create innovations that boost the sustainable competitive strength of industry and well-being of society."* The Distributed Sensor Systems department of the TNO expertise center Technical Sciences has an interest in using WSNs for several studies on acoustics, structural monitoring and smart mobility platforms. These studies are mostly conducted by domain experts, for whom developing WSNs is very tedious work. TNO is looking for solutions that can simplify the development of dependable WSNs for their research. A particular approach they are interested in is reconfigurable WSNs, which would allow them to use a single WSN for different research projects.

1.2 Problem Statement

The goal of this thesis project is to develop software for sensor nodes that is able to reconfigure a wireless sensor network at run-time, allowing the sensor network to perform within its requirements under changing conditions. Since TNO is interested in using WSNs for different types of studies, the solution should provide an architecture that can be used for a wide range of sensor network applications. The availability of a generic reconfiguration architecture should simplify the development of dependable WSNs.

TNO has provided a scenario and platform that can be used as an example throughout the development phases. In this application, the structural health of wind turbines is monitored by measuring and analyzing the vibration modes. A network of G-Node sensor nodes [36] should measure the acceleration at different locations on the pillar of a turbine and provide reliable access to this data. This WSN should be able to perform under varying circumstances, in which individual nodes crash, communication links fail, energy depletes or sensor measurements are erroneous.

In this thesis, a middleware solution is proposed that allows a developer to involve run-time context information in the (re)configuration of WSN components. A structure is proposed for incorporating the required knowledge about the application into the middleware. The middleware design is implemented as a proof-of-concept on a typical WSN platform. For assessing the approach of the reconfiguration middleware, a WSN application for structural health monitoring has been developed. Reconfiguration scenarios have been defined for this case study and incorporated into the middleware. The advantages and disadvantages of the middleware for application performance and dependability are tested with these scenarios. The proposed approach resulted in a reconfigurable WSN application and provides a basis for future run-time reconfigurable networks.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides additional background information on wireless sensor networks, middleware and reconfiguration. Rule-based middleware is discussed in Chapter 3, including the design of such a solution for reconfigurable WSNs. Details about a proof-of-concept implementation for a common WSN platform are provided in Chapter 4. Using a case study, the proposed middleware is evaluated in Chapter 5. Conclusively, Chapter 6 summarizes the work described in this thesis, provides a discussion of the approach and proposes suggestions for future work.

CHAPTER 2

Background

This chapter provides background information on WSNs in general and elaborates on related work about reconfiguration and rule-based intelligence applied to WSNs. It shows that many problems of WSNs are related to the context, while there are only few efforts that use the context to provide a scalable solution for reconfiguring WSNs at run-time.

Section 2.1 focuses on constraints and requirements that come with designing and developing WSNs. Section 2.2 provides more details about the development process of WSNs and how it can benefit from run-time reconfiguration. In Section 2.3, current middleware solutions for WSNs are discussed. Finally, Section 2.4 covers approaches for rule-based intelligence in WSNs.

2.1 Wireless Sensor Networks

The typical deployment scenario for WSNs requires cheap nodes with a small form factor that can operate untethered for a long period. To achieve this, the hardware is comprised of commodity off-the-shelf components with limited computation and storage capacity. The G-Node used in TNO's testbed is equipped with a TI MSP430 16-bit microcontroller operating maximally at 16 MHz, has 8 KB of RAM, 116 KB of ROM, and 8 Mbit of storage capacity [36]. Operations on floating-point numbers are not supported natively. To save energy, the microcontroller supports five low-power modes aside from its fully active mode [17].

The information coming from the WSN is generated by sensors on board the nodes. Using microelectromechanical systems (MEMS) and analog-to-digital converters, values of various physical phenomena can be measured and converted to a binary representation. The sensitivity, accuracy and sampling rate can all be configured by the user. To avoid excessive communication, sensor data is often processed locally before being made available to the rest of the network. For exchanging information, nodes are equipped with a wireless radio. The transmission range is limited and varies unpredictably over time and space, because of physical phenomena like signal reflection, diffraction, shadowing and background noise. This means links

cannot be assumed to be reliable and the network topology can change at run-time. Also, the wireless communication uses a broadcast medium, which means the example network of Figure 1.1 is more accurately modeled by Figure 2.1. The figure shows that transmission ranges overlap, which means simultaneous transmissions can cause collisions at the addressee without the senders knowing it, and disrupt otherwise unrelated communication between other nodes.

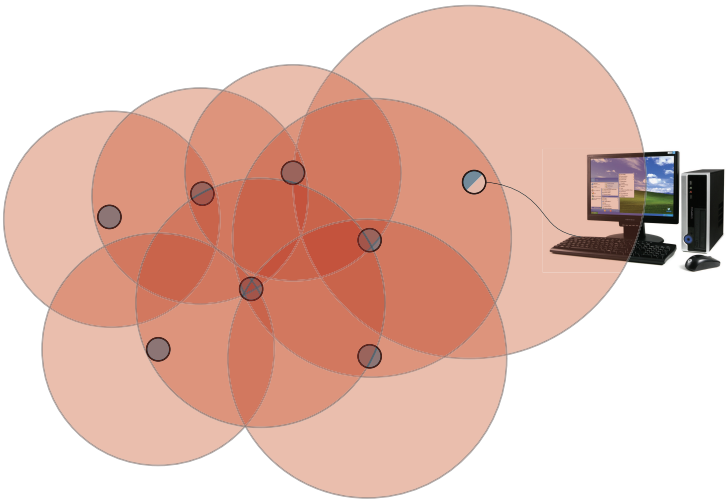
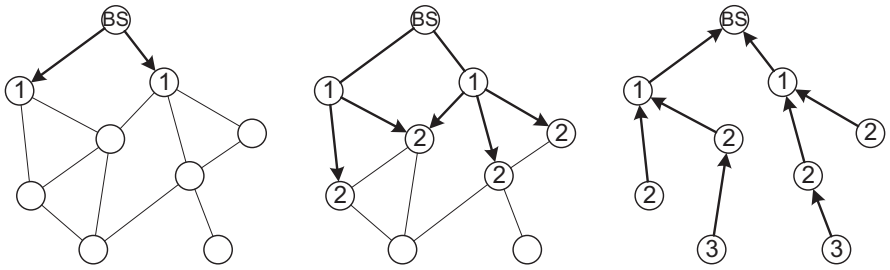


Figure 2.1: The sensor network of Figure 1.1, drawn with idealized wireless transmission ranges.

In reality, the ranges are far more irregular than with this idealized disc-model, making it even more difficult to predict network connectivity [15]. Data from a node also possibly has to travel multiple hops before it reaches its target destination. However, predefining a network structure is typically not possible, which means specialized routing protocols are required that cope with this unpredictability. As an example of setting up data routes after deployment, Figure 2.2 shows the Collection Tree Protocol that can be used to collect sensor data at one point in the network [14]. The nodes set up the collection tree after deployment and use some link quality indicator to determine to which parent they should send their information.

To make a WSN function as long as possible often requires reducing the energy consumption to a minimum. Table 2.1 shows a comparison of the current draw – which is proportional to the energy consumption – for different activities of the G-Node and often-used alternative WSN platforms. Since the radio is the main power consumer for many sensor node platforms, it is put into a low-power (sleep) mode whenever possible. Whenever communication is required, the radio can be turned on briefly, after which it is put back into the sleep mode. A duty-cycle like this can also be used for other components, e.g. sensors with high energy consumption. Because duty-cycling constrains the use of available functionality, the various requirements of the system should be considered when determining the cycle time. Finding the



(a) The base station sends a beacon message.

(b) Its direct neighbors mark the sender as their parent and rebroadcast.

(c) When every node has a parent, data can be sent to the tree root.

Figure 2.2: Formation of a collection tree.

Component		Current draw (mA)		
		Sownet G-Node [36]	Moteiv Tmote [5]	Crossbow Mica2 [37]
Radio	reception	14.6	23	27
	transmission	34.2	21	10
Mcu	active	1	2.4	8
	low power	0.0012	0.021	0.015

Table 2.1: Comparison of G-node components' current draw.

optimum has been a subject of various research efforts [30, 39].

2.2 WSN Development Approach

To design a WSN application, knowledge of many elements of the context is essential as they influence the operation greatly. However, because some aspects of the context are unpredictable and changing, many design choices are based on assumptions and approximations.

As an example, consider the case study of a sensor network for structural health monitoring. During the development of such a WSN, it is unknown which influences the nodes might experience, which nodes might crash, and how long exactly the sensor nodes will last with the available energy. Because of these uncertainties, the quality of service of the WSN might develop over time like the solid blue line in Figure 2.3.

Initially, the WSN will perform well because the context matches the expectations during development. However, at some point t_1 after deployment, a sensor node might have changed position because of some external event, making its sensor readings less accurate and the WSN decrease its performance. Also, another node might experience high interference, resulting in many retransmissions from that node. The higher energy consumption can cause the battery to run out at t_2. If eventually at

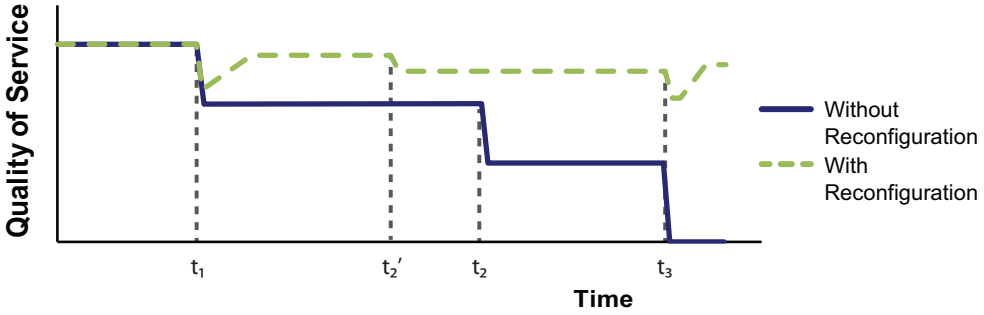


Figure 2.3: Conceptual decrease in quality of service by context changes. At t_3, the centralized component dies, which renders the WSN without reconfiguration useless.

t_3 the node that aggregates sensor information and transmits it to the base station crashes, the complete WSN will be rendered useless.

If the inherently changing context can be encompassed into the design of the WSN, the application's performance could resemble the course of the dashed green line in Figure 2.3 much more. Whenever (an element of) the context changes, the WSN should be able to detect this and act on it, such that it maintains a suitable configuration even when the context is unpredictable. Going back to the example, the WSN could notice one of its sensors is generating inaccurate results at t_1 and either adjust its sensitivity or disable its sensor to prevent incorrect readings. At t_2', the node that is experiencing much interference could detect the higher energy demand. By increasing the length of its duty-cycle, it can avoid completely running out of energy at t_2, with only a small decrease in performance. Additionally, if a component providing important functionality crashes, the functionality could be transferred to another node, in which case the WSN could continue to run after t_3. In this way, a developer does not have to determine the exact configuration, sensor accuracy or other properties at design time. Instead, he or she can delegate these choices to the network, and let them be determined at run-time.

For these and many other scenarios, reconfiguration can increase the lifetime and suitability of a WSN in a changing context. However, because of the limited resources of the nodes, WSN deployments are often tailored to specific application requirements in order to get the maximum performance possible [28]. This results in a large variety of application-specific components. As an extension to these components, reconfiguration could also take many forms, tailored specifically to each application. However, when the application-specific elements can be isolated, a generalized middleware can be constructed that provides a uniform solution for reconfiguring many different applications. The key to this generalization is to be able to separate scenario-specific information from general reconfiguration constructs. The resulting reconfiguration middleware allows a developer to focus on possible failures individually, without having to consider combinations of failures or worry about how the detection and reconfiguration logic should be implemented.

2.3 Reconfiguration Middleware for WSNs

Earlier research efforts for WSN middleware have focused on various goals, ranging from simplifying localized data sharing between nodes [40, 6] to providing a complete and generic data collection platform [41, 27]. Fewer publications in recent literature discuss solutions for reconfiguring WSNs.

Reconfiguration can be performed in several ways. One solution is to monitor the WSN's quality of service at the base station and reprogram (parts of) the sensor nodes when necessary. An alternative is to locally monitor the context and perform reconfiguration on each node individually.

2.3.1 Centralized Reconfiguration

Kogekar et al. [20] detect the need for reconfiguration of a WSN by creating a global model based on information coming from sensor nodes at run-time. Next, a design-space search is performed in order to come up with a suitable new configuration, which is subsequently transferred on to the sensor nodes. However, because the tool chain of the dominant WSN operating system, TinyOS, compiles software components to a static image, this requires replacing the entire code image after deployment. This uses much bandwidth and energy [29].

Maté provides a method for creating virtual machines for sensor nodes that execute small script-like programs [22]. These scripts can be sent to a node and loaded and unloaded at run-time. Although this approach is more efficient than reprogramming, a disadvantage is that applications are required to be expressed as a list of generic operations, with little room for specialized functionality.

FlexCup [29] and Dynamic TinyOS [31] therefore focus on adding flexibility in node images. They do not exactly qualify as middleware, as they mainly alter the TinyOS compilation process to allow for linking software components on the node after deployment. This way, nodes can be reconfigured by uploading new individual software components to the nodes and deleting unused parts. Because in some cases this still requires a restart of individual nodes, it is not considered run-time adaptation. However, the reconfiguration does occur after deployment and if the state of a node can be restored after reconfiguration, the WSN can continue its operation. Also, determining when to change which component is done manually in these solutions.

The biggest concern for these solutions is that the reasoning over reconfiguration is done centrally and requires a global view of the WSN. This is not feasible for most actual deployments, because centralized algorithms scale poorly. Tracking all changes in the network requires much communication and thus energy, and global views of a network often suffer from large delays.

2.3.2 Distributed Reconfiguration

A solution in which nodes reconfigure themselves without a centralized algorithm, is Impala [25]. Each node contains an application adapter and update component.

These respond to certain events in the network and periodically check system and application parameters. Software components are modeled by a finite state machine. If specific preconditions are satisfied, the application is transfered to a different state. A downside of this approach is its granularity: an application can only have a limited number of states, and even small changes take up a complete state. Also, little attention is paid to the reasoning on reconfiguration, and the authors have not yet succeeded in implementing it on a sensor node platform.

Solutions like TinySOA [33] and OASiS [21] take a service-oriented approach to reconfiguring WSNs. Here, nodes publish their functionality as a service to which other nodes can subscribe. When an event in the network or the context decreases the functionality that a specific node offers, the subscribed nodes try to find other nodes with a better service level for the required functionality. However, in sparse networks where there are few alternative nodes with a certain functionality, the application can still fail, since these solutions do not consider changing the nodes themselves, but only the relations between them.

In ASCENT, reconfiguration is modeled as a coverage problem [3]. Nodes either actively contribute to some network functionality or remain passive to save energy, depending on the number of active nodes in their neighborhood. This ensures specific functionality is available for each area in the network. However, this can only be achieved in a sufficiently dense network. As with service-oriented sensor networks, ASCENT cannot prevent an application from failing in case a node fails while there are no alternative nodes with similar functionality.

Distributed reconfiguration promises a more scalable approach for WSNs, but the available solutions generally assume dense networks, place a heavy burden on system resources or are not generally applicable to multiple WSN applications. There has been little focus on how reconfiguration can be performed locally, in case little coordination is possible. This thesis focuses on a lightweight solution for localized reconfiguration, suitable for a large range of WSN applications, including sparse networks.

2.4 Reconfiguration Intelligence

Achieving automatic reconfiguration requires some intelligent component to reason about when to change which components. In the larger area of computer science, artificial intelligence has been a research topic for many years and has branched into many subcategories. Some of these can be interesting to apply in the context of wireless sensor networks.

When considering WSNs that reconfigure according to their context, one solution could be to treat the reconfiguration as a multi-parameter optimization problem. Several methods exist for solving these type of problems, like genetic algorithms or linear programming techniques [43, 35]. In ZeroCal, this approach is used to find optimal duty cycle periods for a low-power MAC protocol [30]. However, applying this to all elements that make up a WSN's context is complex. More importantly, us-

ing reconfiguration could make it possible to target different scenarios with different assumptions and base components. It would be nearly impossible to come up with a single accurate model for all contexts and configurations in which a WSN would run, and shape this as an optimization problem.

An alternative solution is to use a heuristics based approach like expert systems, to reason over reconfiguration. In expert systems, the necessary information is represented by facts in a database [32]. A repository of rules encapsulates knowledge about the system and is used to infer new information or determine which action is suitable. In FACTS [38] these constructs are used to support an event-condition-action based middleware. DSN [4] uses similar constructs, but only to simplify WSN development by creating a new declarative programming language. Although the authors argue that it simplifies WSN development, the solution is used to define standard node behavior, but both solutions do now allow to reconfigure the node's application depending on its context.

MoMi [7] uses a rule-based system to detect faulty nodes in a wireless sensor network. Observations about local and surrounding nodes are compared with each other based on predefined rules, after which conflicting observations are sent to a gateway. The gateway in turn generates a prediction of possibly faulty nodes. The conclusions however do not lead to node reconfiguration; as such it provides techniques for monitoring the WSN that should be used in combination with other solutions for reconfiguration. Additionally, the use of a central gateway makes the solution poorly scalable for large networks.

CHAPTER 3

Middleware Design

Because a WSN's operation is interlinked with an unpredictable context, a way to improve the quality of service could be to encompass context changes into the WSN design process. This thesis focuses on this idea by providing a middleware that supports the reconfiguration of sensor nodes at run-time. Reconfiguring sensor nodes requires knowledge about the application; supporting a wide range of applications requires a uniform way to represent this knowledge, loosely coupled to the middleware's operation. A design is proposed in which developers provide application knowledge, indicate the elements of interest of the context and define capabilities for reconfiguration in a uniform way, such that the middleware can perform reconfiguration for a large variety of applications.

This chapter describes the design of the middleware. In Section 3.1, more details are provided about the requirements and the general setup for such a middleware. Section 3.2 discusses how the information about the WSN's application and context that are required for run-time reconfiguration is represented in the middleware. A distinction is made between knowledge required during the design of the system and during operation after deployment. Section 3.3 goes into detail about how this information is subsequently used by the inference engine for analyzing the run-time environment. Finally, Section 3.4 explains how reconfiguration is performed once the middleware has inferred that such an action is required.

3.1 Goals and Requirements

As mentioned in Section 2.3, there have been several efforts on producing reconfigurable sensor networks. This thesis focuses on providing a middleware that reasons about when to reconfigure sensor nodes, based on run-time information and a model of the WSN application. To make the solution applicable to a large variety of networks, reconfiguration should be performed without the requirement of a centralized component. This means information is gathered and processed locally on individual nodes, which allows reconfiguration to be performed without relying on communi-

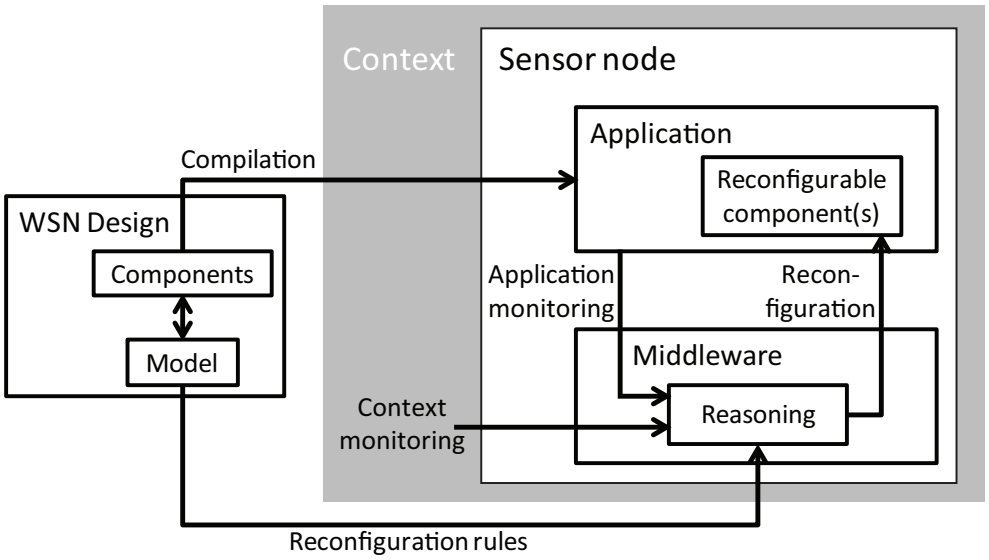


Figure 3.1: The proposed approach for reconfigurable sensor nodes.

cation. Nonetheless, the design should incorporate possibilities for extension with coordinated reconfiguration if possible.

WSNs are often tailored to their use case, which results in a large variety of very specific software components. As reconfiguration applies to these components, it can also very specifically suit only the component that requires reconfiguration. However, this results in very little code reuse, weak separation of concerns and maintenance problems when those components are integrated into an application. In order to create a generalized middleware that provides a uniform reconfiguration solution to many different applications, the application- and component-specific elements should be isolated. The key to this generalization is to separate scenario-specific information from general reconfiguration constructs.

The solution proposed in this thesis is schematically depicted in Figure 3.1. During development, the developer determines the requirements and defines the design of the WSN. This results in the components that make up the sensor network application, including those that can be reconfigured. To be able to reason about when to reconfigure which components, the middleware requires knowledge about the application, its domain and its requirements. The developer provides a model of this information, from which a system representation for the middleware can be generated. The middleware uses this in combination with the run-time information the node gets from monitoring its context and application, to infer if reconfiguration of specific components of the application is required.

An important requirement for middleware in general is that it should not take up much resources; it is used as an extension of a platform on which the actual application itself should be able to run. For middleware on wireless sensor networks, this translates to limited memory, computation and energy use. This thesis focuses on re-

configuration on the local node, requiring no additional communication, thus limiting a possibly high additional energy demand. This however also means that reconfiguration can only be performed using a limited view of a node and its context, with possibly a limited influence on the WSN as a whole. To extend a node's reconfiguration capabilities, a suggestion for future work is to share run-time and design-time information among neighbors. Several solutions have been proposed in literature on how to achieve this efficiently [6, 40].

Developing WSN applications requires keeping account of many problems that interfere with communication and other operational aspects, as mentioned in Section 2.1. To limit the focus, the solution functions as middleware on top of TinyOS, which provides a platform for controlling hardware, communication, scheduling, et cetera. Hardware and software components provide status and diagnostic information at run-time. Obviously, reconfiguration requires the components to be reconfigurable and adjustable.

3.2 Knowledge Representation

To be able to reconfigure the application at run-time, the middleware requires knowledge about the application. However, a generalized approach requires application-specific information to be separated from common reconfiguration logic; the latter is applicable to all applications and thus can be generalized. Application-specific information consists of which capabilities are available for effectuating a change in the application and of information about when those capabilities should be used. If this information can be provided in a uniform way, the middleware can reason about reconfiguration independent of the specifics of the application. This section provides more details on how this knowledge of the application is provided to the middleware.

There is a difference between information supplied to the middleware at run-time and at design-time. As depicted by the bottom arrow in Figure 3.1, design-time information consists of the model of the WSN application transformed into a concrete set of reconfiguration rules. The middleware uses these rules in its reasoning about reconfiguration. The model concerns the available components and (reconfiguration) functionality, and how they depend on the context of the node. For the case study, an element in the model could be the presence of a sensor component, of which the gain depends on the sample variance. This information is defined before the creation and deployment of the sensor network and does not change during operation. Note however that it contains descriptions of how components depend on *run-time* properties.

At run-time, the middleware requires information about a sensor node's context, including the applications run-time parameters. This flow of information is represented in Figure 3.1 by the arrows labeled with 'monitoring'. It contains details of sensor node's sensed environment and the state of the hardware and software components. As the design-time information provided the middleware with knowledge about how components depend on the context, run-time information provides the de-

tails of this context. Combined, these allow the middleware to make decisions about the application. In the case study, this could be the number of neighbors of a node, its accelerometer's sensitivity, or the node's battery level.

The following subsections provide more detail about how this knowledge is represented in the middleware.

3.2.1 State Monitoring

The provided model states how the application depends on the node's context. As this context is dynamic and encompasses many elements that are unknown during design-time, it has to be monitored at run-time. The middleware has to match this run-time information with the component requirements defined in the model of the application. This means it has to be represented in a uniform way. In the proposed solution, the context of each node of the sensor network is represented as a (set of) variable(s). These variables can contain local information about the node itself, e.g. the variance of its sensor, or about network characteristics, e.g. the number of 1-hop neighbors. Using this model, the state of a node's environment at a specific time can be captured in a set of these variables.

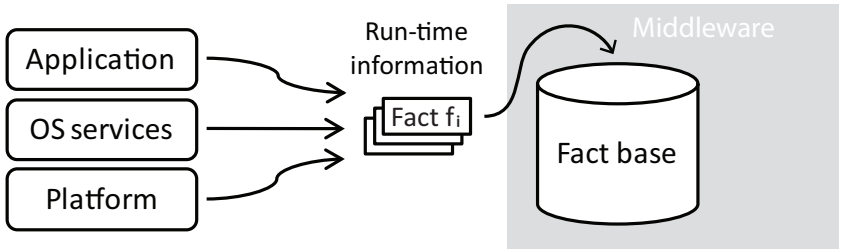


Figure 3.2: Schematic overview of context monitoring. Components offer context information as facts to the fact base.

Each of these variables possibly relates to a different component on the sensor node, which means context information comes from components across different system layers. To unify this information, the middleware maintains a local repository in which any component can store variable valuations, through a standardized interface. Figure 3.2 provides a schematic overview of the collection of this run-time information. The values for each element of the context are stored as simple data constructs called *facts*. The repository in which the collection of facts is stored will be referred to as the *fact base* and is similar to the construct used in FACTS [38]. However, in FACTS the authors use this to determine application behavior, whereas here only reconfiguration actions are deduced from it. The fact base defines the context of each node. Example 3.1 illustrates how the context of a node of the case study is stored. Each fact contains the variable-name as a unique identifier, the value of this variable, and additional meta-data like the time of the most recent update. Every component can introduce and update facts whenever it notices a change in its environment, possibly overwriting previous values.

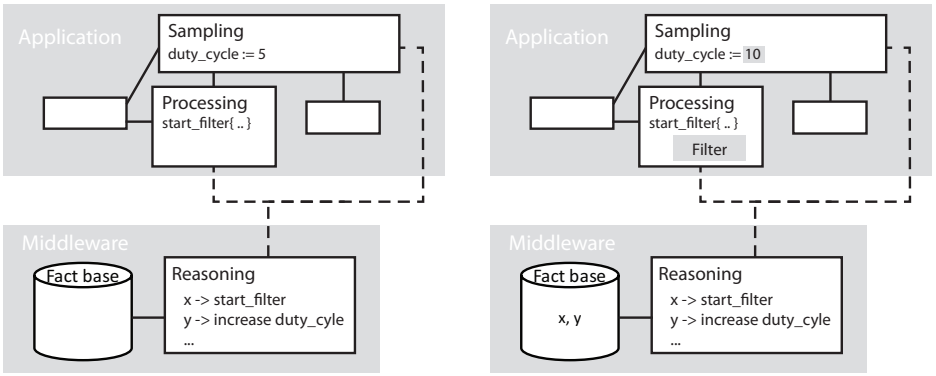
Example 3.1. Run-time information about the average sensed acceleration, remaining energy and routing protocol can be represented by the following set of facts:

$$f_1 = (\text{sensor.average}, 4\ \text{g}, 23{:}59 - 04\text{-}02\text{-}2011)$$
$$f_2 = (\text{battery.energy}, 50\ \text{mAh}, 08{:}45 - 05\text{-}05\text{-}2011)$$
$$f_3 = (\text{routing.protocol}, 2, 00{:}01 - 01\text{-}01\text{-}2011)$$

3.2.2 Application and Configuration Model

As the fact base provides information about the context of the WSN at run-time, the middleware analyzes this to reason about whether the application requires reconfiguration in that context. The requirements for reconfiguration and the available functions to perform reconfiguration are part of the model of the application, which is provided to the middleware during design-time.

In the application model, a configuration is defined as a set of interconnected functional units, and have specific properties and parameters. As an example, consider Figure 3.3 in which the (re)configuration of a typical WSN application is presented. Components of the application are represented by boxes and interface with each other via the solid lines. The middleware itself also consists of software components (e.g. the reasoning component), with specialized interfaces represented by the dashed lines, to the application components that are subject to reconfiguration.



(a) Before (b) After

Figure 3.3: Example model of a sensor application undergoing reconfiguration. The introduction of x and y to the fact base results in a filtering component being activated and the duty-cycle being incremented from 5 to 10.

Depending on the context information of the fact base, the middleware uses these interfaces to indicate that a specific reconfiguration is required, along with any parameters needed for the reconfiguration. The reconfiguration commands defined in these interfaces shall be called *actions*. Each action can consist of a sequence of operations. Actions can for instance command components to change their parameters,

or start or stop them. In Figure 3.3b, the insertion of x and y to the fact base results in the processing component starting a filtering-algorithm, and an increase in the duty-cycle parameter of the sampling component. In turn, this can lead to new facts being entered into the fact base.

During the design of each component, the developer defines to which actions the component can respond and what the subsequent operations for that action are, thus implementing a specific interface defined by the middleware. This approach allows the components to execute the reconfiguration operations regardless of the reasoning behind it, while the middleware can reason about reconfiguration regardless of how the specific actions are actually implemented.

3.2.3 Conditional Reconfiguration

Besides knowledge about the application components and their supported reconfiguration actions, the middleware requires knowledge about what type of reconfiguration should be performed in which context. As this does not change after deployment, it is also part of the model provided to the middleware during design-time.

A developer specifies the reconfiguration behavior using a high-level programming language. During compilation, a collection of rules is generated from this specification and stored in the *rule base*. This allows the middleware to reason over each of the available reconfiguration actions.

Rules are a relationship between one or multiple *conditions* on the context and an *action*. If the logical conjunction of the conditions is true, the action is triggered.

$$condition_1 \wedge condition_2 \wedge \ldots \wedge condition_n \rightarrow action$$

Using a collection of rules to describe knowledge at design-time and produce new information at run-time is a well known concept in the area of artificial intelligence. The approach is similar to the *knowledge base* of an *expert system*, a concept first described by Feigenbaum et al. [10].

A condition is a boolean function that operates on an element of a node's context (represented by a fact) and a predefined *value*. The comparison operator can be any developer-defined function, e.g. $\leq$ or $>$, or possibly operate on a fact's meta-data, e.g. 'inserted later than'. The operator does not necessarily have to be binary: if the high-level programming language requires more complex operators, the value can contain a reference to other data structures to which the fact can be compared, including a null-value for unary operation, or other facts.

$$fact, operator, value \rightarrow boolean$$

When all conditions for a rule are true, the action is performed. An action consists of operations that a reconfigurable software component performs. By using simple identifiers for each action, the specific implementation can be abstracted away from the middleware.

All conditions in a rule have to be satisfied in order for the middleware to conclude the rule is satisfied, which implies a logical *conjunction* of conditions. In the case that some reconfiguration is required for multiple distinct contexts, a *disjunction* is required, in which any of several conditions can be true for a certain rule. To represent this in the rule base, multiple rules can be created that result in the same action when satisfied. An overview of possible relations between facts, conditions, rules and actions is provided in Figure 3.4.

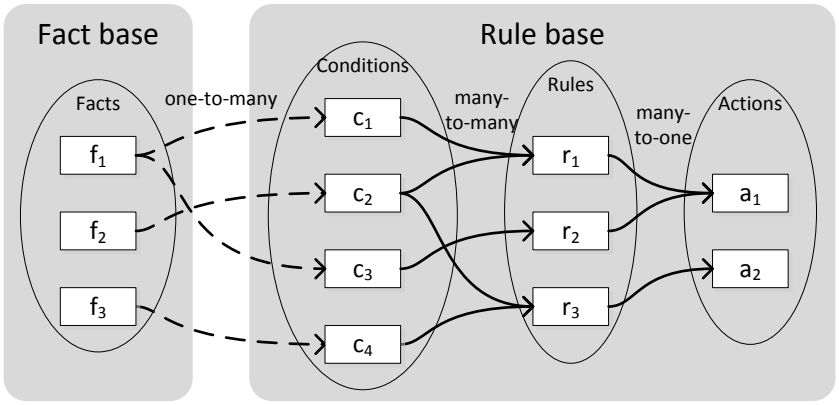


Figure 3.4: Relations between facts, conditions, rules and actions.

A fact f can be referred to by multiple conditions. A conjunction of multiple conditions can be used in a rule. Conditions can also be used in multiple rules. Also, multiple rules can lead to the same action a. Note the difference in relationships indicated by the use of different arrows: the actual facts in the fact base are inserted to the fact base at run-time, while the fact IDs, rules, conditions and actions that use facts are defined at design-time in the rule base. To distinguish the different relationships between the data elements, the definitions in Table 3.1 are used throughout the rest of this thesis. Example 3.2 explains the various combinations in which rules and conditions can be used.

Set	Description	Example from Figure 3.4
$relatedConditions\,(f)$	Set of conditions by fact f.	$relatedConditions\,(f_2) = \{c_2, c_3\}$
$requiredConditions\,(r)$	Set of conditions for a rule r	$requiredConditions\,(r_1) = \{c_1, c_2\}$
$relatedRules\,(c)$	Set of rules that use condition c	$relatedRules\,(c_1) = \{r_1\}$
$possibleRules\,(a)$	Set of rules that lead to action a	$possibleRules\,(a_1) = \{r_1, r_2\}$

Table 3.1: Definitions used to distinguish relationships between data elements in the rule base.

Example 3.2. Consider a typical sensor node containing an acceleration sensor, a communication component and battery. The system requires that the sensor sensitivity should be decreased (action a_1) if the measured acceleration becomes a little too

high (condition c_1), but only if the energy level is low (condition c_2). This can be represented by a rule r_1. The second rule r_2 states that if the acceleration reaches extreme values (condition c_3), sensitivity should be decreased regardless of the energy level. This rule thus results in the same action a_1 (decreasing the sensor sensitivity), but requires a different condition. This condition nonetheless tests the value of a fact about *sensor.average*. Thirdly, a reconfiguration in the routing component is required (action a_2) if the energy level becomes too low. This rule r_2 can reuse condition c_2 in combination with a fourth condition regarding the routing protocol. Using the facts of Example 3.1, this results in the following set of conditions and rules:

Conditions	
c_1:	sensor.average > 10
c_2:	battery.energy < 50
c_3:	sensor.average > 100
c_4:	routing.protocol $== 2$

Rules	
r_1:	$c_1 \wedge c_2 \rightarrow a_1$
r_2:	$c_3 \rightarrow a_1$
r_3:	$c_2 \wedge c_4 \rightarrow a_2$

Table 3.2: Rule base for Example 3.2.

3.3 Rule-Based Inference

When all the required knowledge is provided to the middleware, the middleware can reason over the node's context, determine the required reconfiguration actions, and request these actions from the reconfigurable components. Details about the reasoning process are provided next.

Using facts as defined in Section 3.2.1, the middleware is notified whenever a component notices a change in (some element of) the context. The inference engine compares this with the design-time knowledge stored in the rule base, and signals whenever all conditions for a specific action are satisfied, such that the corresponding action can be performed.

Traditionally, expert systems can use *forward* or *backward* chaining to come up with conclusions based on a rule base. With backward chaining, the middleware performs *hypothesis tests* in which it looks for possible reconfiguration actions that should be performed. For each action, the relevant conditions would be checked to see if all of them are satisfied. Using forward chaining, the inference engine uses facts as entry point. For each fact, it checks whether any of the related conditions are satisfied. Subsequently, the inference engine checks for which rules all required conditions are satisified.

Backward chaining might get a natural preference over forward chaining because it reasons from the point of the resulting actions. However, this does not use the information that conditions can only change if a fact is updated, while these changes may happen only sporadically. The forward chaining alternative does allow to use this information, which is why it is the algorithm of choice for this middleware design.

Updating or creating a fact f triggers the inference process, which iterates over the conditions of all rules. Conditions not in $relatedConditions\,(f)$ can be skipped, as well as remaining conditions that are used by rules that have been falsified by earlier condition evaluations. After the rule base has been evaluated, the actions for all satisfied rules are executed.

While the approach of checking only conditions referring to the changed fact might be more efficient than the standard backward inference, it is still naive. Several optimizations are possible, which will be discussed in Chapter 4. The Rete algorithm is a more efficient pattern matching algorithm, introduced by Forgy [12]. However, because this approach is known to consume more memory, and the sensor nodes are required to run the middleware next to an actual application, the preference was to use the naïve algorithm instead. The evaluation in Section 4.6 shows that the run-time overhead is very low and poses no problem for use in an actual sensing application.

3.4 Actuation

Once the inference engine has found a satisfied rule, the middleware should bring about the action on the specific components. As mentioned in Section 3.2 and shown in Figure 3.3, each component that is subject to reconfiguration shares an interface with the middleware. Whenever a rule is satisfied, the middleware sends a request for the specific action via these interfaces, along with the required parameters. Each component can subsequently execute its own specific operations required for the action. This not only allows multiple components to respond simultaneously to a reconfiguration, but also informs components that do not require any change about a possible delay in interfacing with the components that do. Because the middleware only indicates which reconfiguration action should be taken, without specifying how this should be performed, it provides a generic reconfiguration component that can be used with various system architectures.

To verify that a reconfiguration successfully adapted the system to the new context requirements, actions can result in new facts being entered or updated in the fact base. Entering these derived facts can be done explicitly by the reconfiguring software component, e.g. by inserting a fact that states that a specific action has been performed. Fact updates can also result implicitly from performing an action when, as a result of the reconfiguration, a parameter in a related component changes and causes that component to update a fact. The fact update triggers the inference process again to re-evaluate the conditions affected by the new update. When the updated fact subsequently does not result in satisfied rules, the related actions will not fire and the reconfiguration stops.

The process of feeding back the results of a reconfiguration to the middleware introduces new dynamics into the system and should therefore be carefully examined when designing the reconfiguration logic. The details regarding this process are application specific: no general solution exists for this. Finding solutions for these type of problems in adaptive systems is subject of ongoing research.

CHAPTER 4

Implementation

A proof-of-concept middleware has been implemented to assess the rule-based reconfiguration approach on a typical WSN platform. Using the implementation, details of the approach of reconfigurable middleware were refined, and memory and performance characteristics of the middleware were evaluated. The implementation's hardware platform provided by TNO consists of G-Node sensor nodes [36], which have been applied in various commercial applications. This platform was designed to be compatible with TinyOS [24], a commonly used operating system for sensor nodes.

This chapter discusses the details of this implementation. Section 4.1 will go deeper into the TinyOS platform and the characteristics of developing software for this platform. In Section 4.2, more details are provided about the implementation of the fact base and how it stores the node's context information. Section 4.3 provides more information about the system representation of reconfiguration rules. Section 4.4 discusses how the reasoning of the inference engine is implemented, while the implementation of reconfiguration actions is discussed in Section 4.5. The chapter closes with an evaluation of the implementation, concerning both memory and processing overhead, in Section 4.6.

4.1 Implementation Platform

The proof-of-concept middleware is implemented on TinyOS, an operating system for WSNs [24]. TinyOS is based on a programming language called nesC, which is an extension to C and allows developers to structure their software using components named *modules* and *configurations*. In a module, certain functionality is implemented using "private" state and functions that operate on this state. Configurations *wire* components using (bi-directional) *interfaces*. Interfaces describe *events* that a component can generate and *commands* that can be called by a user. This approach corresponds well with the model described in Section 3.2.3. More information about TinyOS can be found in the book by Levis and Gay [23].

The reconfigurable middleware is implemented as a configuration that can be wired to system and application components. A schematic figure of the middleware components and interfaces is provided in Figure 4.1. In this figure, public interfaces are marked by grey ovals, components are represented by squares and arrow labels indicate the internal wiring of interfaces. The three interfaces allow a WSN application to supply input for and read monitoring information and receive reconfiguration signals from the middleware. The function of the components will be explained in the following sections.

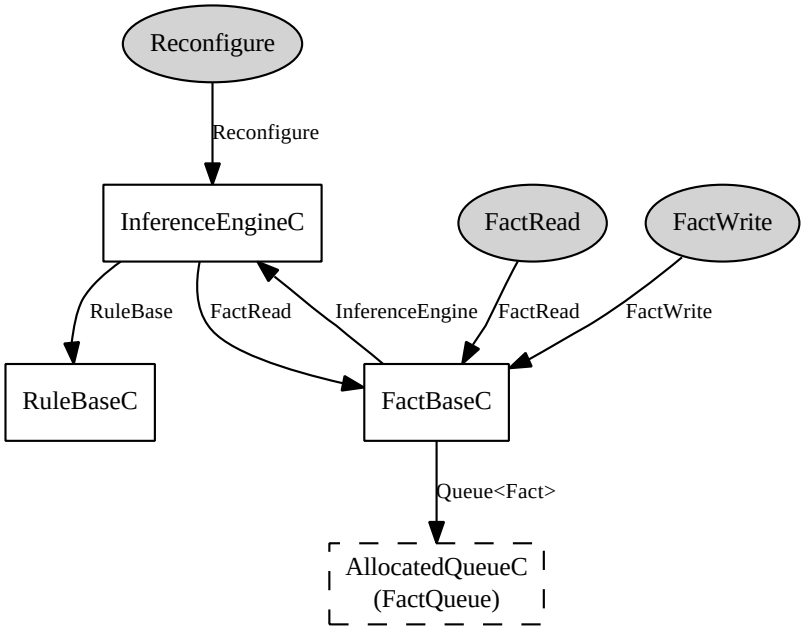


Figure 4.1: Schematic overview of the essential components of the middleware. Public interfaces are marked by grey ovals, components are represented by squares and arrow labels indicate the internal wiring of interfaces.

4.2 Component Monitoring

Components that provide monitoring information can supply this to the middleware via the *FactWrite* interface, provided in Listing 4.1. The component supplies an 8-bit identifier and either a new value for the parameter or a delta value that is added or subtracted to the stored value. Updating a fact with a delta value saves a component from having to read a fact before writing a new value, e.g. when incrementing a timeout counter. In case the application does require reading facts, for instance when an energy saving reconfiguration adapts a duty-cycle depending on the current battery level, this is possible via the *FactRead* interface.

```
interface FactWrite {
  /* Decrease the fact with the provided ID with the
   * provided delta value, down to but not beyond zero.    */
  command error_t decrementFact(uint8_t id, uint16_t delta);

  /* Increase the fact with the provided ID with the
   * provided delta value, up to but not beyond UINT16_MAX.*/
  command error_t incrementFact(uint8_t id, uint16_t delta);

  /* Set the fact with the specified ID to the specified
   * value.                                                */
  command error_t writeFact(uint8_t id, uint16_t value);

  /* Signaled when the Middleware is ready to accept facts.*/
  event void startDone();
}
```

Listing 4.1: The FactWrite interface used by components to provide context information to the middleware.

TinyOS requires static allocation of all data structures, which means the current implementation uses fixed sized arrays to store its information. As new values for facts overwrite previous values, the maximum number of facts in the fact base is defined at design-time, providing safety from running out of memory at run-time. However, it can lead to over-provisioning when some facts are not provided by any application component while memory has been reserved for them. Future work could focus on more dynamic allocation for the fact base.

The current implementation supports 16-bit fact values and stores these with a 32-bit timestamp as a fact struct in a fixed-sized array. The identifier is used as the index of the array position at which the fact struct is stored, which brings the memory cost of a single fact to 6 bytes. Fact values are initialized to zero and are updated each time a component defines a new value.

When a fact is updated, it means a change in the context has been detected. To achieve the optimal configuration for the application in this changed context, the middleware should check whether a reconfiguration is required. As explained in Section 3.3, the inference process uses the fact change as a starting point for evaluating the rule base.

4.3 Rule Base

The design-time information about the application model is represented by a set of reconfiguration rules, as explained in Section 3.2. This allows the middleware to match the design-time information with run-time facts, in order to infer which reconfiguration actions should be performed. The following description explains how these rules

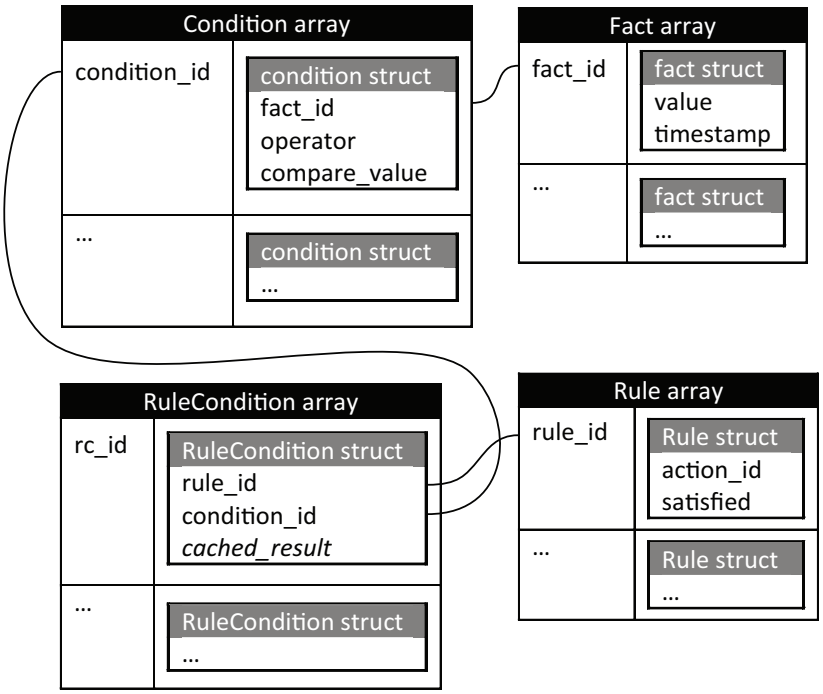


Figure 4.2: Data structures used for knowledge representation. The `cached_result` field in the RuleCondition struct is optional and speeds up the inference process.

are stored in such a way that they can be used by the inference engine. Ultimately, the rule base will be generated from the application model by a compiler, so that a developer can define the reconfiguration behavior in a human-readable language.

The data structures used to store the rules are depicted in Figure 4.2. Each condition is stored as a tuple of an 8-bit fact ID, an 8-bit reference to a comparator operator and the 16-bit operand to which the fact should be compared. This tuple is stored in a *condition array*, indexed by the ID of the condition.

A rule is a relationship between one or more conditions and an action, as explained in Section 3.2.3. The *rule array* stores the 8-bit action identifier for each rule, at the location pointed to by the rule identifier, along with a variable that indicates whether the rule is satisfied. When multiple rules leading to the same action are required, the same action ID can be stored at different locations in the array. This represents a disjunction of conditions for the corresponding action.

The relationships between conditions and actions are stored in an array with *RuleCondition* tuples. This allows conditions to be used by multiple rules, without requiring multiple entries in the condition array. Each RuleCondition tuple reifies the relation between a single condition and a single rule, by storing references to their locations in the respective arrays. In this way, the collection of RuleConditions that refer to the same rule ID r make up the conjunction of the conditions for that rule:

namely $requiredConditions\,\{r\}$. The RuleCondition optionally contains a boolean variable in which the previous evaluation of the condition is cached. This will be further explained below.

4.4 Inference Engine

Using the data structures described above, the inference process reasons over the provided rules and indicates which rules are satisfied in the context represented in the fact base. This section provides more details about the inference process. A schematic representation of the process is provided in Figure 4.3, which indicates the different stages in the process.

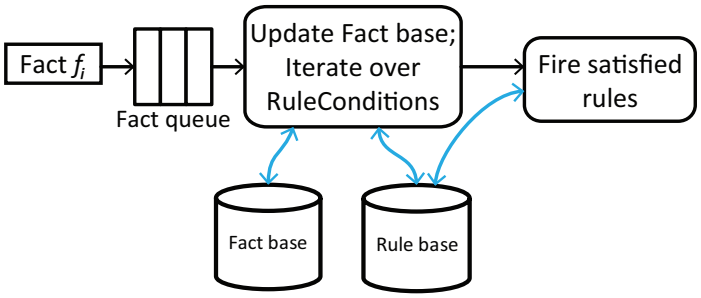


Figure 4.3: Schematic overview of inference process. The dark arrows indicate the flow of the inference process, the light arrows show the flow of information.

If the fact base is allowed to be changed during the inference process, race conditions might occur between fact updates and condition evaluations, making the outcome of the inference process unpredictable. To counteract this, each time a component provides a new fact, the fact is buffered in a queue until the inference engine is ready to process it.

Processing a fact means updating the necessary values in the fact base. After this, the rules in the rule base are checked to see if they are satisfied in the new context. The actions for the satisfied rules are subsequently executed.

The process of labeling the rules is represented by Algorithm 4.1. Each rule starts out as satisfied, i.e. the variable `satisfied` of the rule struct is *true*. The inference process iterates over all RuleConditions, as they provide references to both the condition that requires evaluation and the rule that possibly is falsified by that condition. As an optimization, RuleConditions for which the rule has been falsified by previous conditions, are skipped in line 5.

A second optimization allows the inference engine to skip evaluation of conditions that are not affected by the updated fact; instead the cached result from a previous evaluation can be used, as shown in line 12. This however does require an additional boolean variable in the RuleCondition struct, as indicated in Figure 3.4. Furthermore, using a cache introduces an additional cache update operation for conditions that do require evaluation; some RuleConditions therefore cannot be skipped in line 5. As

Algorithm 4.1 labelRules(updatedFact)

```
for all ruleCondition in RuleCondition_array do
  condition ← ruleCondition.getCondition()
  rule ← ruleCondition.getRule()
  if rule previously falsified and condition does not use updatedFact then
    continue to next ruleCondition
  end if
  condition_satisfied ← true
  if condition uses updatedFact then
    condition_satisfied ← evaluate condition
    update cached_result
  else
    condition_satisfied ← cached_result
  end if
  if not condition_satisfied then
    falsify rule
  end if
end for
```

such, the cache provides a trade-off between memory and inference speed, where more complex rules, i.e. rules with many conditions, will be sped up more than the simpler rules.

If the condition evaluates to *false*, the rule's `satisfied` variable is updated accordingly. Subsequent RuleConditions of this rule are skipped: they can no longer make the rule true, since a rule requires all conditions to be satisfied. If the condition evaluates to *true*, the rule's `satisfied` variable remains unchanged: a subsequent condition could still falsify the rule.

When the inference engine has iterated over all RuleConditions, the inference process iterates over all rules and signals the actions of rules that have not been falsified. How this results in the required reconfiguration is explained below.

4.5 Reconfiguration Actions

When all conditions for a rule are satisfied, the inference engine signals an event with the corresponding action ID. Using a form of the publish-subscribe pattern, reconfigurable components can subscribe to the actions of the middleware for which it contains a reconfiguration implementation. Currently, this is implemented with a nesC interface, that defines how an event is signaled by the middleware. The reconfigure interface is provided in Listing 4.2. A reconfigurable component wires to the middleware by implementing the event function, thus subscribing to the actions. By parameterizing the interface as in Listing 4.3, the implementing components can ignore the action requests concerning other components. Providing the correct pa-

```
interface Reconfigure {

  /* Signals whenever a rule is satisfied,
   * providing the corresponding actionid                    */
  event void action(error_t error, uint8_t actionid);
}
```

Listing 4.2: The reconfigure interface via which reconfigurable components receive requests for actions.

```
configuration ReconfigurableConfiguration {
  implementation {
    components Middleware;
    components new ReconfigComponentA();

    ReconfigComponentA.reconfigure ->
        Middleware.Reconfigure[RECONFIG_COMPONENT_A];
  }
}
```

Listing 4.3: Wiring the parameterized interface.

rameters to wire each component to its corresponding reconfiguration interface can easily be automated using the application model provided by the developer.

This approach of the publish-subscribe pattern could easily be expanded in future work by including communication with other nodes. This provides a way to support distributed reconfiguration in which neighboring nodes take up part of the reconfiguration implementation. There have been several publications about implementing the publish-subscribe mechanism efficiently for WSNs [34, 1, 16].

4.6 Evaluation

As the reconfiguration middleware is meant to run alongside a WSN application, its resource use should be minimal. The reconfiguration middleware described above has been put to several tests to measure its resource consumption. Both memory consumption and processing overhead are taken into account. This thesis does not focus on cooperative reconfiguration, which means the node's energy consumption will not be affected by any additional communication. This means considering the processing overhead will also provide an estimation on the middleware's energy requirements.

4.6.1 Memory Use

Since TinyOS only supports static memory allocation, a memory analysis of the compiled program reveals the required space for the middleware's components in both

ROM and RAM. The middleware's main components are the fact base, rule base and inference engine. The fact base and rule base contain application specific information and vary in size depending on the number of rules and facts that have to be stored. The following results are obtained by a memory dump tool and consider both the variable as the invariable memory use.

Component	Size (bytes)
Fact base	380
Inference Engine	228
Glue code	230
Total	838

Table 4.1: Size of middleware components in read-only memory, excluding rules and compiler optimizations.

The ROM required for the middleware's executable code and program constants is listed for each component in Table 4.1, excluding storage required for rules or facts. The TinyOS toolchain applies optimizations such as inlining during compilation, which decreases the actual ROM use depending on the structure of the application [13]. The results shown here are obtained with these optimizations disabled to be able to consider the worst-case scenario. The G-Node can store 116 KB in ROM, which means the middleware uses less than 1% of the available space.

As rules do not change at run-time, they are also stored in ROM. The size of the rule base depends on multiple elements: the Rule, Condition and RuleCondition arrays. Each rule requires a 2-byte entry in the Rule array. Besides this, a variable number of entries in the Condition and RuleCondition arrays are required, depending on the number of conditions required for a rule and the amount of conditions shared by multiple rules. Each entry in the Condition array takes up 4-bytes. An entry in the RuleCondition array is needed every time a rule requires a condition. It takes up 4 bytes when evaluation results are not cached and 6 bytes if they are. For a single rule based on one condition, this gives a total size of 9, respectively 11 bytes.

The size in RAM depends on the number and size of facts and rules as well. In the current implementation, all facts take up six bytes in RAM: two bytes for the value and four for the timestamp. Rules require an additional 1 byte for storing whether they are satisfied. When caching of condition evaluations is used, another two bytes per each RuleCondition are required. Additionally, the queue that the inference engine uses to buffer facts requires 7 bytes per entry.

4.6.2 Processing Performance

Besides memory, the processing overhead of the middleware should be reduced to a minimum as this could interfere with the application. Since this thesis focuses on local reconfiguration, the processing overhead will also be the primary source of additional energy consumption. However, as Table 2.1 shows, the energy consumption

of a sensor node is mostly affected by communication, only remotely followed by the processing done by the microcontroller. For this reason, the energy consumption will not be considered in this evaluation.

Methodology

Measuring the processing time required for evaluating conditions and searching the rule base requires the middleware giving a precise start and stop signal, without affecting the actual processing time. For this analysis, the middleware was altered: an output pin of the G-Node hardware was set high before a fact was updated, and set low directly when the matching reconfiguration action was received by a component. Using an oscilloscope, the pulse width on this pin can be measured, which provides the time that the middleware is busy. This way, the processing time was measured for several scenarios.

Processing response and cycle time

Reconfiguration is intended to adapt the software's components such that it can operate in a changing context. The quicker the middleware responds to a change, the lesser the application is interrupted and the more time the application spends in an optimal configuration.

	Inference time (ms)
Cached	0.301
Uncached	0.286

Table 4.2: Processing time from the initial fact update to the end of the inference process, averaged over 1000 measurements.

A simple experiment was performed to test the reasoning time of the middleware. Using a rule base consisting of a single rule requiring a single condition, a new fact was inserted as soon as the action from the previous fact insertion was triggered. After every 1000 action events, an output pin of the G-node is alternated. This pin was measured using an oscilloscope. This process was repeated 1000 times, for the middleware with and without caching the condition evaluation. The results are provided in Table 4.2, showing that the reasoning over a single rule is performed in well under a millisecond. The table furthermore shows that without caching, the inference process is performed more quickly than with caching condition evaluations. This is because updating the cache requires some time, while for a rule with a single condition, there is no condition evaluation that can be skipped.

A second experiment was conducted in which a large amount of facts was entered to the fact base consecutively to test the throughput of the inference engine. However, as the TinyOS scheduler will not schedule the inference process until after the last fact update, the size of the fact queue is the limiting factor in this situation. Using

principles of queuing theory, an optimal queue length can be found, depending on the number of rules and facts, and the available memory for the application in question.

Scalability

To assess the scalability of the reasoning process with relation to the size of the rule base, a fictional rule base was created with varying numbers of conditions and rules. One fact was periodically updated, resulting in the last condition of each rule being falsified. For the cases in which rules require multiple conditions, the remaining conditions were unaffected by the fact update.

		Without cache			With cache		
		# conditions for each rule					
		1	5	10	1	5	10
# rules	1	0.45	0.54	0.66	0.46	0.51	0.58
	5	0.56	1.02	1.60	0.60	0.86	1.19
	10	0.69	1.62	2.77	0.76	1.29	1.94
	25	1.09	3.41	6.31	1.27	2.58	4.23
	50	1.75	6.39	7.88	2.10	4.67	5.69
	75	2.41	7.40	9.45	2.94	5.80	7.70
	100	4.92	8.28	11.50	3.77	6.82	9.46

Table 4.3: Inference time in milliseconds, for varying number of rules and conditions.

Table 4.3 shows the inference time for both the rule base with and without a cache for the condition evaluations. When rules require only one condition, the inference process takes slightly shorter when conditions are not cached. As explained previously, this is because rules based on a single condition do not require any evaluation of other conditions aside from the one affected by the fact. However, as rules require more conditions, using cached condition evaluations provides an increasing advantage over evaluating every condition on each iteration. In the most optimal case provided by the table, the advantage is over 17%. However, because in that case, the rule base contains 1000 RuleConditions (10 conditions for 100 rules), caching requires an additional 2 kilobytes of RAM.

The table also shows that the inference times are all well under one second. Compared to duty cycles typically used in WSNs, this is very short. Furthermore, the number of conditions tested here is very high compared to what a typical application would require, as can be seen in the case study in Chapter 5.

CHAPTER 5

Case Study

A case study regarding structural health monitoring has been used to evaluate the practical use of the middleware. Modal analysis is a method that structural engineers use to determine the vibration characteristics of a structure, from which its strength and overall structural health can be determined. Several publications have previously used sensor networks in this context [42, 26, 19]; TNO is interested in the possibilities of WSNs for monitoring off-shore wind turbines and other applications.

For this thesis, a modal analysis application has been implemented, which provides vibration characteristics of a structure. Four scenarios were defined in which the quality of service of the application would decrease. For these scenarios, reconfigurations were defined that can limit the decrease. These reconfigurations were represented as rules and incorporated into the proof-of-concept middleware. Using the resulting reconfigurable WSN, the proposed approach was assessed on effectiveness and efficiency.

In this chapter, details are described of the application design and performance of a structural health monitoring WSN. Section 5.1 explains the process of modal analysis and provides details of a WSN designed for this purpose. In Section 5.2, four scenarios are described which decrease the quality of service of the WSN. For each of the scenarios, details are provided about how the proposed middleware reconfigures the WSN to suit its context. Results of experiments performed on the implementation are provided in Section 5.3. This chapter closes with a discussion of these results and the approach of the middleware in Section 5.4.

5.1 Sensor Network Design

In modal analysis, the vibration patterns of a structure are determined by analyzing the acceleration at different locations on the structure during a certain time period. By applying a fast Fourier transform (FFT) to the acceleration time series, peak vibration frequencies can be determined. The phase at those peak frequencies of all locations combined provide the shape of the vibration. Doing this periodically al-

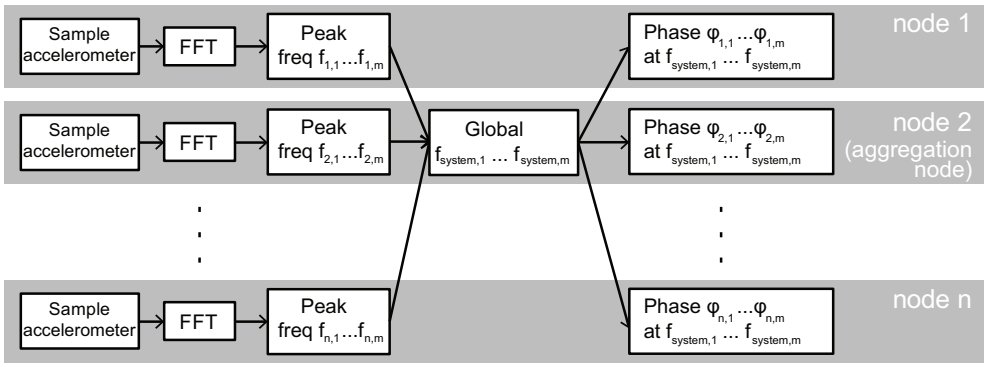


Figure 5.1: Steps in the modal analysis algorithm.

lows a structural engineer to determine a change in vibrations, which could indicate a weak spot or failure in the structure.

Using the approach of Zimmerman et al. [42], displayed in Figure 5.1, the amount of data transmitted by sensor nodes can be reduced greatly compared to nodes simply sending accelerometer samples. At each measurement location, a sensor node is placed that samples its accelerometer, and applies a 1024-point FFT. Using the power spectral density, the peak frequencies on that point on the structure are determined by the node. For simplicity, the case study limits the number of frequencies to the strongest one. The peak frequency values of all nodes are collected at one of the nodes, which determines the most common peak frequency in the network and broadcasts this back to all nodes again. This node, which will be referred to as *aggregation node*, beacons periodically to inform other nodes about where to send their peak frequencies. Finally, each sensor node determines the phase at this frequency and transmits this to the base station. The case study repeats this process periodically.

This algorithm has been implemented on three G-Nodes equipped with an ac-

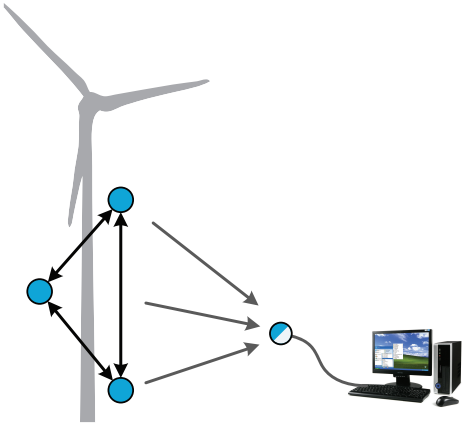


Figure 5.2: Schematic representation of the network setup used for the case study.

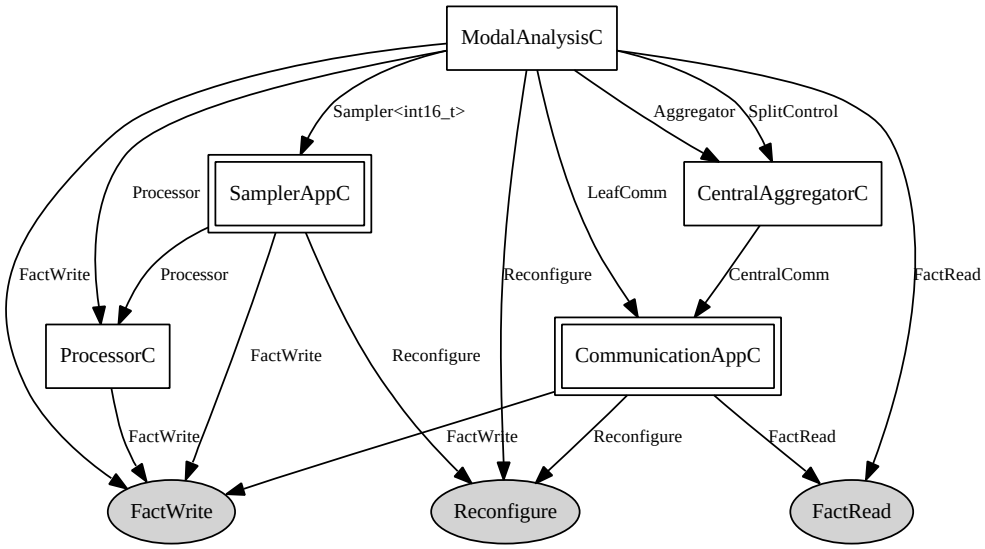


Figure 5.3: Overview of the modal analysis application components, using the same representation as in Figure 4.1. Doubly lined squares represent a component consisting of multiple sub-components.

celerometer sensor board. A separate sensor node acts as a base station that collects the transmitted data for storage and further processing on a pc. Figure 5.2 shows the network setup schematically with communication directions represented by arrows. An overview of the TinyOS components is provided in Figure 5.3. Components providing information to the middleware are wired to the *FactWrite* interface. Reading from the middleware and listening to reconfiguration actions is done via the *FactRead* and *Reconfigure* interfaces, respectively. The *CentralAggregatorC* component can be started and stopped via the *SplitControl* interface.

5.2 Reconfiguration

Using this application, four scenarios were thought of in which the quality of service of the sensor network would drop, which are listed below.

1. Recover from a crash of the aggregation component.
2. Share the load of aggregation to balance energy consumption.
3. Adjust accelerometer sensitivity when needed.
4. Decrease the sampling duty cycle when a node is low on energy.

First of all, the centralized aggregation is a weak spot: the sensor network stops completely if the particular node crashes. Secondly, this aggregation also causes an imbalance in power consumption among the nodes. When this functionality could be delegated, the network lifetime increases as the time that the first node runs out of

energy can be postponed. Thirdly, the sensitivity of the accelerometer is adjustable, ranging from $\pm 2g$ to $\pm 16g$. The optimal setting can vary at run-time, depending on the strength of the pillar and the wind. A fourth issue again relates to the energy consumption: because it is unknown if and when a change in the vibration pattern occurs, it is difficult to asses how long the WSN should run. When a node is low on energy, it could increase the time between measurements, which decreases its energy consumption in exchange for fewer measurement samples.

The following subsections provide more details on these scenarios and how they are handled by the reconfiguration middleware.

5.2.1 Restarting aggregation component

An obvious weak spot in this sensor network is the use of a central component that aggregates the peak frequencies from other nodes and determines the sample frequency for phase sampling. Assume failures of each node in the network are independent with a probability modeled by an exponential distribution with rate λ per time unit, and that failure of a node not running the aggregation component does not influence the network performance. The reliability of the network, i.e. the probability that the network will perform its intended function during a specified period of time t, will be equivalent to the reliability of the aggregating node, which is $R(t) = \Pr\{T > t\} = e^{-\lambda t}$.

However, when the central component can be restarted at any of n different nodes, the lifetime of the network is extended to the lifetime of the last crashing node. Keeping the failure model of each node the same, the reliability of the network changes to $R(t) = \sum_{k=1}^{n} \binom{n}{k} \left(e^{-\lambda t}\right)^k \left(1 - e^{-\lambda t}\right)^{n-k}$. The difference between these two reliability functions is visualized in Figure 5.4 for $\lambda = 0.01/h$ and $n = 3$. If for instance a reliability of 80% is required, reconfiguration increases the network lifetime from 22 hours to 87 hours.

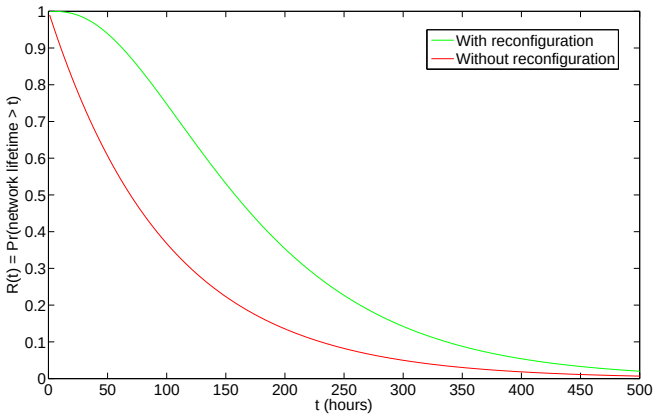


Figure 5.4: Reliability of the modal analysis network. Note that any unit of time can be used as long as it is consistent with the failure rate $\lambda = 0.01/h$.

Action	Implementation

a_1	`Central_aggregator.start();`
	`FactWrite.writeFact(central_islocal, TRUE);`
	`TimeOutCounter.stop();`

a_2	`Central_aggregator.stop();`
	`FactWrite.writeFact(central_islocal, FALSE);`
	`TimeOutCounter.start();`

Table 5.1: Pseudo-code for the actions that enable and disable the aggregation component.

Conditions		Rules	
c_1:	$central\_timeout > 10$	r_1:	$c_1 \wedge c_2 \rightarrow a_1$
c_2:	$central\_islocal == FALSE$	r_2:	$c_3 \wedge c_4 \rightarrow a_2$
c_3:	$central\_timeout < 10$		
c_4:	$central\_islocal == TRUE$		

Table 5.2: Rule base for scenario 1.

The information required for the middleware to handle this scenario, is provided in Table 5.1 and Table 5.2. The implementation of action a_1 is provided by the modal analysis application and starts a new aggregation component. This action should be called whenever the aggregation component is not functioning properly for some reason. In order for a node to detect this, it can maintain the time since the last message of the aggregation node. A time-out fact can then be inserted when the node has not received a message in the last two seconds, which can easily be used for a condition in the rule base: c_1 is satisfied if the number of time-outs is higher than 10. When the aggregation component is started, counting time-outs is no longer required. Therefore, a_1 also includes an operation to stop the counter. To prevent restarting the aggregation component at the node on which it was already enabled, c_2 requires this component to be disabled locally for the rule to be satisfied, thus completing rule r_1.

Similarly, action a_2 provides a way to disable the aggregation component when a node detects that another node already provides this functionality. As a message from the aggregation node resets the number of time-outs, c_3 and c_4 can be used in rule r_2. An election algorithm could obviously further enhance this solution to determine which of the nodes should retain its aggregation functionality.

5.2.2 Delegating aggregation component

The collection and dissemination of frequencies by the aggregation component requires additional communication, resulting in a higher energy use for the node on which the aggregation is performed. If this energy load can be divided among all

Conditions		Rules	
c_5:	$battery\_energy < 500$	r_3:	$c_4 \wedge c_5 \rightarrow a_2$

Table 5.3: Rule base addition for scenario 2.

nodes, the network lifetime, i.e. the time until the first node dies, can be prolonged substantially, without requiring an increase in battery capacity.

This reconfiguration can be achieved by simply disabling the aggregation component when the energy level drops below a certain point. The application can periodically check this using the microprocessor's voltage sensor as an indication of the remaining energy. The other nodes detect the absence of an aggregation component and reconfigure according to rule r_1. Obviously, that rule has to be changed to prevent the node from re-enabling the aggregation when it has just been disabled; this will not be considered in this case study.

As disabling the aggregation component is already provided by action a_2, this scenario only requires a new rule r_3 as provided in Table 5.3. This rule depends on a new condition c_5 requiring a certain battery level, and the already present condition c_4 of Table 5.2.

5.2.3 Adapting accelerometer sensitivity

A different problem that might arise concerns the sensitivity of the accelerometer sensor, for instance because of hardware aging, or deterioration of the mount. The analog acceleration values are converted to a 32-bit representation, with ranges of $\pm 2g$, $\pm 4g$, $\pm 8g$ or $\pm 16g$ [8]. When the sensitivity is set too low and heavy vibrations occur, the power spectral density resulting from the FFT can contain overflowed values, which makes determining the peak frequencies impossible. When overflow is detected, the sensor sensitivity should decrease to allow stronger vibrations to be detected.

Action	Implementation

a_3	`Sensor.increaseSensitivity();`
	`FactWrite.writeFact(sensor_sensitivity,`
	`Sensor.getSensitivity());`
	`FactWrite.resetFact(PROCESSOR_OVERFLOW);`

Table 5.4: Pseudo-code for the action adjusting the accelerometer sensitivity.

This scenario can easily be detected at run-time, by making the processor component record a fact each time the amplitude at the peak frequency is very high or generates an overflow. The number of overflows can subsequently be used as indication for whether reconfiguration is required or not. This is encoded in Table 5.5 by condition c_6 for rule r_3. The implementation of action a_3 as provided in Table 5.4

Conditions		Rules	
c_6:	$processor\_overflow > 5$	r_4:	$c_6 \rightarrow a_3$

Table 5.5: Rule base addition for scenario 3.

adjusts the sensitivity of the accelerometer, records this in the fact base and resets the overflow count.

Detecting the opposite case, i.e. whether the sensitivity is too low, can be done by counting the times that the amplitude at the peak frequency is equal to or nearing zero. The rest of this case is similar to a too high sensitivity and omitted for reasons of brevity.

5.2.4 Adapting sensor cycle period

The fourth scenario for reconfiguration allows a node to save power when the energy depletes more rapidly than expected. Again using the microprocessor's voltage sensor as an indication, the application can periodically check if the node can reach its target lifetime. By increasing the time between consecutive runs of the modal analysis algorithm, the energy consumption can be lowered, which prolongs the node's lifetime.

Action	Implementation

a_4	`Sampler.increasePeriod();`
	`FactWrite.writeFact(sampler_period,`
	`Sampler.getPeriod());`

Table 5.6: Pseudo-code for the action adjusting the sampling duty cycle.

Conditions		Rules	
c_7:	$sensor\_rate = 15$	r_5:	$c_6 \wedge c_7 \rightarrow a_4$

Table 5.7: Rule base addition for scenario 4.

The required operations for this reconfiguration action are provided in Table 5.6 and can be implemented by the sampling component. The action consists of adjusting the duty cycle parameter of the sampling component and updating the value in the fact base. The rule r_5 triggering this action is provided in Table 5.7. As the requirement on the energy level is already provided in condition c_6, only one additional condition c_7 is required, which asserts the duty cycle has not already been decreased.

The duty cycle of the modal analysis can be adjusted to any value: other rules similar to r_5 can be defined to provide an appropriate value for various energy levels.

In this way the developer can define duty cycle adjustments with smaller or larger steps.

5.3 Results

By incorporating the middleware with the reconfiguration rules explained above, a modal analysis application results which can reconfigure itself according to its context after deployment. The following section discusses the properties and test results of the resulting WSN. The additional memory use and processing time of the middleware are discussed to assess the additional load on a node's resources.

5.3.1 Memory Use

As mentioned in Section 4.6, memory is a scarce resource for nodes. Considering the middleware should run alongside the WSN application, it is even more important to keep the additional memory consumption limited. The size of the data structures used in the assessment are shown in Table 5.8.

Fact array	Rule array	Condition array	RuleCondition array	Fact queue
30	10	28	54	100

Table 5.8: Number of bytes in the data structures used in the case study.

The modal analysis application was compiled in different configurations to obtain the actual additional memory use of the middleware. The compilation included any optimizations performed by the nesC compiler. In Table 5.9 the RAM and ROM use are provided as reported by the TinyOS tool-chain. Considering the G-Node platform has available 8 KB of RAM and 116 KB of ROM, the middleware requires an additional 2.7% respectively 1.4% of these resources when caching is disabled, and 2.9% respectively 1.4% when caching is used.

	RAM	(extra)	ROM	(extra)
Without middleware	5333		24598	
With middleware, without caching	5554	(221)	26182	(1584)
With middleware, with caching	5572	(239)	26244	(1646)

Table 5.9: Memory requirements for modal analysis application in bytes.

The absolute increase in ROM is considerably more than the sum of the memory of each component as provided in Section 4.6.1. This can be explained by the fact that besides the middleware components, the monitoring and reconfiguration functionality provided by the application also increase the memory use. Compared to the size of the application, the middleware constitutes an increase in RAM and ROM use of 4.1% respectively 6.4% when caching is not used, and 4.5% respectively 6.7% when caching is used.

5.3.2 Processing Load

To assess the impact of the middleware on the activity of the microcontroller, a timing analysis was performed. By slightly modifying the TinyOS scheduler- and sleep components, the total time could be recorded which the microcontroller spends in each of the six supported power modes. When multiplied with the average energy consumption of the microcontroller in each power mode, the accumulated times provide a representative picture of the total energy consumption of this component [9].

The experiment ran the modal analysis application for eight hours both with and without the middleware, after which the accumulated time spent in each of the power modes was divided by the experiment's run time. In order to solely get the cost of the extra processing of monitoring and inferencing, no rules were actually fired during the experiment. The middleware was set to use caching for condition evaluations. The modal analysis algorithm was performed every fifteen seconds, with the aggregation node beaconing every two seconds.

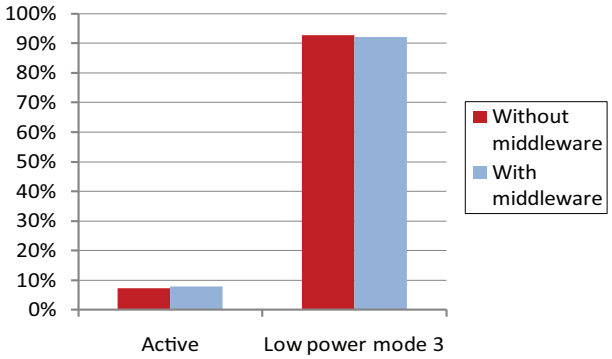


Figure 5.5: Fraction of time the microcontroller spent fully active and in low power mode 3 during the 8 hour run.

The results are depicted in Figure 5.5. The microcontroller is put in low power mode 3 for the majority of the time, namely 92.18% and 92.73% for with and without the middleware running, respectively. The microcontroller was fully turned on during the remaining 7.82% and 7.27% of the total run time. The other power modes are not used by the application. The experiment shows that the middleware increases the active time of the microcontroller by only 7.6%. Given the current ratings of 525 μA during active mode and 0.6 μA during low power mode 3 [17], this constitutes an increase of 7.4% in the energy consumption of the microcontroller.

5.4 Discussion

For the provided use case, the middleware successfully adapted the application to its context. Some issues arose during the development of the system however, which are inherent to the approach and/or could be targeted in future work.

As noted in Section 2.3, the internal system representation of facts, conditions and rules are not easily readable by humans but rely on being generated from a high-level programming language. As such, it is difficult to maintain an overview of the complete behavior of the system as the rule base grows, in case errors occur or other circumstances require detailed inspection.

The requirement of conditions to evaluate to true in order to trigger some reconfiguration action introduces some inefficiencies. For instance, the two rules used in the case study regarding the start and stop of the central aggregation component use two different conditions: c_2 checks if the component is active, while c_4 checks if the component is inactive. A method to encode this into a single condition that can evaluate to either true or false could be interesting to further increase the reuse of conditions and optimize the rule base. Extending the comparator operations on conditions can also lead to additional expressiveness and memory savings.

Caching the results of previous evaluations of conditions slightly reduces the inference time for complex rules, i.e. rules with multiple conditions and lengthy comparison operators. However, the trade-off between inference speed and memory use might favor not caching results most of the time, as a decrease of some milliseconds in inference time requires up to 25% of the total available RAM.

The inference engine uses a naive algorithm to search the rule base. However, the evaluation and case study proved that what would be regarded a large rule base can still be processed in a small time period compared to typical WSN duty cycles. The microprocessor spends only about 7% longer in active mode compared to when the middleware is not used. As this is the only source of additional energy consumption and the energy consumption of the microprocessor is very little compared to the radio, the middleware will have little effect on the total energy consumption of a sensor node. Rule bases that stretch the limits of the inference time are not realistic for typical WSN applications. Also, more computationally efficient solutions, e.g. based on the Rete algorithm, would probably reach the memory limits of many WSN platforms in these cases, as they trade off inference time for memory.

Although the memory use of the current approach also scales poorly, it is still well within the limits of the platform and leaves enough room for a complete sensing application on the nodes. Compression of the rule base might be possible during the generation of the rule base, reducing the size of facts, conditions and rules where possible. Providing monitoring input is very simple via the use of the fact base, but it can lead to storing information twice when component parameters are also part of the monitoring data. Also, reconfigurations that depend on previous state or time require explicitly providing this information to the fact base. The reconfiguration itself performs well in a practical application; getting the steps of the reconfiguration actions correctly is the most concerning issue for a sensor network developer.

CHAPTER 6

Conclusions and Future Work

Because the quality of service of WSNs largely depends on the run-time context, a method is required that encompasses this context into the design, such that a WSN is able to reconfigure after deployment. Previous work has focused on centralized (partial) reprogramming or required dense networks. The goal of this thesis project is to develop software that supports the local reconfiguration of sensor nodes at run-time, allowing a WSN to perform within its requirements under changing conditions.

This thesis proposes a middleware that represents a WSN's context on each node as a set of facts. An inference engine on each node contains rules that are based on these facts. Upon a change in the context, the inference engine checks these rules and indicates what actions should be performed by the application, such that it suits the new context.

The middleware was implemented and evaluated using a case study. The approach proved successful in reconfiguring the application to four context changes, while the evaluation showed the middleware required little additional memory and processing time compared to the application. The impact of this approach on the development of reconfigurable WSNs will be discussed next. In Section 6.2 suggestions will be given for future work on this research topic.

6.1 Discussion

The idea behind middleware for reconfigurable sensor networks is to provide WSN applications a means to cope with run-time context changes, simplify the design and implementation of adaptive sensor networks and help creating reusable components. The middleware described in this thesis improves this by providing a simple construct to monitor the context and automatically trigger reconfiguration when the context requires this. The approach improves the WSN design process as it can save on costly resources and delay certain design decisions to run-time, when more or more up-to-date information is available for reasoning about the optimal solution.

When designing distributed systems like WSNs, a developer has to take into account the changes in the context of the application. These changes can have vari-

ous causes: failure of sensor components themselves, e.g. sensor or battery aging or faulty memory; an external event, e.g. a node falling off its mount or the movement of an object for a tracking application; or possibly a change in information requirements from the user.

To deal with these variable conditions, designs traditionally incorporate additional resources, e.g. over-provisioning the available energy, using high-efficiency chips, incorporating multiple processing algorithms, or providing redundant communication links. These solutions are costly however, and do not always provide the optimal result; mostly it is chosen because it can be provided during design time and does not require developer-interaction after deployment.

However, with reconfigurable middleware, the configuration of a WSN is not invariably fixed after deployment. It allows to incorporate multiple configurations into the system and use run-time information in the design choices for a new configuration. For the case study, this can result in applying a different processing algorithm, or trading-off sample-density for network lifetime by adjusting the duty-cycle. As the WSN takes the configuration of the available resources that, according to its specification, suits its context, the highest possible quality of service can be approached in the different contexts. Reconfiguration allows for a graceful degradation of functionality as a cheaper and/or smarter alternative to achieving maximum functionality through costly over-provisioning or redundancy.

Using the reconfiguration middleware as elaborated in this thesis, rules are created that start, stop and adjust the required functionality depending on the run-time information. This way, the proposed middleware chooses the most suitable configuration of the available components for each context. This provides a structured approach to reconfiguration, allowing for design guidelines opposed to application-specific ad-hoc solutions.

6.2 Future Work

The focus of this thesis has been on providing a solution that allows certain design-time decisions to be delayed to run-time. This required representing knowledge about the WSN in a way that enables a generic middleware to reason over these decisions and reconfigure the application to its context. This leaves open a number of issues.

The internal representation of the design-time knowledge is difficult to create and maintain for developers. A high-level language can be developed which allows a more human-readable definition of the application model and requirements. This definition will subsequently be compiled to the internal representation. This will provide a better overview of the complete reconfiguration behavior and makes it easier to verify correctness, further simplifying the development process for reconfigurable WSNs. Also, the proposed solution supports state-dependent reconfigurations only by treating time as a regular monitored element in the fact base. Future work can focus on considering time as a separate concept in the middleware, which goes towards temporal reasoning about reconfiguration.

A second direction for future work to focus on is expanding the proposed approach with methods to perform distributed context monitoring and reconfiguration actions. This allows nodes to incorporate a wider view of its context and also effect coordinated reconfiguration of nodes in a certain region. The current approach already allows interaction with neighbors via the user-defined actions, but several publications have proposed solutions for information sharing between nodes, which could be used to enhance this functionality and increase efficiency [6, 40]. This can result in middleware that is able to perform simple local reconfiguration or more complex, coordinated reconfiguration if necessary.

Taking the idea of coordinated reconfiguration across multiple nodes one step further, a second level of rule compilation could be developed. This second level translates global system specifications and requirements to local rules; the previously mentioned compilation can subsequently translate these local rules to the internal representation. This additional level brings reconfiguration middleware to the research area of large-scale adaptive systems.

Finally and most importantly, leveraging the presented approach, future work can focus on extending the design methodology and patterns for reconfigurable WSNs. First works in this direction were recently proposed by Fleurey et al. [11].

Bibliography

[1] M. Albano and S. Chessa. Publish/subscribe in wireless sensor networks based on data centric storage. In *Proceedings of the 1st International Workshop on Context-Aware Middleware and Services: affiliated with the 4th International Conference on Communication System Software and Middleware (COMSWARE 2009)*, pages 37–42. ACM, 2009.

[2] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004. ISSN 1545-5971.

[3] A. Cerpa and D. Estrin. Ascent: Adaptive self-configuring sensor networks topologies. *IEEE transactions on mobile computing*, 3(3):272–285, 2004.

[4] D. Chu, L. Popa, A. Tavakoli, J.M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, page 188. ACM, 2007.

[5] Moteiv Corporation. *Tmote Sky Datasheet*, 2006. URL `http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf`.

[6] P. Costa, L. Mottola, A.L. Murphy, and G.P. Picco. TeenyLIME: transiently shared tuple space middleware for wireless sensor networks. In *Proceedings of the international workshop on Middleware for sensor networks*, pages 43–48. ACM, 2006.

[7] A. De Jong, M. Woehrle, and K. Langendoen. MoMi: model-based diagnosis middleware for sensor networks. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, pages 19–24. ACM, 2009.

[8] Analog Devices. *ADXL345 Digital Accelerometer Data Sheet*. Analog Devices, Inc., One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, USA, a edition, 2010.

[9] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 28–32. ACM, 2007.

[10] E.A. Feigenbaum, Stanford University. Dept. of Computer Science, and Stanford University. Heuristic Programming Project. *The art of artificial intelligence: I. Themes and case studies of knowledge engineering*. Computer Science Department, School of Humanities and Sciences, Stanford University, 1977.

[11] Franck Fleurey, Brice Morin, and Arnor Solberg. A model-driven approach to develop adaptive firmwares. In *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems*, SEAMS '11, pages 168–177. ACM, 2011.

[12] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem* 1. *Artificial intelligence*, 19(1):17–37, 1982.

[13] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11. ACM, 2003.

[14] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 1–14. ACM, 2009.

[15] G.P. Halkes and K.G. Langendoen. Practical considerations for wireless sensor network algorithms. *Wireless Sensor Network*, 2(6):441–446, jun 2010. ISSN 1945-3078. URL `http://www.es.ewi.tudelft.nl/papers/2010-Halkes-theory-vs-practice.pdf`.

[16] U. Hunkeler, Hong Linh Truong, and A. Stanford-Clark. MQTT-S; a publish/-subscribe protocol for wireless sensor networks. In *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, pages 791 –798, Jan. 2008.

[17] Texas Instruments. *MSP430x12x MIXED SIGNAL MICROCONTROLLER*. Texas Instruments Incorporated, Post Office Box 655303, Dallas, Texas 75265, USA, slas312c edition, September 2004.

[18] J.M. Kahn, R.H. Katz, and K.S.J. Pister. Next century challenges: mobile networking for "Smart Dust". In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278. ACM, 1999.

[19] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 254–263. ACM, 2007.

[20] S. Kogekar, S. Neema, B. Eames, X. Koutsoukos, A. Ledeczi, and M. Maroti. Constraint-guided dynamic reconfiguration in sensor networks. In *Information Processing in Sensor Networks (IPSN), 2004*, pages 379–387. IEEE, 2004.

[21] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, and J. Sztipanovits. Oasis: A programming framework for service-oriented sensor networks. In *Proceedings of the 2nd International Conference on Communication Systems Software and Middleware (COMSWARE)*, pages 1–8. IEEE, 2007.

[22] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. *ACM SIGARCH Computer Architecture News*, 30(5):85–95, 2002. ISSN 0163-5964.

[23] P. Levis and D. Gay. *TinyOS programming*. Cambridge University Press, 2009. ISBN 9780521896061.

[24] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, 35, 2005.

[25] T. Liu and M. Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118. ACM, 2003.

[26] J.P. Lynch and K.J. Loh. A summary review of wireless sensors and sensor networks for structural health monitoring. *Shock and Vibration Digest*, 38(2): 91–130, 2006.

[27] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM, 2003.

[28] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97. ACM, 2002.

[29] P. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. *Wireless Sensor Networks*, pages 212–227, 2006.

[30] A. Meier, M. Woehrle, M. Zimmerling, and L. Thiele. ZeroCal: Automatic MAC protocol calibration. *Distributed Computing in Sensor Systems*, pages 31–44, 2010.

[31] W. Munawar, M.H. Alizai, O. Landsiedel, and K. Wehrle. Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks. In *2010 International Conference on Communications (ICC)*, pages 1–6. IEEE, 2010.

[32] Michael Negnevitsky. *Artificial Intelligence: A Guide to Intelligent Systems*, chapter 2. Pearson Education Limited, 2 edition, 2002.

[33] A. Rezgui and M. Eltoweissy. Service-oriented sensor-actuator networks: Promises, challenges, and the road ahead. *Computer Communications*, 30(13): 2627–2648, 2007. ISSN 0140-3664.

[34] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner. Mires: a publish/subscribe middleware for sensor networks. *Personal and Ubiquitous Computing*, 10(1):37–44, 2006.

[35] N. Srinivas and K. Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, 2(3):221–248, 1994.

[36] SOWNet Technologies. *G-Node G301 whitepaper*. SOWNet Technologies, Delftechpark 26, 2628 XH Delft, 2010.

[37] Crossbow Technology. *Mica2 Wireless Measurement System Datasheet*, 6020-0042-04 edition, 2003. URL `https://www.eol.ucar.edu/rtf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf`.

[38] K. Terfloth, G. Wittenburg, and J. Schiller. Facts-a rule-based middleware architecture for wireless sensor networks. In *Proceedings of the 1st international conference on Communication System Software and Middleware (COMSWARE)*. Citeseer, 2006.

[39] T. Van Dam and K. Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 171–180. ACM, 2003.

[40] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation-Volume 1*, page 3. USENIX Association, 2004.

[41] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD record*, 31(3):9–18, 2002.

[42] A.T. Zimmerman, M. Shiraishi, R.A. Swartz, J.P. Lynch, et al. Automated modal parameter estimation by parallel processing within wireless monitoring systems. *Journal of Infrastructure Systems*, 14:102, 2008.

[43] H.J. Zimmermann. Fuzzy programming and linear programming with several objective functions* 1. *Fuzzy sets and systems*, 1(1):45–55, 1978.